Compiler Support for Sparse Matrix Computations

PROEFSCHRIFT
TER VERKRIJGING VAN DE GRAAD VAN DOCTOR
AAN DE RIJKSUNIVERSITEIT TE LEIDEN,
OP GEZAG VAN DE RECTOR MAGNIFICUS DR. L. LEERTOUWER,
HOOGLERAAR IN DE FACULTEIT DER GODGELEERDHEID,
VOLGENS BESLUIT VAN HET COLLEGE VAN DEKANEN
TE VERDEDIGEN OP WOENSDAG 29 MEI 1996
TE KLOKKE 15.15 UUR

DOOR

AART JOHANNES CASIMIR BIK

GEBOREN TE GOUDA IN 1969

Promotiecommissie

Promotor:	Prof. dr. H.A.G. Wijshoff	
Referenten:	Prof. dr. M.J. Wolfe	(Oregon Graduate Institute of Science and Technology, USA)
	Prof. dr. C.D. Polychronopoulos	(University of Illinois at Urbana-Champaign, USA)
Overige leden:	Prof. dr. J.N. Kok	
	Prof. dr. J. van Leeuwen	(Universiteit Utrecht)
	Prof. dr. ir. L.A. Peletier	
	Prof. dr. H.A. van der Vorst	(Universiteit Utrecht)
	Dr. P.M.W. Knijnenburg	

This research has been supported by the Netherlands Computer Science Research Foundation (SION) with funds from the Netherlands Organization for Scientific Research (NWO).

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Bik, Aart Johannes Casimir

Compiler support for sparse matrix computations / Aart Johannes Casimir Bik. - [S.l. : s.n.]. - Ill.
Thesis Rijksuniversiteit Leiden. - With ref.
ISBN 90-9009442-3
NUGI 855
Subject headings: compilers / sparse matrices / data structure transformations.

He which soweth sparingly shall reap also sparingly...

2 CORINTHIANS 9 : 6

Preface

In this dissertation, consisting of two parts, I present the results of my research in compiler support for sparse matrix computations, which has been inspired by the research proposal [218].

In the first part, preliminaries are given to keep this dissertation as self-contained as possible and to present some general purpose techniques that are useful in the second part of the dissertation. Some issues related to the implementation of loop transformations (chapter 2 and 3) have appeared in publications [33, 36, 120].

The second part of this dissertation deals with the presentation of an automatic data structure selection and transformation method. This method is used by a sparse compiler, which is a compiler capable of automatically converting a dense program into semantically equivalent sparse code [27]. First, the organization of the sparse compiler is outlined, and a brief overview of the phases of the automatic data structure selection and transformation method is given (chapter 4). Thereafter, a discussion of automatically analyzing and pre-processing the original dense program and analyzing nonzero structures of sparse matrices is given (chapters 5 and 6). These methods appeared in publications [28, 34, 38]. Moreover, the actual data structure selection and code generation method are presented (chapters 7 and 8), which have been published in [25, 30, 32, 35, 39]. Some initial experimentation indicating the potential of a sparse compiler has also been included (chapter 9). Finally, some more advanced transformations are explored (chapter 10), which have been presented in [26, 31], and conclusions and topics for future research are given (chapter 11).

Because notational conventions, definitions and methods have been altered during research, some small inconsistencies with previous publications may occur in this dissertation.

Many have contributed to this research, for which I am very grateful. In particular, I would like to thank my parents for their constant love and support.

Aart J.C. Bik

Contents

Part I

Preliminaries and Basic Results

Chapter 1

Introduction

In many fields of science and engineering, large problems are encountered that can only be solved by executing an enormous amount of floating point operations. Solving these problems in a reasonable amount of time requires substantial computing power. Moreover, the lasting desire to obtain more accurate results in less time is responsible for the fact that there will always be a demand for even higher performance. The discipline that is concerned with making the solution of these large problems possible is referred to as **high performance computing**. Amongst many innovations, two approaches that are most notable with respect to this dissertation emerged from this discipline.

First, because many architectural advances have been made to keep up with the demands for higher performance, exploiting *the specific hardware characteristics of the target machine* is extremely important. Because effectively exploiting these characteristics is a complex and cumbersome task for the programmer, so-called **restructuring compilers** have been developed to provide some support in obtaining high performance. Another, less obvious approach to keep methods to solve large problems feasible is to exploit *characteristics of the data operated upon*. In particular, many numerical applications in science and engineering operate on large sparse matrices, which are matrices with many zero elements. The storage requirements and computational time of such applications may be reduced substantially if advantage of the zero elements is taken. Storage is saved if only the nonzero elements of a sparse matrix are stored explicitly, while less computations are performed if redundant operations on zero elements are avoided. In fact, exploiting sparsity may be the only way to keep solving a problem feasible. Although exploiting sparsity may also be a complex and cumbersome task for the programmer, only limited compiler support for sparse matrix computations has been developed in the past. In this dissertation, we try to make a step towards resolving this omission by presenting a **sparse compiler** that completely supports the development of sparse matrix computations.

1.1 Exploiting Hardware Characteristics

Forced by demands for higher performance, computer designers have tried to keep up with these demands. In addition, restructuring compilers were developed to provide some support in effectively exploiting the architectural advances that have been made.

1.1.1 Architectural Advances

At the technological level, higher performance can be obtained by increasing the speed of circuits and enhancing packaging densities. Due to physical limitations on the maximum speed of electronic components, however, other means to obtain higher performance are required.

Most architectural advances are aimed either at reducing **latency**, i.e. the time between start and completion of an operation, or at increasing **bandwidth**, i.e. the width and rate of operations [103, 111][129, ch2][135][175, ch1][229][234, ch2]. A **memory hierarchy**, ranging from fast registers and a small high-speed cache to slower but larger main memory, has been introduced to reduce the average memory latency, thereby relying on the spatial and temporal locality exhibited by most programs. Memory bandwidth can be enhanced by using wider data paths (of which the switch from bit-serial to bit-parallel data paths is the most obvious example), or by introducing multiple memory paths. The memory bandwidth can be further increased by dividing memory into independent memory banks, called **memory interleaving**, where memory requests to different banks can be processed independently by these banks. Reducing execution latency usually involves technological advances that reduce the clock cycle time. The execution bandwidth (also referred to as **throughput**), can be increased by instruction pipelining, a technique in which the execution of instructions is divided in a number of stages and subsequent instructions are allowed to be simultaneously active in the different stages. In **pipelined vector processors**, a similar technique is applied to functional units, which is referred to as data pipelining. We can distinguish between memory-to-memory pipelined vector processors, where vectors stream directly from memory to pipelined functional units and back, and register-to-register pipelined vector processors, where operands and results must first be stored in vector registers. Finally, throughput can be improved by the incorporation of multiple functional units or even the duplication of complete processors to obtain a parallel computer. Although traditionally parallel computers were used to increase the throughput of multiprogrammed operating systems [194], nowadays these architectures are used more often to reduce the execution time of a single application by means of parallel processing.

At control level, we can use the taxonomy of Flynn [89] to distinguish between SISD, SIMD, or MIMD architectures. The SISD class is formed by the conventional uni-processors. In a SIMD architecture, also referred to as a **processor array**, a single control unit dispatches one instruction to an ensemble of simple processing elements that execute this instruction synchronously on different data items, where a mask must be used for conditionally executed instructions. A MIMD architecture consists of a number of asynchronously executing processors.

As illustrated in figure 1.1, at memory level, we can distinguish between message-passing (distributed memory) architectures, where each processor has its own local memory, and shared-address space architectures, where memory is shared over all processors [129, ch2]. The latter architectures can be further divided into uniform memory access (UMA) architectures, in which all memory locations are at a uniform distance, and non-uniform memory access architectures (NUMA), in which processors have their own local memory and where shared memory may or may not be present. In the latter case, access to memory of other processors is supported in hardware (in contrast, in a message-passing architecture, access to remote memory requires explicitly message passing in the code).

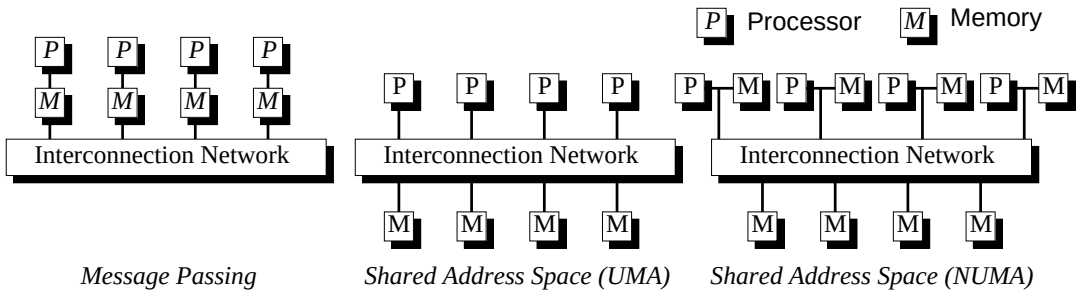


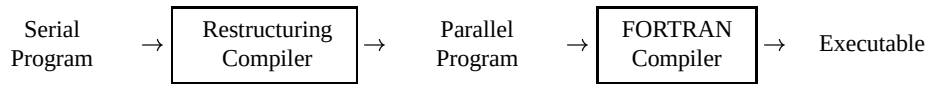
Figure 1.1: Memory Organization of Parallel Computers

Typically, interconnection networks like a bus, crossbar switch, or multistage interconnection network are used in shared-address space architectures, whereas interconnection networks like a ring, tree, mesh or hypercube are used in message-passing architectures. Shared address space and message-passing MIMD architectures are often also referred to as **multiprocessors** and **multicomputers** respectively.

1.1.2 Restructuring Compilers

Although some architectural advances remain reasonably invisible to the programmer, others must be dealt with explicitly to obtain high performance. For example, effectively exploiting the memory hierarchy requires rewriting a program to operate on small data sets that fit in cache, whereas the same program must be rewritten into a form that operates on long vectors to enhance the performance on a pipelined vector processor or processor array, where having stride-1 accesses becomes important for machines with low-order memory interleaving. Efficiently executing different iterations of a loop on a multiprocessor requires yet other program transformations, whereas re-targeting a code for a message-passing architecture requires even more programming effort, because explicit message passing must be added to the program.

Because exploiting the hardware characteristics of the target machine may be a complex and cumbersome task for the programmer, **restructuring compilers** have been developed to support this exploitation. Although many restructuring compilers focus on FORTRAN, which still is a heavily used programming language in science and engineering, the techniques used by these compilers are applicable to other imperative languages as well. After a serial program has been analyzed, a restructuring compiler performs a number of semantics preserving *program transformations* to make effective use of the specific features of the target machine where, in particular, exploiting implicit parallelism is important. To obtain high performance, the application of these program transformations must be governed by an appropriate strategy, and the problem of determining such a strategy, referred to as the **phase-ordering problem**, is still an important research topic [217]. Although, in principle, machine code could be generated directly, most restructuring compiler perform a **source-to-source translation** [147], which enables the programmer to examine the parallel program arising after program restructuring. As depicted below, a conventional compiler can be used thereafter to actually generate machine code for a particular target architecture:



Automatic program restructuring has a number of advantages. First, it enables the parallelization of existing serial software, thereby preserving the enormous investments that have been made in the past to develop this software. Furthermore, it enables programmers that are only familiar with serial programming to exploit the benefits that are offered by a particular target architecture, whereas existing tools to develop serial software can still be used. Mapping one serial program automatically to several parallel computers reduces the complexity of development and maintenance of parallel programs substantially, and offers some means to achieve portability between these architectures. Finally, automatically exploiting implicit parallelism is less error-prone and may gain insight in the constructs required in future parallel languages.

There are some severe limitations though. Because preserving the semantics of the original serial program is the most important requirement of any restructuring compiler, only conservative approximations of, for example, the data dependences arising in the program can be made. This may imply that a restructuring compiler fails to parallelize a code fragment that could be parallelized by a programmer with more knowledge about the actual data dependences.

Moreover, some serial algorithms are just not amenable to parallelization, but must be rewritten into a different semantically equivalent algorithm to allow for more parallelism. In an attempt to overcome these limitations, many restructuring compilers operate interactively, i.e. the compiler cooperates with the programmer during program restructuring.

1.2 Exploiting Data Characteristics

Exploiting the occurrence of many zero elements in large sparse matrices may yield substantial savings with respect to both the storage requirements and computational time of a numerical application. In the past, however, only limited compiler support has been developed for sparse matrix computations.

1.2.1 Sparse Matrix Computations

If many elements in a matrix are zero, then this matrix is called a **sparse matrix**. In contrast, a matrix containing many nonzero elements is referred to as a **dense matrix**. Both the storage requirements and computational time of an application that operates on sparse matrices can be reduced substantially in comparison with an application that operates on dense matrices by only storing nonzero elements and avoiding redundant operations on zero elements [70, 72, 78, 97, 169, 235].

Example: Below, two FORTRAN fragments performing the operation $\vec{b} \leftarrow \vec{b} + A\vec{x}$ are given. In the dense fragment, a two-dimensional array `A` is used to store all elements of the matrix, whereas a more complex sparse storage scheme (data structure) is used in the sparse fragment to avoid redundant operations on zero elements:

Dense Fragment:

```
REAL A(M,N)
...
DO I = 1, M
  DO J = 1, N
    B(I) = B(I) + A(I,J) * X(J)
  ENDDO
ENDDO
```

Sparse Fragment:

```
REAL    VAL_A(SZ)
INTEGER ROW_A(SZ), COL_A(SZ), NNZ_A
...
DO IJ = 1, NNZ_A
  I    = ROW_A(IJ)
  J    = COL_A(IJ)
  B(I) = B(I) + VAL_A(IJ) * X(J)
ENDDO
```

Possible contents of these storage schemes are illustrated in figure 1.2. For this example, 25 elements are stored and operated upon in the dense fragment, whereas only 5 nonzero elements are stored and operated upon in the sparse fragment. However, some additional storage, referred to as overhead storage, is required in the sparse storage scheme to reconstruct the underlying matrix. The row and column index of each nonzero element of A are stored as well, while an additional scalar records the total number of elements that are actually stored in the arrays (because some additional space may be present to allow for the insertion of more nonzero elements). Nevertheless, the total storage requirements are reduced with respect to dense storage of A (viz. 16 vs. 25 memory cells). For larger sparse matrices, more extensive savings can be expected.

Note that no substantial savings in computational time arise from protecting the loop-body of the dense fragment with the test '`(A(I,J).NE.0.0)`', because this test would still be executed `M`·`N` times. In contrast, the loop-body of the sparse fragment is only executed `NNZ_A` times. Keeping the storage requirements as well as the amount of work truly proportional to the number of nonzero elements in a sparse matrix is one of the most important objectives in sparse matrix computations [69][78, ch2][97, ch2][169, p1-3][235].

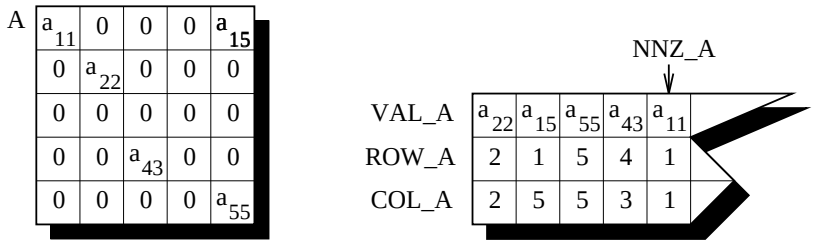


Figure 1.2: Dense Storage vs. Sparse Storage

As already illustrated by this small example, however, achieving this objective may be a complex and cumbersome task for the programmer. The use of complicated sparse storage schemes usually obscures the actual functionality of the code, making both the development and maintenance of sparse codes a non-trivial task. Furthermore, the occurrence of subscripted subscripts induced by sparse storage schemes (cf. the use of `ROW_A` and `COL_A` in the example) usually disables most compiler optimizations because the compiler must make very conservative estimates about the actual data dependences that may occur at run-time. These problems are even aggravated when sparsity preserving methods (see appendix A for a brief overview) must be incorporated in the sparse code. Despite all these problems, however, only limited compiler support for sparse matrix computations has been developed in the past. Therefore, in the next section, we propose an alternative approach to develop sparse codes.

1.2.2 Compiler Support for Sparse Matrix Computations

Because restructuring compilers are very useful to automatically detect and exploit implicit parallelism in serial software, the question arises whether it is also possible to let a restructuring compiler convert code that operates on simple data structures into a format that exploits certain characteristics of the data operated on. In contrast to conventional restructuring compilers, mainly focusing on program transformations, this approach must allow for the application of *data structure transformations* as well.

For applications involving sparse matrices, this approach implies that all computations on these matrices may simply be defined on two-dimensional arrays. A special kind of restructuring compiler, which we will refer to as a **sparse compiler**, transforms these simple data structures into more complex sparse data structures, thereby reducing storage requirements and computational time.

Analogous to the approach taken by conventional restructuring compilers, a source-to-source translation is performed. The sparse compiler automatically transforms a dense program operating on two-dimensional arrays into code that operates on sparse storage schemes. As depicted below, the resulting sparse code is compiled by a conventional FORTRAN compiler for a particular target architecture thereafter:

Besides the fact that dealing with sparsity of matrices at the compilation level rather than at the programming is less error-prone, this approach has a number of other advantages. First, the complexity of writing and maintaining sparse codes is reduced substantially, which enables programmers that are not familiar with sparse matrix computations to easily produce sparse code. Second, applying data dependence analysis to the dense code usually yields more accurate information, which allows for more program transformations.

Because the sparse compiler can account for characteristics of both the nonzero structure and the target machine (provided that these characteristics are made available in some manner), as well as the actual operations performed while selecting a suitable sparse data structure, one dense program can be converted into a range of sparse versions, each of which is tailored for a particular instance of the same problem. Program transformations may be applied to the dense program in case this data structure selection cannot be resolved efficiently. Finally, just as traditional restructuring compilers enable the re-use of existing serial software, a sparse compiler enables the re-use of parts of existing dense code.

Elaboration of these ideas have resulted in the development and implementation of a prototype sparse compiler. In this dissertation, we present the automatic data structure selection and transformation method used by this sparse compiler to automatically convert a dense program into semantically equivalent code that exploits the sparsity of data operated upon.

Chapter 2

Preliminaries

In this chapter, a brief overview of some important concepts that are used throughout this dissertation is given. In particular, concepts that are useful for program analysis and program restructuring as well as for sparse matrix computations are presented.

Systems of linear equations or inequalities in integer-valued variables and affine transformations are useful to represent many program constructs and transformations in a formal manner. In addition, many problems in science and engineering require the solution of a sparse system of linear equations. Therefore, first some preliminaries from geometry and linear algebra are given. Thereafter, a number of useful methods that are used extensively in program analysis and program restructuring are discussed.

2.1 Preliminaries from Geometry and Linear Algebra

In this section, some concepts of geometry and linear algebra are presented. For a detailed presentation, the reader is referred to [40, 42, 61, 100, 104, 153, 172, 203].

2.1.1 Cartesian Spaces

Given a fixed natural number $d \in \mathcal{N}$, the d-dimensional **Cartesian space** consists of all d-tuples $(x_1, \ldots, x_d) \in \mathcal{R}^d$. This means that each coordinate x_i in a tuple is a real number. For $d = 1$, $d = 2$, and $d = 3$, the corresponding Cartesian spaces define the Euclidean straight line, Euclidean plane, and Euclidean space, respectively, for which direct graphical interpretations exist. However, we do not restrict ourselves to these values of d, but allow for Cartesian spaces of arbitrary dimension.

Each tuple $(x_1, \ldots, x_d) \in \mathcal{R}^d$ in such a d-dimensional Cartesian space can be thought of as a point X or as the components of the position vector $OX = \vec{x}$ representing this point X, where point $O = (0, \ldots, 0)$ is referred to as the **origin**. Usually we do not distinguish between points and position vectors and simply refer to point X by means of the (position) vector $\vec{x} \in \mathcal{R}^d$.

The following operations are defined on two vectors $\vec{x}, \vec{y} \in \mathcal{R}^d$ and a scalar $\lambda \in \mathcal{R}$:

$$\left\{ \begin{array}{rcl} \vec{x} + \vec{y} & = & (\quad x_1 + y_1, \quad \ldots, \quad x_d + y_d \quad) \\ \lambda \cdot \vec{x} & = & (\quad \lambda \cdot x_1, \quad \ldots, \quad \lambda \cdot x_d \quad) \end{array} \right.$$

In these operations, vectors are expressed as row vectors. A vector can also be expressed as column vector, denoted as $\vec{x} = (x_1, \ldots, x_d)^T$ in the text for notational convenience:

$$\vec{x} = \left(\begin{array}{c} x_1 \\ \vdots \\ x_d \end{array} \right)$$

The opposite vector of a vector $\vec{x}$ is defined as $-\vec{x} = -1 \cdot \vec{x}$. This vector has the property that $-\vec{x} + \vec{x} = \vec{0}$, where $\vec{0} = (0, \ldots, 0)$. The subtraction of two vectors $\vec{x}$ and $\vec{y}$ is defined as $\vec{x} - \vec{y} = \vec{x} + (-\vec{y})$. Furthermore, the scalar product of two vectors $\vec{x}, \vec{y} \in \mathcal{R}^d$ is defined as follows:

$$\vec{x} \cdot \vec{y} = \sum_{i=1}^{d} x_i \cdot y_i \tag{2.1}$$

The vectors are **perpendicular** or **orthogonal**, denoted by $\vec{x} \perp \vec{y}$, if and only if these vectors have a zero scalar product, i.e. $\vec{x} \cdot \vec{y} = 0$. In addition, this scalar product can be used to give $\mathcal{R}^d$ the structure of a metric space by defining the following notion of distance between vectors, denoted by $d(\vec{x}, \vec{y})$:

$$d(\vec{x}, \vec{y}) = \sqrt{(\vec{x} - \vec{y}) \cdot (\vec{x} - \vec{y})} \tag{2.2}$$

A set $X \subseteq \mathcal{R}^d$ is called **bounded** if for a particular $\delta > 0$ and $\vec{x} \in X$, we have $d(\vec{x}, \vec{y}) < \delta$ for all $\vec{y} \in X$. This set X is called **unbounded** otherwise.

2.1.2 Linear and Affine Subspaces

We say that a vector $\vec{x} \in \mathcal{R}^d$ is a **linear combination** of a finite set $X = \{\vec{x}_1, \ldots, \vec{x}_k\}$ if there exist scalars $\lambda_i \in \mathcal{R}$ such that:

$$\vec{x} = \sum_{i=1}^{k} \lambda_i \cdot \vec{x}_i \tag{2.3}$$

If, additionally, $\lambda_1 + \ldots + \lambda_k = 1$, then $\vec{x}$ is called an **affine combination** of this set. A set $X = \{\vec{x}_1, \ldots, \vec{x}_k\}$ is **linearly independent** if $\lambda_1 \cdot \vec{x}_1 + \ldots + \lambda_k \cdot \vec{x}_k = \vec{0}$ implies that $\lambda_i = 0$ for all $1 \leq i \leq k$. All other sets are **linearly dependent**. A set $X' = \{\vec{x}_0, \ldots, \vec{x}_k\}$ is **affinely independent**, if the set $X = \{\vec{x}_1 - \vec{x}_0, \ldots, \vec{x}_k - \vec{x}_0\}$ is linearly independent, and **affinely dependent** otherwise. In $\mathcal{R}^d$, the sets X and X' can only be linearly and affinely independent respectively, if we have $k \leq d$.

Given a linearly independent set $X = \{\vec{x}_1, \ldots, \vec{x}_k\}$, where each $\vec{x}_i \in \mathcal{R}^d$, the set $S \subseteq \mathcal{R}^d$ consisting of all linear combinations of X forms a k-dimensional **linear subspace** of $\mathcal{R}^d$:

$$S = \{\vec{x} \in \mathcal{R}^d \mid \vec{x} = \sum_{i=1}^{k} \lambda_i \cdot \vec{x}_i\}$$

Given an affinely independent set $X' = \{\vec{x}_0, \ldots \vec{x}_k\}$, where each $\vec{x}_i \in \mathcal{R}^d$, the set $S \subseteq \mathcal{R}^d$ consisting of all affine combinations of X' forms a k-dimensional **affine subspace** of $\mathcal{R}^d$ (also called a flat):

$$S = \{\vec{x} \in \mathcal{R}^d \mid \vec{x} = \sum_{i=0}^{k} \lambda_i \cdot \vec{x}_i \text{ and } \lambda_0 + \ldots + \lambda_k = 1\}$$

Each linear subspace defined by a linearly independent set $X = \{\vec{x}_1, \ldots, \vec{x}_k\}$ is an affine subspace through the origin defined by the affinely independent set $X' = \{\vec{0}, \vec{x}_1, \ldots \vec{x}_k\}$. Hence, the dimension of linear and affine subspaces is defined consistently in this manner. Conversely, each affine subspace consists of the translate of a certain linear subspace of the same dimension.

A one-dimensional affine subspace, defined by an affinely independent set $X' = \{\vec{x}_0, \vec{x}_1\}$, is called a **straight line**. Since a linear combination $\lambda_0 \cdot \vec{x}_0 + \lambda_1 \cdot \vec{x}_1$ is an affine combination if $\lambda_0 + \lambda_1 = 1$, a line consists of all $\vec{x}$ satisfying the next equation, where $\lambda \in \mathcal{R}$:

$$\vec{x} = (1 - \lambda) \cdot \vec{x}_0 + \lambda \cdot \vec{x}_1 \tag{2.4}$$

Because X' is affine independent, the singleton set $X = \{\vec{x_1} - \vec{x}_0\}$ is linearly independent. Since this is true for $\vec{x}_0 \neq \vec{x}_1$, we see that a line is defined by two *different* points. If we also require that $0 \leq \lambda \leq 1$, then a **straight line segment** is defined. We can rewrite the previous equation into the following form, in which the line is defined by a position vector $\vec{x}_1$ and a free vector $\vec{d} = \vec{x}_1 - \vec{x}_0$, denoting the **direction** of this line:

$$\vec{x} = \vec{x}_0 + \lambda \cdot (\vec{x}_1 - \vec{x}_0) = \vec{x}_0 + \lambda \cdot \vec{d}$$

An affinely independent set $X' = \{\vec{x}_0, \vec{x}_1, \vec{x}_2\}$ defines a two-dimensional affine subspace, referred to as a **plane**, which can also be defined by a position vector and two linearly independent vectors.

2.1.3 Hyperplanes

A $(d-1)$-dimensional affine subspace $S \subseteq \mathcal{R}^d$ defined by an affinely independent set X with cardinality d is called a **hyperplane**. Alternatively, a hyperplane $S \subseteq \mathcal{R}^d$ may be defined in Cartesian form, since it consists of all $\vec{x} \in \mathcal{R}^d$ satisfying the **linear equation** $\vec{a} \cdot \vec{x} = b$ for certain fixed nonzero **normal vector** $\vec{a} \in \mathcal{R}^d$ and a scalar $b \in \mathcal{R}$:

$$S = \{\vec{x} \in \mathcal{R}^d \mid a_1 \cdot x_1 + \ldots + a_d \cdot x_d = b\}$$

In $\mathcal{R}$, $\mathcal{R}^2$, and $\mathcal{R}^3$ a hyperplane corresponds to a single point, a line, and a plane, respectively. The graphical interpretation of three linear equations is given in figure 2.1. For $d > 3$, there is no direct graphical interpretation.

The intersection of a number of hyperplanes $S_1, \ldots, S_c$ in $\mathcal{R}^d$, where each $S_i \subseteq \mathcal{R}^d$ is of the form $S_i = \{\vec{x} \in \mathcal{R}^d \mid a_{i1} \cdot x_1 + \ldots + a_{id} \cdot x_d = b_i\}$, can be easily represented by a system of (simultaneous) linear equations. Such a system can be expressed in the matrix form shown below for a $c \times d$ coefficient matrix A and a right-hand side vector $\vec{b} \in R^c$:

$$\begin{pmatrix} a_{11} & \ldots & a_{1d} \\ \vdots & \ddots & \\ a_{c1} & & a_{cd} \end{pmatrix} \begin{pmatrix} x_1 \\ \\ \vdots \\ \\ x_d \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ b_c \end{pmatrix}$$

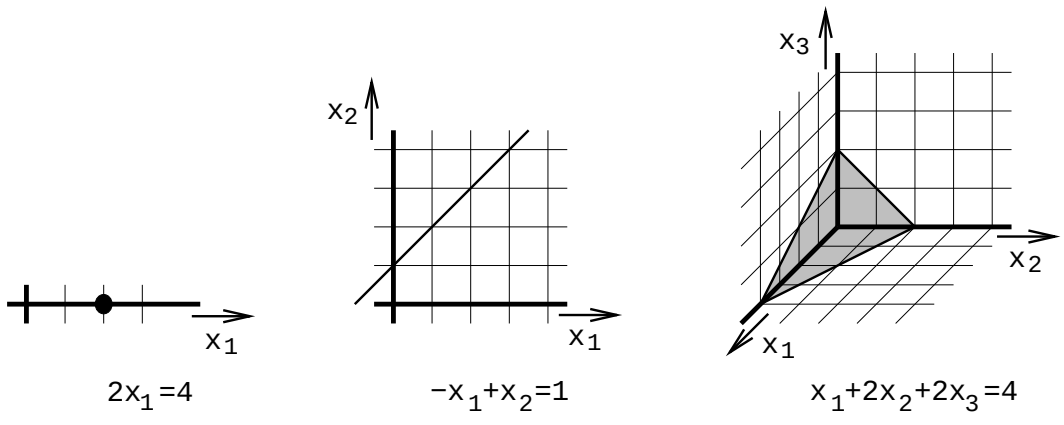


Figure 2.1: Graphical Interpretation of Linear Equations

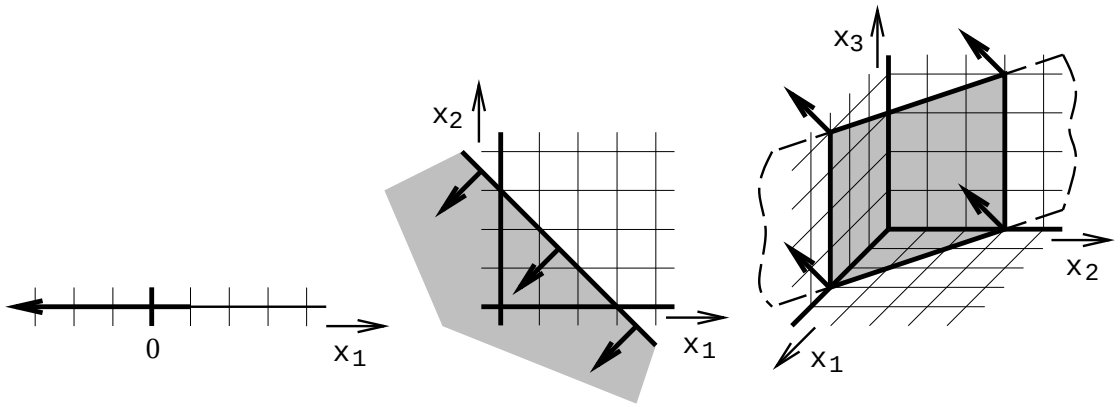


Figure 2.2: Half-Spaces in $\mathcal{R}$, $\mathcal{R}^2$ and $\mathcal{R}^3$

Usually, the variables are omitted and the system is represented by the column augmented matrix $(A \mid \vec{b})$. The system is called **homogeneous** if $\vec{b} = \vec{0}$ and **in-homogeneous** otherwise. A vector $\vec{x} \in \mathcal{R}^d$ that satisfies all equations in this system simultaneously is called a **solution** of this system. The set $S \subseteq \mathcal{R}^d$ of all solutions forms the intersection of the hyperplanes:

$$S = S_1 \cap \ldots \cap S_c$$

If $\text{rank}(A \mid \vec{b}) > \text{rank}(A)$, then this intersection is empty, and the system is called **inconsistent**. If $\text{rank}(A \mid \vec{b}) = \text{rank}(A)$, then the system is called **consistent** and the intersection of hyperplanes forms a $(d - \text{rank}(A))$-dimensional affine subspace of $\mathcal{R}^d$. In this case, there may be a unique solution, or there may be infinitely many solutions.

A system with $c = 2$ and $d = 2$, for instance, represents two lines. These lines may be parallel, coinciding or intersecting, corresponding to an inconsistent system, or a consistent system with infinitely many solutions or a unique solution, respectively.

2.1.4 Half-Spaces

A set $H \subseteq \mathcal{R}^d$ consisting of all $\vec{x} \in \mathcal{R}^d$ satisfying a **linear inequality** $\vec{a} \cdot \vec{x} \leq b$ for a fixed nonzero vector $\vec{a} \in \mathcal{R}^d$ and a scalar $b \in \mathcal{R}$ is called a **closed half-space** in $\mathcal{R}^d$:

$$H = \{\vec{x} \in \mathcal{R}^d \mid a_1 \cdot x_1 + \ldots + a_d \cdot x_d \leq b\}$$

If a strict inequality $\vec{a} \cdot \vec{x} < b$ is used in this definition, H is called an **open half-space**. The complement of a closed half-space $H = \{\vec{x} \in \mathcal{R}^d \mid \vec{a} \cdot \vec{x} \leq b\}$, denoted by $\overline{H}$, is defined as the open half-space $\overline{H} = \{\vec{x} \in \mathcal{R}^d \mid \vec{a} \cdot \vec{x} > b\}$. Note that we can always convert an inequality with '$\leq$' or '$<$' into a form with a '$\geq$' or '$>$' and vice versa by multiplying the inequality by -1. Similarly, the complement of an open half-space is formed by a closed half-space. A half-space H and its complement $\overline{H}$ partition $\mathcal{R}^d$ into two disjoint sets, since $H \cup \overline{H} = \mathcal{R}^d$ and $H \cap \overline{H} = \emptyset$.

Obviously, each closed half-space consists of all points on one side of a hyperplane. For example, in figure 2.2 we show the closed half-spaces in $\mathcal{R}$, $\mathcal{R}^2$, and $\mathcal{R}^3$ that are defined by the inequalities $x_1 \leq 1$, $x_1 + x_2 \leq 3$, and $x_1 + x_2 \leq 3$ respectively. In these cases, the hyperplanes defined by the equations $x_1 = 1$, $x_1 + x_2 = 3$ and $x_1 + x_2 = 3$ correspond to a point, line, and a plane parallel to the x_3-axis, respectively. In $\mathcal{R}$ and $\mathcal{R}^2$ the corresponding closed half-space is referred to as a half-line and half-plane respectively.

A set $S \subseteq \mathcal{R}^d$ is **convex** if and only if for each $\vec{x}, \vec{y} \in S$, we have $\lambda \cdot \vec{x} + (1 - \lambda) \cdot \vec{y} \in S$ for all $0 \leq \lambda \leq 1$ as well, i.e. all points on a line segment with arbitrary end-points $\vec{x}$ and $\vec{y}$ in S are also contained in this set.

Each half-space is convex. Because the intersection of a number of convex sets is also convex, the intersection of a number of half-spaces is a convex set.

Any set $PS \subseteq \mathcal{R}^d$ consisting of the intersection of a *finite* number of closed half-spaces $H_1, \ldots, H_c$ in $\mathcal{R}^d$ is called a **polyhedral set**:

$$PS = \bigcap_{i=1}^{c} H_i$$

A *bounded* polyhedral set forms a **convex polytope**, called a line segment, convex polygon, and convex polyhedron in $\mathcal{R}$, $\mathcal{R}^2$, and $\mathcal{R}^3$ respectively.

A half-space $H \subseteq \mathcal{R}^d$ for which the equality $PS \cap H = PS$ holds is called **redundant** with respect to a polyhedral set $PS \subseteq \mathcal{R}^d$. The following obvious property can be used to detect redundant half-spaces:

Proposition 2.1 *A half-space H is redundant with respect to a polyhedral set PS if and only if the equation $PS \cap \overline{H} = \emptyset$ holds.*

Because a polyhedral set $PS \subseteq \mathcal{R}^d$ is defined by the intersection of a finite number of closed half-spaces, a convenient representation of PS consists of a system of linear inequalities. Assuming that PS is defined by the closed half-spaces $H_1, \ldots, H_c$ in $\mathcal{R}^d$, where each $H_i \subseteq \mathcal{R}^d$ gives rise to a linear inequality $a_{i1} \cdot x_1 + \ldots + a_{id} \cdot x_d \leq b_i$, the polyhedral set can be represented by a system of linear inequalities $A\vec{x} \leq \vec{b}$:

$$\begin{pmatrix} a_{11} & \ldots & a_{1d} \\ \vdots & \ddots & \\ a_{c1} & & a_{cd} \end{pmatrix} \begin{pmatrix} x_1 \\ \\ \vdots \\ \\ x_d \end{pmatrix} \leq \begin{pmatrix} b_1 \\ \vdots \\ b_c \end{pmatrix}$$

Analogous to the representation of a system of linear equations, we will frequently represent this system of linear inequalities by a column augmented matrix $(A \mid \vec{b})$.

A vector $\vec{x} \in \mathcal{R}^d$ that satisfies all inequalities in $A\vec{x} \leq \vec{b}$ simultaneously is called a solution of the system. A system of linear inequalities is called consistent if at least one solution exists. The system is called inconsistent otherwise. Obviously, the set of solutions forms a polyhedral set $PS \subseteq \mathcal{R}^d$.

Example: In figure 2.3 we show a convex polyhedron formed by the intersection of the half-spaces defined by the following system of linear inequalities:

$$\begin{pmatrix} 0 & 0 & -1 \\ -1 & 0 & 0 \\ 0 & -1 & 0 \\ 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \leq \begin{pmatrix} 0 \\ -2 \\ -2 \\ 8 \end{pmatrix}$$

A closed half-space defined by an inequality like $x_3 \leq 4$ would be redundant with respect to this polyhedron, because the intersection between the polyhedron and the open half-space defined by $x_3 > 4$ is empty. Indeed, the closed half-space defined by $x_3 \leq 4$ does not contribute to the shape of the polyhedron.

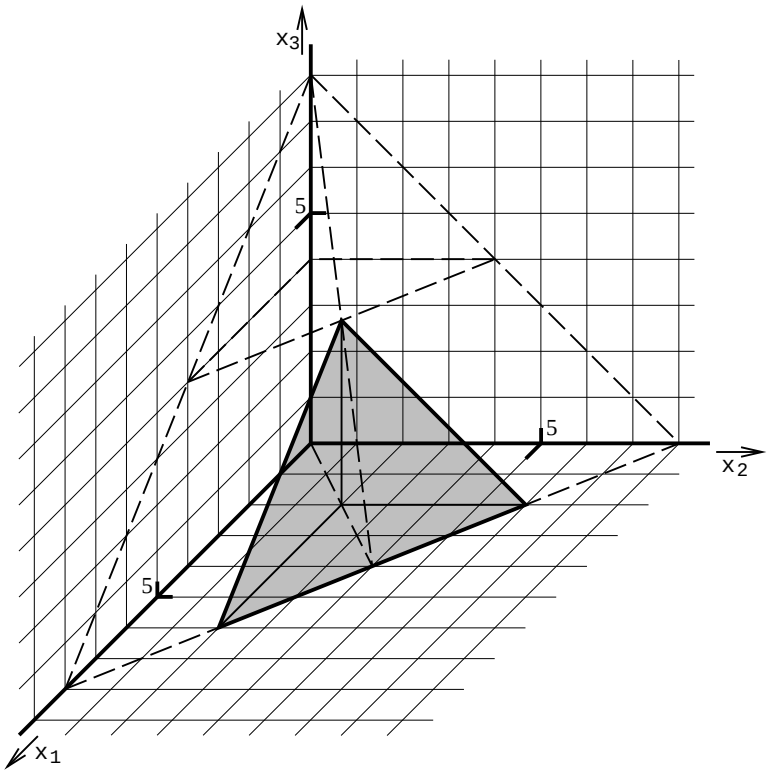


Figure 2.3: Convex Polyhedron

2.1.5 Linear and Affine Transformations

A mapping $F : \mathcal{R}^d \to \mathcal{R}^c$ having the following properties for all $\vec{x}, \vec{y} \in \mathcal{R}^d$ and $\lambda \in \mathcal{R}$, is called a **linear transformation**:

$$\begin{cases} F(\vec{x}+\vec{y}) &= F(\vec{x}) + F(\vec{y}) \\ F(\lambda \cdot \vec{x}) &= \lambda \cdot F(\vec{x}) \end{cases}$$

Each linear transformation satisfies $F(\vec{0}) = \vec{0}$ and $F(-\vec{x}) = -F(\vec{x})$. Furthermore, each linear transformation $F : \mathcal{R}^d \to \mathcal{R}^c$ can be expressed in matrix form as $F(\vec{x}) = W\vec{x}$, where W is a $c \times d$ matrix. The **kernel** of a linear transformation $F : \mathcal{R}^d \to \mathcal{R}^c$, forming a linear subspace of $\mathcal{R}^d$, is defined as follows:

$$\ker F = \{\ \vec{x} \in \mathcal{R}^d \mid F(\vec{x}) = \vec{0}\ \}$$

A mapping $F : \mathcal{R}^d \to \mathcal{R}^c$ which can be expressed as $F(\vec{x}) = \vec{v} + W\vec{x}$, for a $c \times d$ matrix W and a constant vector $\vec{v} \in \mathcal{R}^c$, is called an **affine transformation**. Hence, linear transformations are formed by affine transformations having $\vec{v} = \vec{0}$.

A $d \times d$ matrix U with integer elements that satisfies $\det(U) = \pm 1$ is called a **unimodular matrix**. A linear transformation $F : \mathcal{R}^d \to \mathcal{R}^d$ that can be expressed as $F(\vec{x}) = U\vec{x}$ for a unimodular matrix U is called a **unimodular transformation**. Because the product of two unimodular matrices is also unimodular (viz. UU' is an integer matrix and $\det(UU') = \det(U) \cdot \det(U') = \pm 1$), unimodular transformations are closed under composition. A unimodular transformation maps a discrete point $\vec{x} \in \mathcal{Z}^d$ to a discrete point $\vec{y} = U\vec{x} \in \mathcal{Z}^d$. Moreover, since U^{-1} exists and is also a unimodular matrix, each discrete point $\vec{y} \in \mathcal{Z}^d$ in the image of a polyhedral set PS under F uniquely corresponds to a discrete point $\vec{x} \in PS$ according to the equation $\vec{x} = U^{-1}\vec{y}$.

2.2 Some Useful Methods

In this section, we discuss some methods that are used extensively in this dissertation.

2.2.1 Extended Euclidean Algorithm

The **greatest common divisor** of the integers $\alpha_1, \ldots, \alpha_d$, denoted by $g = \gcd(\alpha_1, \ldots, \alpha_d)$, is the greatest (positive) integer dividing all these integers (for all i, $\alpha_i \bmod g = 0$).

Below, we present an implementation of the extended euclidean algorithm in pseudo-code (cf. [18][100, p199-202][122, p14] [229, p93-96][234, p141]). Given the integers α_1 and α_2, this algorithm computes the greatest common divisor $g = \gcd(\alpha_1, \alpha_2)$ and yields two other integers x and y satisfying $\alpha_1 \cdot x + \alpha_2 \cdot y = g$ as a side-effect:

```
integer function gcd(a1, a2, var x, var y)
begin
  c1 := abs(a1); c2 := abs(a2);
  x1 := 1;       x2 := 0;
  while (c2 > 0) do
    x1 := x1 - ⌊c1 / c2⌋ * x2;
    c1 := c1 - ⌊c1 / c2⌋ * c2;
    swap(x1, x2);
    swap(c1, c2);
  enddo
  gcd := c1;
  x   := (a1 == 0) ? 0 : ( (a1 > 0) ?  x1 : -x1);
  y   := (a2 == 0) ? 0 : ( (c1 - a1 * x)  / a2 );
end
```

This function can also be used to compute the greatest common divisor of a number of integers by repetitively using the following equation for $d \geq 3$:

$$\gcd(\alpha_1, \ldots, \alpha_d) = \gcd(\alpha_1, \gcd(\alpha_2, \ldots, \alpha_d))$$

These integers are called **relatively prime** if $\gcd(\alpha_1, \ldots, \alpha_d) = 1$.

2.2.2 Completion Method for Unimodular Matrices

Unimodular transformations provide a convenient representation of some loop transformations, as is discussed further in chapter 3. The following completion method of a unimodular matrix U of which only the first row is specified will be useful to construct a loop transformation that satisfies a particular goal. In addition, because U^{-1} is required to implement this loop transformation, we also present a method to construct this inverse simultaneously [29].

Conventional Completion Method

Given an arbitrary vector $\vec{\alpha} \in \mathcal{Z}^d$ of which the components are relatively prime, a unimodular matrix of the following form exists [19, p55-59][159, p13-15]:

$$U = \begin{pmatrix} \alpha_1 & \ldots & \alpha_d \\ u_{21} & \ldots & u_{2d} \\ \vdots & \ddots & \\ u_{d1} & & u_{dd} \end{pmatrix}$$

The construction of the desired integer matrix is based on the fact that, given a $k \times k$ integer matrix U_k with $|\det(U_k)| = g_k$, where $g_k = \gcd(\alpha_1, \ldots, \alpha_k)$, another $(k+1) \times (k+1)$ integer matrix U_{k+1} with $|\det(U_{k+1})| = g_{k+1}$ can be constructed from U_k in a relatively easy manner.

Hence, if $\alpha_1 \neq 0$, then the following sequence can be constructed, in which the final matrix is the desired matrix with $|\det(U)| = g_d = 1$:

$$(\alpha_1) = U_1 \rightarrow U_2 \rightarrow \ldots \rightarrow U_d = U$$

Unfortunately, the completion method cannot always be initiated for $k = 1$, since a prefix of zeros may appear in $\vec{\alpha}$. However, for any $\vec{\alpha} \in \mathcal{Z}^d$ with $\gcd(\alpha_1, \ldots, \alpha_d) = 1$, an m exists such that $\alpha_i = 0$ for all $1 \leq i < m$ and $\alpha_m \neq 0$. Hence, in general, the completion method can be initiated for $k = m$ with the following $m \times m$ matrix U_m satisfying $\det(U_m) = (-1)^{m+1} \cdot \alpha_m$, which implies that $|\det(U_m)| = g_m$:

$$U_m = \begin{pmatrix} 0 & \ldots & 0 & \alpha_m \\ 1 & & & 0 \\ & \ddots & & \vdots \\ & & 1 & 0 \end{pmatrix} \tag{2.5}$$

Now, suppose that a $k \times k$ integer matrix U_k has been constructed with $(\alpha_1, \ldots \alpha_k)$ as first row and $|\det(U_k)| = g_k$. First, two integers γ and β must be determined such that the following equation holds:

$$g_k \cdot \gamma - \alpha_{k+1} \cdot \beta = g_{k+1}$$

These integers are obtained as a side-effect of the extended euclidean algorithm, if we compute $g_{k+1} = \gcd(g_k, \alpha_{k+1})$ as $\gcd(g_k, -\alpha_{k+1})$. The next $(k+1) \times (k+1)$ integer matrix U_{k+1} in the sequence with $(\alpha_1, \ldots, \alpha_{k+1})$ as first row can be easily obtained by extending the previous matrix as follows, in which all divisions evaluate to integer values:

$$U_{k+1} = \begin{pmatrix} & & & \alpha_{k+1} \\ & & & 0 \\ & U_k & & \vdots \\ & & & 0 \\ \frac{\alpha_1 \cdot \beta}{g_k} & \ldots & \frac{\alpha_k \cdot \beta}{g_k} & \gamma \end{pmatrix} \tag{2.6}$$

Expansion of the determinant of this matrix by the last column reveals that the next equation holds, where the $k \times k$ matrix E_k denotes the matrix that is obtained after eliminating the first row and last column of U_{k+1}:

$$\det(U_{k+1}) = (-1)^{k+2} \cdot \alpha_{k+1} \cdot \det(E_k) + \gamma \cdot \det(U_k)$$

Consequently, E_k can be written as the following product:

$$E_k = \begin{pmatrix} 0 & 1 & & \\ \vdots & & \ddots & \\ 0 & & & 1 \\ \frac{\beta}{g_k} & 0 & \ldots & 0 \end{pmatrix} U_k$$

Because $\det(E_k) = \left((-1)^{k+1} \cdot \beta / g_k\right) \cdot \det(U_k)$ and for $k \geq 1$ the expression $(-1)^{2k+3}$ is equal to -1, the following equations hold:

$$\det(U_{k+1}) = \frac{\det(U_k)}{g_k} \cdot (g_k \cdot \gamma - \alpha_{k+1} \cdot \beta) = \frac{\det(U_k)}{g_k} \cdot g_{k+1}$$

Since either $\det(U_k) = g_k$ or $\det(U_k) = -g_k$ holds, an integer matrix U_{k+1} is constructed that satisfies the following equation:

$$|\det(U_{k+1})| = g_{k+1}$$

Because the components of $\vec{\alpha} \in \mathcal{Z}^d$ are relatively prime, repetitive extending the matrix in this manner eventually results in a unimodular matrix $U = U_d$ with $|\det(U)| = 1$.

In [18, 159, 224], the case $d = 2$ is considered separately. In order to obtain a unimodular 2×2 matrix U with (α_1, α_2) as first row, the integers u_{21} and u_{22} are required such that $\det(U) = \alpha_1 \cdot u_{22} - \alpha_2 \cdot u_{21}$ is either $+1$ or -1:

$$U = \begin{pmatrix} \alpha_1 & \alpha_2 \\ u_{21} & u_{22} \end{pmatrix}$$

The extended euclidean algorithm can be used to construct this matrix directly, since if the greatest common divisor $\gcd(\alpha_1, \alpha_2)$ is computed as $\gcd(\alpha_1, -\alpha_2)$, then the integers u_{22} and u_{21} satisfying $\alpha_1 \cdot u_{22} + (-\alpha_2) \cdot u_{21} = \gcd(\alpha_1, \alpha_2) = 1$ are obtained as a side-effect. The inverse of such a 2×2 unimodular matrix can be easily obtained by using the following equation:

$$U^{-1} = \frac{1}{\det(U)} \cdot \begin{pmatrix} u_{22} & -\alpha_2 \\ -u_{21} & \alpha_1 \end{pmatrix}$$

Computation of U^{-1} is not so straightforward in general. However, because the inverse of the matrix is required to implement a loop transformation defined by U, we present an efficient method to construct U^{-1} simultaneously with the completion of U in the following section.

Extended Completion Method

If a matrix A is modified into $A + \Delta A$, where the modification can be expressed as $\Delta A = VSW^T$, then the inverse of the modified matrix $A + \Delta A$ can be obtained from the inverse of A using the matrix modification formula [78, p243-244]:

$$(A + VSW^T)^{-1} = A^{-1} - \underbrace{A^{-1}V(S^{-1} + W^T A^{-1} V)^{-1} W^T A^{-1}}_{\text{Correction Term}} \tag{2.7}$$

This formula provides a convenient method to derive the changes in the inverse of the original matrix that are required to obtain the inverse of the modified matrix. Hence, since for $m \leq k \leq d$, we have $\det(U_k) \neq 0$, the question arises whether the matrix modification formula can be used to construct $U_m^{-1} \rightarrow \ldots \rightarrow U_d^{-1}$ simultaneously with the construction of $U_m \rightarrow \ldots \rightarrow U_d$ during the completion method. In this manner, the completion method also yields the inverse of the desired matrix $U = U_d$.

A subtlety that must be dealt with is the fact that each modification changes the order of the matrix. However, we can view each extension of U_k into U_{k+1} as a modification of a matrix A into $A + \Delta A$, where A and, hence, A^{-1} are defined as follows:

$$A = \begin{pmatrix} U_k & \\ & 1 \end{pmatrix} \qquad A^{-1} = \begin{pmatrix} U_k^{-1} & \\ & 1 \end{pmatrix}$$

The modification $\Delta A = VSW^T$ can be expressed as follows (cf. formula (2.6)):

$$V = \begin{pmatrix} 1 & 0 \\ 0 & \vdots \\ \vdots & 0 \\ 0 & 1 \end{pmatrix} \quad S = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \quad W^T = \begin{pmatrix} 0 & \dots & 0 & \alpha_{k+1} \\ \frac{\alpha_1 \cdot \beta}{g_k} & \dots & \frac{\alpha_k \cdot \beta}{g_k} & \gamma - 1 \end{pmatrix}$$

The expression $W^T A^{-1}$ in the correction term defined by (2.7) has the following form, in which we have used the fact that the product $(\alpha_1, \dots, \alpha_k)U_k^{-1}$ yields the first row of the $k \times k$ identity matrix:

$$W^T A^{-1} = \begin{pmatrix} 0 & \dots & 0 & \alpha_{k+1} \\ \frac{\alpha_1 \cdot \beta}{g_k} & \dots & \frac{\alpha_k \cdot \beta}{g_k} & \gamma - 1 \end{pmatrix} \begin{pmatrix} U_k^{-1} & \\ & 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 & \dots & 0 & \alpha_{k+1} \\ \frac{\beta}{g_k} & 0 & \dots & 0 & \gamma - 1 \end{pmatrix}$$

Hence, expression $(S^{-1} + W^T A^{-1} V)$ has the following form:

$$(S^{-1} + W^T A^{-1} V) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} + \begin{pmatrix} 0 & \alpha_{k+1} \\ \frac{\beta}{g_k} & \gamma - 1 \end{pmatrix} = \begin{pmatrix} 1 & \alpha_{k+1} \\ \frac{\beta}{g_k} & \gamma \end{pmatrix}$$

Because the determinant of the resulting matrix is $\gamma - (\alpha_{k+1} \cdot \beta)/g_k$, which can be rewritten into $(g_k \cdot \gamma - \alpha_{k+1} \cdot \beta)/g_k = g_{k+1}/g_k$, the inverse of this matrix is defined as follows:

$$(S^{-1} + W^T A^{-1} V)^{-1} = \frac{g_k}{g_{k+1}} \cdot \begin{pmatrix} \gamma & -\alpha_{k+1} \\ -\frac{\beta}{g_k} & 1 \end{pmatrix}$$

Equation $-\alpha_{k+1} \cdot \beta = g_{k+1} - g_k \cdot \gamma$ implies that $A^{-1} V (S^{-1} + W^T A^{-1} V)^{-1} W^T A^{-1}$ has the following form, in which elements of U_k^{-1} are denoted as $\overline{u}_{ij}$:

$$\frac{g_k}{g_{k+1}} \cdot \begin{pmatrix} \overline{u}_{11} & 0 \\ \vdots & \vdots \\ \overline{u}_{k1} & 0 \\ 0 & 1 \end{pmatrix} \underbrace{\begin{pmatrix} \gamma & -\alpha_{k+1} \\ -\frac{\beta}{g_k} & 1 \end{pmatrix} \begin{pmatrix} 0 & 0 & \dots & 0 & \alpha_{k+1} \\ \frac{\beta}{g_k} & 0 & \dots & 0 & \gamma - 1 \end{pmatrix}}_{\begin{pmatrix} \frac{g_{k+1}}{g_k} - \gamma & 0 & \dots & 0 & \alpha_{k+1} \\ \frac{\beta}{g_k} & 0 & \dots & 0 & \frac{g_{k+1}}{g_k} - 1 \end{pmatrix}}$$

Hence, the following correction term must be subtracted from the inverse of A to obtain the inverse of $A + \Delta A$:

$$\begin{pmatrix} (1 - \frac{\gamma \cdot g_k}{g_{k+1}}) \cdot \overline{u}_{11} & 0 & \dots & 0 & \frac{\alpha_{k+1} \cdot g_k}{g_{k+1}} \cdot \overline{u}_{11} \\ \vdots & \vdots & \ddots & & \vdots \\ (1 - \frac{\gamma \cdot g_k}{g_{k+1}}) \cdot \overline{u}_{k1} & 0 & \dots & 0 & \frac{\alpha_{k+1} \cdot g_k}{g_{k+1}} \cdot \overline{u}_{k1} \\ \frac{\beta}{g_{k+1}} & 0 & \dots & 0 & 1 - \frac{g_k}{g_k + 1} \end{pmatrix} \tag{2.8}$$

Since most elements remain unaffected by the correction term, the matrix modification formula reveals a convenient method to construct $U_m^{-1} \to \dots \to U_d^{-1}$ simultaneously with the construction of the sequence $U_m \to \dots \to U_d$.

One final difficulty that must be dealt with is that, since we have $|\det(U_k)| = g_k$, and $g_k \neq 1$ may hold for $m \leq k < d$, matrices in this sequence are not necessarily unimodular. Hence, fractions may appear in some U_k^{-1}.

As is shown below, each matrix U_k^{-1} can be represented as $(1/\alpha_m) \cdot \tilde{U}_k$ for a $k \times k$ integer matrix $\tilde{U}_k$. Therefore, even the sequence $U_m^{-1} \rightarrow \ldots \rightarrow U_d^{-1}$ can be constructed with only integer arithmetic.

The completion method is initiated with the matrix U_m defined in (2.5) and we can represent the inverse of this matrix as follows, where $\alpha_m \neq 0$:

$$U_m^{-1} = \frac{1}{\alpha_m} \cdot \tilde{U} = \frac{1}{\alpha_m} \cdot \begin{pmatrix} 0 & \alpha_m & & \\ \vdots & & \ddots & \\ 0 & & & \alpha_m \\ 1 & 0 & \ldots & 0 \end{pmatrix}$$

Thereafter, as defined by the correction term (2.8), a representation $(1/\alpha_m) \cdot \tilde{U}_{k+1}$ of U_{k+1}^{-1} is obtained from the representation $(1/\alpha_m) \cdot \tilde{U}_k$ of U_k^{-1} as follows, where elements of $\tilde{U}_k$ are denoted as $\tilde{u}_{ij}$ and where $p = \gamma \cdot (g_k/g_{k+1})$ and $q = -\alpha_{k+1} \cdot (g_k/g_{k+1})$:

$$U_{k+1}^{-1} = \frac{1}{\alpha_m} \begin{pmatrix} p \cdot \tilde{u}_{11} & \tilde{u}_{12} & \ldots & \tilde{u}_{1k} & q \cdot \tilde{u}_{11} \\ \vdots & \vdots & \ddots & & \vdots \\ p \cdot \tilde{u}_{k1} & \tilde{u}_{k2} & & \tilde{u}_{kk} & q \cdot \tilde{u}_{k1} \\ -\frac{\alpha_m \cdot \beta}{g_{k+1}} & 0 & \ldots & 0 & \frac{\alpha_m \cdot g_k}{g_{k+1}} \end{pmatrix}$$

Because g_{k+1} divides both g_k and α_m by construction, all divisions evaluate to integer values. Moreover, because $|\det(U)| = 1$ holds for the final matrix $U = U_d$, dividing all elements of $\tilde{U}_d$ by α_m yields an integer matrix that is equal to the inverse of this matrix.[1]

Example: Below, the successive steps for the construction of a 4×4 unimodular matrix U with first row $(8, 6, 4, 1)$ and the inverse U^{-1} are illustrated:

$$\begin{array}{ll} k = 1 & (8) \qquad \frac{1}{8} \cdot (1) \\ k = 2 & \begin{pmatrix} 8 & 6 \\ 1 & 1 \end{pmatrix} \qquad \frac{1}{8} \cdot \begin{pmatrix} 4 & -24 \\ -4 & 32 \end{pmatrix} \\ k = 3 & \begin{pmatrix} 8 & 6 & 4 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad \frac{1}{8} \cdot \begin{pmatrix} 4 & -24 & -16 \\ -4 & 32 & 16 \\ 0 & 0 & 8 \end{pmatrix} \\ k = 4 & \begin{pmatrix} 8 & 6 & 4 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -4 & -3 & -2 & 0 \end{pmatrix} \qquad \frac{1}{8} \cdot \begin{pmatrix} 0 & -24 & -16 & -8 \\ 0 & 32 & 16 & 8 \\ 0 & 0 & 8 & 0 \\ 8 & 0 & 0 & 16 \end{pmatrix} \end{array}$$

Because $|\det(U_k)| = \gcd(\alpha_1, \ldots, \alpha_k)$ holds for all $1 \leq k \leq 4$, the first three matrices are not unimodular (viz. $\gcd(8, 6, 4) = 2$). Therefore, the division by 8 required in the inverse matrices yields integer elements in the last matrix only.

Some experiments on an HP 9000/720 indicate that the extended completion method can be implemented more efficiently than explicit construction of the inverse of a unimodular matrix after application of the conventional completion method [29]. However, because in practice the size of each matrix is limited by the maximum nesting depth of loops in a program, only a slight reduction in execution time may be expected.

2.2.3 Solving a System of Linear Equations

In this section, we briefly glance at methods to solve a system of linear equations.

[1] In fact, these divisions can already be performed at any step k for which $\gcd(\alpha_1, \ldots, \alpha_k) = 1$.

Elementary Row and Column Operations

There are three **elementary row (column) operations**: (i) multiplying a row (column) of a matrix by a nonzero scalar, (ii) adding an arbitrary multiple of one row (column) of a matrix to another row (column), and (iii) interchanging two rows (columns) of a matrix. Applying one elementary row (column) operation to the identity matrix yields a so-called **elementary matrix**.

Applying an elementary row operation to a $c \times d$ matrix A is equivalent to pre-multiplying this matrix with the corresponding elementary $c \times c$ matrix E (i.e. computing EA, where E is obtained by applying the elementary row operation to the $c \times c$ identity matrix). Likewise, applying an elementary column operation to a $c \times d$ matrix A is equivalent to post-multiplying this matrix with the corresponding elementary matrix E (i.e. computing AE, where E is obtained by applying the elementary column operation to the $d \times d$ identity matrix). Hence, if the matrix A' is obtained from another matrix A by applying m elementary row operations represented by $R_1, \ldots, R_m$ and l elementary column operations represented by $C_1, \ldots, C_l$, then A' can be written as follows:

$$A' = R_m \ldots R_1 A C_1 \ldots C_l$$

For integer matrices, we limit ourselves to the following elementary *integer* row or column operations: (i) multiplying a row or column by -1 (reversal), (ii) adding an integer multiple of a row or column to another row or column (skewing), and (iii) interchanging two rows or columns. The corresponding elementary matrices are unimodular. Hence, applying any finite sequence of elementary integer row or column operations to an integer matrix is equivalent to either pre- or post-multiplying the matrix with a unimodular matrix, formed by the product of the corresponding elementary matrices [19, p26-31].

Systems of Linear Equations

Repetitive application of elementary row or column operations to the column augmented matrix $(A \mid \vec{b})$ representation of a system of linear equation $A\vec{x} = \vec{b}$ can be used to convert this system into an equivalent system (i.e. a system with the same solution set) whose solutions are easier to determine. For instance, elementary row operations can be used to convert a column augmented matrix into a matrix that is in **echelon form**. This means that in each row the column index of the first nonzero element is greater than the column index of the first nonzero element in preceding rows, and all zero rows appear last. In appendix A, the situation where A is a square non-singular matrix (i.e. $\det A \neq 0$) is considered. In this case, converting $(A \mid \vec{b})$ into echelon form is equivalent to converting A into upper triangular form. We also discuss how symmetry or sparsity of A can be exploited to reduce the storage requirements and computational time of the solution method.

The following proposition can be used to solve an *integer* system of linear inequalities with integer-valued variables (so-called linear diophantine equations) [19, p59-66]:

Proposition 2.2 *Given a $c \times d$ integer matrix A, an integer column vector $\vec{b}$ with c components, and a $d \times d$ unimodular matrix R such that $E = RA^T$ is an integer matrix in echelon form, then all integer solutions of $A\vec{x} = \vec{b}$ are given by $\vec{x} = [(\lambda_1, \ldots, \lambda_d)R]^T$ for arbitrary $\lambda_i \in \mathcal{Z}$ satisfying $[(\lambda_1, \ldots, \lambda_d)E]^T = \vec{b}$.*

An echelon reduction algorithm that computes such a unimodular matrix R with integer arithmetic only is presented in [19, p32-39]. This algorithm also provides a convenient method to compute the greatest common divisor of a number of integers since for $A = (\alpha_1, \ldots, \alpha_d)$, a matrix $E = (e_1, 0, \ldots, 0)^T$ is obtained with $\gcd(\alpha_1, \ldots, \alpha_d) = |e_1|$. It also provides an alternative method to construct a unimodular matrix with a given row or column [19, p55-59].

2.2.4 Solving a System of Linear Inequalities

The Fourier-Motzkin elimination method [10, 19] [61, p84-85][62][229, ch4] can be used to test the consistency of a *reasonably small* system of linear inequalities $A\vec{x} \leq \vec{b}$, or to convert this system into a form in which the lower and upper bounds of each variable x_i are expressed in terms of the variables $x_1, \ldots, x_{i-1}$ only. In particular, we focus on an implementation for integer systems [33].

Intuition Behind the Elimination Method

Central to so-called Fourier-Motzkin elimination is the observation that variable x_k can be eliminated from a system $A\vec{x} \leq \vec{b}$ by replacing each pair-wise combination of two inequalities that define a lower and upper bound on x_k as follows, where we assume that $c_1 > 0$ and $c_2 > 0$,

$$\left\{ \begin{array}{lcl} L & \leq & c_1 \cdot x_k \\ c_2 \cdot x_k & \leq & U \end{array} \right. \quad \rightarrow \quad c_2 \cdot L \leq c_1 \cdot U \tag{2.9}$$

After this elimination, which can be done with only integer arithmetic if all coefficients are integers, another system of linear inequalities not involving x_k results. For real-valued variables, the original system is consistent if and only if the second system is consistent [62]. For integer-valued variables, however, the projection (2.9) may be inexact. For example, eliminating variable x_1 from $16 \leq 3 \cdot x_1$ and $2 \cdot x_1 \leq 11$ yields a consistent system $32 \leq 33$, whereas the original system has no solution for $x_1 \in \mathcal{Z}$ (viz. $\lceil \frac{16}{3} \rceil \leq x_1 \leq \lfloor \frac{11}{2} \rfloor$).

Fourier-Motzkin Elimination

Given a system of linear inequalities represented by a $c \times (d+1)$ column augmented integer matrix $(A \mid \vec{b})$, Fourier-Motzkin proceeds by successively eliminating the variables in reverse order. Starting with $A^{(d)} = A$ and $\vec{b}^{\,(d)} = \vec{b}$, the following sequence of column augmented integer matrices is generated:

$$(A^{(d)} \mid \vec{b}^{\,(d)}) \rightarrow (A^{(d-1)} \mid \vec{b}^{\,(d-1)}) \rightarrow \ldots \rightarrow (A^{(1)} \mid \vec{b}^{\,(1)}) \rightarrow \vec{b}^{\,(0)} \tag{2.10}$$

Each $m^{(k)} \times (k+1)$ column augmented integer matrix $(A^{(k)} \mid \vec{b}^{\,(k)})$ in this sequence represents inequalities in the first k variables as follows:

$$A^{(k)} \begin{pmatrix} x_1 \\ \vdots \\ x_k \end{pmatrix} \leq \vec{b}^{\,(k)}$$

At each step k, the rows in the column augmented matrix $(A^{(k)} \mid \vec{b}^{\,(k)})$ are reordered so that for particular $1 \leq p^{(k)} < q^{(k)} \leq m^{(k)}$, we have $a_{ik}^{(k)} > 0$ for $1 \leq i \leq p^{(k)}$, $a_{ik}^{(k)} < 0$ for $p^{(k)} < i \leq q^{(k)}$ and $a_{ik}^{(k)} = 0$ for $q^{(k)} < i \leq m^{(k)}$. This reordering gives rise to three sets of linear inequalities in which only positive coefficients occur for variable x_k:[2]

[2] For integer-valued variables, each inequality $a_{i1}^{(k)} \cdot x_1 + \ldots + a_{ik}^{(k)} \cdot x_k \leq b_i^{(k)}$ may be simplified into the inequality $a_{i1}^{(k)}/g \cdot x_1 + \ldots + a_{ik}^{(k)}/g \cdot x_k \leq \lfloor b_i^{(k)}/g \rfloor$, where $g = \gcd(a_{i1}^{(k)}, \ldots, a_{ik}^{(k)})$. This is done for the first $q^{(k)}$ rows in our implementation [33].

$$\begin{cases} a_{ik}^{(k)} \cdot x_k & \leq & b_i^{(k)} - \sum_{j=1}^{k-1} a_{ij}^{(k)} \cdot x_j & \text{for } 1 \leq i \leq p^{(k)} \\ -b_i^{(k)} + \sum_{j=1}^{k-1} a_{ij}^{(k)} \cdot x_j & \leq & (-a_{ik}^{(k)}) \cdot x_k & \text{for } p^{(k)} < i \leq q^{(k)} \\ \sum_{j=1}^{k-1} a_{ij}^{(k)} \cdot x_j & \leq & b_i^{(k)} & \text{for } q^{(k)} < i \leq m^{(k)} \end{cases}$$

After the reordering, the first $p^{(k)}$ rows in $(A^{(k)} \mid \vec{b}^{\,(k)})$ define the upper bounds of variable x_k in terms of only the variables $x_1, \ldots, x_{k-1}$. Moreover, the next $q^{(k)} - p^{(k)}$ rows define the lower bounds of x_k in terms of only the variables $x_1, \ldots, x_{k-1}$. For integer-valued variables, the lower and upper bounds can be expressed as follows:

$$\max_{p^{(k)} < i \leq q^{(k)}} \left\lceil \frac{b_i^{(k)} - \sum_{j=1}^{k-1} a_{ij}^{(k)} \cdot x_j}{a_{ik}^{(k)}} \right\rceil \leq x_k \leq \min_{1 \leq i \leq p^{(k)}} \left\lfloor \frac{b_i^{(k)} - \sum_{j=1}^{k-1} a_{ij}^{(k)} \cdot x_j}{a_{ik}^{(k)}} \right\rfloor \tag{2.11}$$

The other rows represent inequalities in which x_k is not involved.

Subsequently, the next column augmented matrix in the sequence (2.10) is obtained by eliminating variable x_k from the system according to (2.9). This implies that the first $q^{(k)}$ inequalities are replaced by $p^{(k)} \cdot (q^{(k)} - p^{(k)})$ new inequalities, which gives rise to the following system:

$$\begin{cases} \sum_{j=1}^{k-1} (a_{ik}^{(k)} \cdot a_{i'j}^{(k)} - a_{i'k}^{(k)} \cdot a_{ij}^{(k)}) \cdot x_j & \leq a_{ik}^{(k)} \cdot b_{i'}^{(k)} - a_{i'k}^{(k)} \cdot b_i^{(k)} & 1 \leq i \leq p^{(k)} < i' \leq q^{(k)} \\ \sum_{j=1}^{k-1} a_{ij}^{(k)} \cdot x_j & \leq b_i^{(k)} & q^{(k)} < i \leq m^{(k)} \end{cases}$$

For $m^{(k-1)} = p^{(k)} \cdot (q^{(k)} - p^{(k)}) + m^{(k)} - q^{(k)}$, this system can be represented by an $m^{(k-1)} \times k$ column augmented integer matrix $(A^{(k-1)} \mid \vec{b}^{\,(k-1)})$, which is the next matrix in the sequence. This process is repeated until all variables have been eliminated.

Eventually, a column integer vector $\vec{b}^{(0)}$ results. If any of the components of this vector is negative, then the original system of inequalities is inconsistent (viz. an inequality $0 \leq b_i^{(0)}$ results where $b_i^{(0)} < 0$). Otherwise, the system is consistent in the sense that at least one real solution exists. Since projection (2.9) is inexact for integer-valued variables, however, this does not necessarily imply that there is also an integer solution. Still, this test provides a necessary (*but not sufficient*) condition for the existence of an integer solution.

Example: Consider the following system of linear inequalities:

$$\begin{pmatrix} 0 & -1 & -3 \\ 0 & 0 & -1 \\ -1 & 0 & 6 \\ 0 & 1 & 3 \\ 0 & 0 & 1 \\ 1 & 0 & -6 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \leq \begin{pmatrix} -10 \\ -1 \\ -1 \\ 15 \\ 3 \\ 50 \end{pmatrix}$$

Applying Fourier-Motzkin to this system yields the following sequence of column augmented integer matrices (see also the footnote on the previous page):

$$\left(\begin{array}{rrr|r} -1 & 0 & 6 & -1 \\ 0 & 1 & 3 & 15 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & -1 & -1 \\ 0 & -1 & -3 & -10 \\ 1 & 0 & -6 & 50 \end{array}\right) \rightarrow \left(\begin{array}{rr|r} 0 & 1 & 12 \\ 1 & 2 & 80 \\ -1 & -2 & -21 \\ 0 & -1 & -1 \\ 0 & 0 & 15 \\ 0 & 0 & 294 \\ 0 & 0 & 2 \\ -1 & 0 & -7 \\ 1 & 0 & 68 \end{array}\right) \rightarrow \left(\begin{array}{r|r} 1 & 78 \\ 1 & 68 \\ -1 & 3 \\ -1 & -7 \\ 0 & 15 \\ 0 & 294 \\ 0 & 2 \\ 0 & 118 \\ 0 & 11 \end{array}\right) \rightarrow \left(\begin{array}{r} 81 \\ 71 \\ 71 \\ 61 \\ 15 \\ 294 \\ 2 \\ 118 \\ 11 \end{array}\right)$$

Because all components of the terminating vector are positive, the system is consistent. The bounds of, for instance, x_1 are defined by $\max(-3, 7) \leq x_1 \leq \min(78, 68)$, which can be simplified into $7 \leq x_1 \leq 68$.

Example: The simplification given in the footnote at page 21 may avoid some inexact projections for integer-valued variables, but not all. For example, because $16 \leq 3 \cdot x_1$ and $2 \cdot x_1 \leq 11$ are first simplified into the inequalities $6 \leq x_1$ and $x_1 \leq 5$, projection yields $6 \leq 5$, indicating inconsistency for *integer-valued* variables:

$$\left(\begin{array}{r|r} 1 & 5 \\ -1 & -6 \end{array}\right) \rightarrow \left(\begin{array}{r} -1 \end{array}\right)$$

The projection remains inexact, however, for a similar system of linear inequalities consisting of $0 \leq x_1 \leq 0$, $16 \leq x_1 + 3 \cdot x_2$ and $x_1 + 2 \cdot x_2 \leq 11$:

$$\left(\begin{array}{rr|r} 1 & 2 & 11 \\ 1 & -3 & -16 \\ 1 & 0 & 0 \\ -1 & 0 & 0 \end{array}\right) \rightarrow \left(\begin{array}{r|r} 1 & 0 \\ 1 & 0 \\ -1 & 0 \end{array}\right) \rightarrow \left(\begin{array}{r} 0 \\ 0 \end{array}\right)$$

Because none of the inequalities in the original system can be simplified, Fourier-Motzkin elimination reveals consistency of the system (with real solutions $x_1 = 0$, $16/3 \leq x_2 \leq 11/2$).

Redundant Inequalities

If a closed half-space defined by a linear inequality is redundant with respect to the polyhedral set defined by a system of linear inequalities (see section 2.1.4), then we also say that this inequality is redundant with respect to the system. A redundant inequality can be eliminated to simplify the system. In the following sections we present two simplification methods that are especially useful *if the sequence is used to enumerate discrete points within the polyhedral set defined by the original system*. First, rows with $a_{ik}^{(k)} = 0$ are eliminated from each column augmented matrix $(A^{(k)} \mid \vec{b}^{(k)})$, because the corresponding inequalities together with inequalities arising from projection are also present in column augmented matrices that appear later in the sequence. Thereafter, redundant inequalities involving x_k are eliminated from $(A^{(k)} \mid \vec{b}^{(k)})$ during a backward scan over the sequence, where inequalities represented by previously considered column augmented matrices are preserved, so that inequalities arising from projection may contribute to the simplification. We first present a computationally inexpensive simplification method which assumes that all variables are bounded. Since some redundant inequalities remain undetected by this ad-hoc method, we also present an exact simplification method. Note that simplifications could already be performed *during* elimination to improve the efficiency of the solver itself, as suggested in [229, ch4].

Ad-Hoc Simplification

For each variable x_k, intervals $[l_k^{\min}, l_k^{\max}]$ and $[u_k^{\min}, u_k^{\max}]$ are recorded, indicating the possible values of lower and upper bounds of this variable respectively. The bounds in these intervals may have values in $\mathcal{Z} \cup \{-\infty, +\infty\}$, and for all $1 \leq k \leq d$, we initialize these values as follows:

$$l_k^{\min} = l_k^{\max} = -\infty \quad \text{and} \quad u_k^{\min} = u_k^{\max} = +\infty$$

Subsequently, more accurate values are determined during a backward scan over the sequence of column augmented matrices arising from Fourier-Motzkin elimination.

For each $m^{(k)} \times (k+1)$ column augmented matrix $(A^{(k)} \mid \vec{b}^{\,(k)})$ in this sequence, each remaining row with $a_{ik}^{(k)} \neq 0$ is considered:

$$\left(\begin{array}{cccc|c} a_{i1}^{(k)} & \ldots & a_{i,k-1}^{(k)} & a_{ik}^{(k)} & b_i^{(k)} \end{array} \right) \tag{2.12}$$

Because $x_j \in [l_j^{\min}, u_j^{\max}]$ holds for $j < k$, the extremal values of the corresponding expression not involving x_k are given below, where $a^+ = \max(a, 0)$ and $a^- = \max(-a, 0)$ according to the definitions given in [17][19, p52-54]:

$$\left\{ \begin{array}{rcl} l & = & b_i^{(k)} + \sum_{j=1}^{k-1} (-a_{ij}^{(k)})^+ \cdot l_j^{\min} - (-a_{ij}^{(k)})^- \cdot u_j^{\max} \\ u & = & b_i^{(k)} + \sum_{j=1}^{k-1} (-a_{ij}^{(k)})^+ \cdot u_j^{\max} - (-a_{ij}^{(k)})^- \cdot l_j^{\min} \end{array} \right.$$

If $a_{ik}^{(k)} > 0$, then (2.12) defines an upper bound of x_k that can only have values in the interval $[l', u']$, where $l' = \lfloor l / a_{ik}^{(k)} \rfloor$ and $u' = \lfloor u / a_{ik}^{(k)} \rfloor$. Therefore, if $u_k^{\max} \leq l'$ holds, then this inequality is redundant with respect to previously considered upper bounds of x_k and is eliminated. Similarly, if $u' \leq u_k^{\min}$, this inequality replaces all previously considered inequalities that define upper bounds of x_k. Thereafter, the following assignments are executed:

$$\begin{array}{rcl} u_k^{\min} & := & \min(\ u_k^{\min},\ l'\ ) \\ u_k^{\max} & := & \min(\ u_k^{\max},\ u'\ ) \end{array}$$

If $a_{ik}^{(k)} < 0$, then (2.12) defines a lower bound of x_k that can only have values in the interval $[l', u']$, where $l' = \lceil u / a_{ik}^{(k)} \rceil$ and $u' = \lceil l / a_{ik}^{(k)} \rceil$. Therefore, if $l_k^{\max} \leq l'$, this inequality replaces all previously considered inequalities that define lower bounds of x_k. The inequality is eliminated if $u' \leq l_k^{\min}$. Thereafter, the following assignments are executed:

$$\begin{array}{rcl} l_k^{\min} & := & \max(\ l_k^{\min},\ l'\ ) \\ l_k^{\max} & := & \max(\ l_k^{\max},\ u'\ ) \end{array}$$

After these actions have been performed for all column augmented matrices in the sequence, we obtain a new sequence of matrices in which some redundant inequalities (viz. rows) are eliminated.

Example: Consider the following system of linear inequalities:

$$\begin{pmatrix} -1 & -2 \\ 1 & -1 \\ 1 & 2 \\ -1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \leq \begin{pmatrix} -3 \\ 0 \\ 300 \\ -1 \\ 100 \end{pmatrix}$$

Applying Fourier-Motzkin elimination yields the following sequence, where the terminating column vector indicates the consistency of the system:

$$\left(\begin{array}{rr|r} 1 & 2 & 300 \\ 0 & 1 & 100 \\ -1 & -2 & -3 \\ 1 & -1 & 0 \\ -1 & 0 & -1 \end{array}\right) \rightarrow \left(\begin{array}{r|r} 1 & 100 \\ 1 & 100 \\ -1 & 197 \\ -1 & -1 \\ 0 & 594 \end{array}\right) \rightarrow \left(\begin{array}{r} 297 \\ 99 \\ 297 \\ 99 \\ 594 \end{array}\right)$$

During examination of the last column augmented matrix, the ad-hoc method simplifies the four inequalities that define bounds of x_1 into $1 \leq x_1$ and $x_1 \leq 100$. After the first inequality in the column augmented matrix representation of the bounds of x_2 has been considered, we obtain $u_2^{\min} = \lfloor (300 - 100)/2 \rfloor = 100$ and $u_2^{\max} = \lfloor (300 - 1)/2 \rfloor = 149$.

Since $u' = 100$ holds for the second inequality, we have $u' \leq u_2^{\min}$ and this inequality may replace the first inequality. Similar actions are taken for the inequalities that define lower bounds of x_2, and eventually the following simplified sequence results, from which the terminating column vector also has been eliminated:

$$\left(\begin{array}{rr|r} 0 & 1 & 100 \\ 1 & -1 & 0 \end{array}\right) \rightarrow \left(\begin{array}{r|r} 1 & 100 \\ -1 & -1 \end{array}\right)$$

Exact Simplification

As advocated in [10], proposition 2.1 provides a convenient method to detect redundant inequalities. A linear inequality is redundant with respect to a system of linear inequalities if the system obtained by negating this inequality is inconsistent. For instance, negation of inequality $x_1 \leq 11$ in the following system of inequalities yields $x_1 > 11$, which can be rewritten into $12 \leq x_1$ for integer-valued variables:

$$\begin{array}{rcl} 1 \leq & x_1 & \leq 10 \\ & x_1 & \leq 11 \end{array} \quad \text{Negation} \quad \rightarrow \quad \begin{array}{rcl} 1 \leq & x_1 & \leq 10 \\ 12 \leq & x_1 & \end{array} \quad \text{Elimination} \quad \rightarrow \quad \begin{array}{r} 1 \leq 10 \\ 12 \leq 10 \end{array}$$

The resulting system is inconsistent, indicating the redundancy of the third inequality. Because negating one of the other inequalities in the original system does not introduce an inconsistency, these inequalities are not redundant.

The following rewriting steps are used to negate a lower or upper bound of an integer-valued variable x_k, where $a > 0$:

$$\begin{array}{rcl} L & \leq & a \cdot x_k \\ a \cdot x_k & \leq & U \end{array} \quad \text{Negate} \quad \rightarrow \quad \begin{array}{rcl} a \cdot x_k & < & L \\ U & < & a \cdot x_k \end{array} \quad \text{Integers} \quad \rightarrow \quad \begin{array}{rcl} a \cdot x_k & \leq & L - 1 \\ U + 1 & \leq & a \cdot x_k \end{array}$$

Hence, in general, an inequality represented by the ith row of a column augmented integer matrix $(A \mid \vec{b})$ is negated as follows:

$$\left(\begin{array}{c|c} \vdots & \vdots \\ a_{i1} \ldots a_{ik} & b_i \\ \vdots & \vdots \end{array}\right) \rightarrow \left(\begin{array}{c|c} \vdots & \vdots \\ -a_{i1} \ldots -a_{ik} & -b_i - 1 \\ \vdots & \vdots \end{array}\right) \tag{2.13}$$

These observations give rise to the following exact simplification method during a backward scan over the sequence, possibly after ad-hoc simplifications have been applied to reduce the total number of inequalities that must be examined. At each step, we consider the following column augmented matrix that represents *all* inequalities involving the variables $x_1, \ldots, x_k$:

$$\left(\begin{array}{cccc|c} A^{(1)} & \vec{0} & \vec{0}\ldots\vec{0} & & \vec{b}^{\,(1)} \\ & A^{(2)} & \vec{0}\ldots\vec{0} & & \vec{b}^{\,(2)} \\ & & \ddots & & \vdots \\ & & & A^{(k)} & \vec{b}^{\,(k)} \end{array}\right) \tag{2.14}$$

If *several* inequalities define upper bounds for x_k, then one of these inequalities is negated and Fourier-Motzkin elimination is applied to test the consistency of the resulting system. The inequality is eliminated from both $(A^{(k)} \mid \vec{b}^{\,(k)})$ and matrix (2.14) if this system is inconsistent, or recovered into the original inequality otherwise. This process is repeated until all upper bounds have been considered or only one upper bound remains. Similar steps are performed as long as several not examined lower bounds remain.

Example: The following system of linear inequalities in two variables describes the convex polygon shown in figure 2.4:

$$\begin{pmatrix} -1 & 0 \\ 1 & -1 \\ 2 & -5 \\ 0 & -1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \leq \begin{pmatrix} 0 \\ 4 \\ 2 \\ -2 \\ 8 \end{pmatrix}$$

If the sequence obtained after Fourier-Motzkin elimination is simplified by the ad-hoc method, one redundant bound remains undetected:

$$\left(\begin{array}{cc|c} 0 & 1 & 8 \\ 1 & -1 & 4 \\ 2 & -5 & 2 \\ 0 & -1 & -2 \end{array}\right) \rightarrow \left(\begin{array}{c|c} 1 & 12 \\ -1 & 0 \end{array}\right)$$

Since only one upper and lower bound is defined for x_1, no simplifications are applied to the last column augmented matrix in the sequence. However, three lower bounds are defined for x_2. Hence inequality $x_1 - x_2 \leq 4$ is negated in system (2.14) for $k = 2$. Thereafter, Fourier-Motzkin is used to test consistency of the resulting system:

$$\left(\begin{array}{cc|c} 0 & 1 & 8 \\ \hline -1 & 1 & -5 \\ \hline 2 & -5 & 2 \\ 0 & -1 & -2 \\ 1 & 0 & 12 \\ -1 & 0 & 0 \end{array}\right) \rightarrow \left(\begin{array}{c|c} 1 & 21 \\ 1 & 12 \\ -1 & -8 \\ -1 & -7 \\ -1 & 0 \\ 0 & 6 \end{array}\right) \rightarrow \begin{pmatrix} 13 \\ 14 \\ 21 \\ 4 \\ 5 \\ 12 \\ 6 \end{pmatrix}$$

Because the terminating column vector indicates that the resulting system is consistent, the negated inequality (shown in between lines) is restored into the original inequality. Thereafter, inequality $2 \cdot x_1 - 5 \cdot x_2 \leq 2$ is negated, followed by Fourier-Motzkin elimination to test consistency:

$$\left(\begin{array}{cc|c} 0 & 1 & 8 \\ \hline -2 & 5 & -3 \\ \hline 1 & -1 & 4 \\ 0 & -1 & -2 \\ 1 & 0 & 12 \\ -1 & 0 & 0 \end{array}\right) \rightarrow \left(\begin{array}{c|c} 1 & 12 \\ 1 & 5 \\ 1 & 12 \\ -1 & -7 \\ -1 & 0 \\ 0 & 6 \end{array}\right) \rightarrow \begin{pmatrix} 5 \\ 12 \\ -2 \\ 5 \\ 5 \\ 12 \\ 6 \end{pmatrix}$$

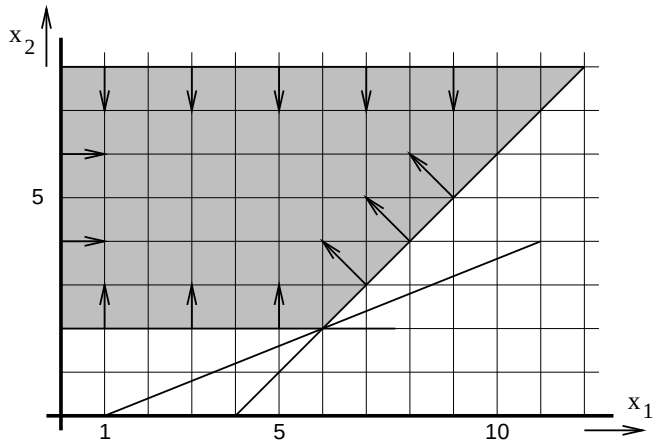


Figure 2.4: Simplification of Lower Bound

Because this system is inconsistent, inequality $2 \cdot x_1 - 5 \cdot x_2 \leq 2$ is redundant with respect to the original system and is eliminated. Since still two upper bounds remain, inequality $x_2 \geq 2$ is also negated, followed by Fourier-Motzkin elimination to test consistency:

$$\left(\begin{array}{rr|r} 0 & 1 & 8 \\ \hline 0 & 1 & 1 \\ \hline 1 & -1 & 4 \\ 1 & 0 & 12 \\ -1 & 0 & 0 \end{array}\right) \rightarrow \left(\begin{array}{r|r} 1 & 12 \\ 1 & 5 \\ 1 & 12 \\ -1 & 0 \end{array}\right) \rightarrow \left(\begin{array}{r} 12 \\ 5 \\ 12 \end{array}\right)$$

The resulting system of inequalities is consistent, which indicates that this inequality must be restored. Eventually, exact simplification yields the following sequence, where rows with $a_{ij}^{(k)} = 0$ in each $(A^{(k)} \mid \vec{b}^{\,(k)})$ are no longer required:

$$\left(\begin{array}{rr|r} 0 & 1 & 8 \\ 1 & -1 & 4 \\ 0 & -1 & -2 \end{array}\right) \rightarrow \left(\begin{array}{r|r} 1 & 12 \\ -1 & 0 \end{array}\right)$$

Note that actually inequality $x_1 \leq 12$ is redundant with respect to the *whole* system of inequalities since it arises naturally from the inequalities $x_2 \leq 8$ and $x_1 - x_2 \leq 4$. However, our simplification methods keep variable x_k bounded in the system represented by $(A^{(k)} \mid \vec{b}^{\,(k)})$.

Comparison of Different Simplifying Methods

In this section, we present the performance of Fourier-Motzkin elimination and the two different simplification methods applied to some $2 \cdot d \times (d+1)$ matrices of the following form:

$$\left(\begin{array}{rccr|r} 1 & & & -1 & 0 \\ & \ddots & & \vdots & \vdots \\ & & 1 & -1 & 0 \\ & & & -1 & 0 \\ -1 & & & 1 & 0 \\ & \ddots & & \vdots & \vdots \\ & & -1 & 1 & 0 \\ & & & 1 & 999 \end{array}\right) \tag{2.15}$$

	Remaining Bounds			Execution Time				
	F.M.	Ad-Hoc	Exact	F.M.	Ad-Hoc	Exact	Total	Exact Only
2	6	4	4	0.2	0.1	0.0	0.1	1.2
3	14	8	6	0.5	0.2	0.9	1.1	4.9
4	34	16	8	1.5	0.4	4.8	5.2	20.6
5	138	36	10	20.7	1.3	23.8	25.1	691.5

Table 2.1: Number of Remaining Bounds and Execution Time in milli-seconds

These systems have the important property that some (but not all) redundant inequalities are eliminated by the ad-hoc method. Applying Fourier-Motzkin elimination to the matrix defined for $d = 3$, for instance, yields the following sequence:

$$\left(\begin{array}{ccc|c} -1 & 0 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 0 & 1 & 999 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & -1 & 0 \\ 0 & 0 & -1 & 0 \end{array}\right) \rightarrow \left(\begin{array}{cc|c} -1 & 1 & 0 \\ 0 & 1 & 999 \\ 1 & -1 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 0 \\ -1 & 0 & 0 \\ 1 & 0 & 999 \\ 0 & 0 & 0 \\ 0 & 0 & 999 \end{array}\right) \rightarrow \left(\begin{array}{c|c} 1 & 999 \\ 1 & 999 \\ -1 & 0 \\ -1 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 999 \\ 0 & 0 \\ 0 & 999 \end{array}\right) \rightarrow \left(\begin{array}{c|c} 999 \\ 999 \\ 999 \\ 999 \\ 0 \\ 0 \\ 999 \\ 0 \\ 999 \end{array}\right)$$

Ad-hoc simplification results in the elimination of some redundant inequalities:

$$\left(\begin{array}{ccc|c} -1 & 0 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & -1 & 0 \end{array}\right) \rightarrow \left(\begin{array}{cc|c} -1 & 1 & 0 \\ 1 & -1 & 0 \end{array}\right) \rightarrow \left(\begin{array}{c|c} 1 & 999 \\ -1 & 0 \end{array}\right)$$

Finally, the exact method performs the following simplification, which is only possible using the inequalities $x_1 \leq x_2 \leq x_1$ arising from projection:

$$\left(\begin{array}{ccc|c} 0 & -1 & 1 & 0 \\ 0 & 1 & -1 & 0 \end{array}\right) \rightarrow \left(\begin{array}{cc|c} -1 & 1 & 0 \\ 1 & -1 & 0 \end{array}\right) \rightarrow \left(\begin{array}{c|c} 1 & 999 \\ -1 & 0 \end{array}\right)$$

In table 2.1, we show the number of inequalities that remain after Fourier-Motzkin elimination, the ad-hoc simplification method and the exact simplification has been applied to matrix (2.15) for $d = 2, 3, 4, 5$. In the same table, we also present the execution time in milli-seconds on an HP 9000/720 for Fourier-Motzkin elimination, the ad-hoc method followed by the exact method, and the exact method without preceding application of the ad-hoc method. All versions are compiled with default optimizations enabled (but have not been fully hand-optimized with respect to e.g. memory allocation, which could yield a substantial reduction in execution time).

This simple example illustrates that applying the exact method can be expensive in comparison with actually performing Fourier-Motzkin elimination. Therefore, it must be possible to disable this exact simplification. Furthermore, the example also illustrates that if exact simplification is desired, then the total simplification time can be reduced substantially by using the ad-hoc method as a filter for the exact method.

Chapter 3

Loop Transformations

Many issues related to serial loops can be formalized using the concepts introduced in the previous chapter. The iteration space of particular loops, for instance, can be represented by a system of linear inequalities in the loop indices. Likewise, certain subscript functions may be expressed as affine transformations from an iteration space to the index space of an array. In this chapter, we first discuss these representations. In addition, we define two relations on statement instances. The execution order, arising from the sequential semantics of loops, induces a *total order* on statement instances. Data dependences, on the other hand, arise from the way in which data is used by these statement instances. In general, data dependences (and control dependences) induce a *partial order* on statement instances.

An important observation for program restructuring is that changing the execution order on statement instances does not affect the results of a program, if none of the dependences is violated. Any program transformation that preserves all dependences, also preserves the semantics of the program. In this chapter, we discuss how this observation can be used in the context of loop transformations. We briefly discuss the exploitation of implicit parallelism by relaxing the execution order induced by individual DO-loops as far as dependences allow. Moreover, we discuss the framework of unimodular transformations that provides a mathematical foundation for some conventional loop transformations. Finally, we present a method that isolates the loop-body of a nested loop for all iterations that satisfy a number of linear inequalities simultaneously.

3.1 Sequential Loops

In FORTRAN, the DO-loop is an important construct to define iteration. If individual DO-loops are used within other DO-loops, a so-called nested loop results:

```
DO I1 = L1, U1
  ...
  DO I2 = L2, U2
    ...
      DO Id = Ld, Ud
        B(I1,...,Id)
      ENDDO
    ...
  ENDDO
  ...
ENDDO
```

3.1.1 Loop Terminology

If no other statements appear in between the individual DO-loops, then the whole loop is called a **perfectly nested loop**. For $d = 2$ and $d = 3$, we speak of double and triple loops respectively.

A loop in which arbitrary statements, or even complete other DO-loops appear in between the individual DO-loops is referred to as a **non-perfectly nested loop**. We assume that each $\mathtt{I}_i$ is an integer-valued variable, called a **loop index**. We will refer to the DO-loop having $\mathtt{I}_i$ as loop index as the $\mathtt{I}_i$-loop. We call $\vec{\mathtt{I}} = (\mathtt{I}_1, \ldots, \mathtt{I}_d)^T$ the **index vector** of the nested loop. The **loop-body** B of this loop consists of a sequence of **indexed statements** at **nesting depth** d. Each individual indexed statement in this loop-body is denoted by $S_i(\vec{\mathtt{I}})$ for some unique label S_i. These labels reflect the relative position of a statement in a program in the sense that if S_i textually appears before S_j, then $i < j$ holds.

If the loop-body is executed for the value $\vec{\mathtt{I}} = \vec{\imath}$, where $\vec{\imath} \in \mathcal{Z}^d$, then we call this vector an **iteration (vector)** of this loop. Substituting the value $\vec{\imath} \in \mathcal{Z}^d$ for $\vec{\mathtt{I}}$ in an indexed statement $S_i(\vec{\mathtt{I}})$ in this loop-body yields the **statement instance** $S_i(\vec{\imath})$ executed during this iteration. Statements at nesting depth zero only have one instance and are usually referred to as scalar statements. The set IS of all iterations for which the loop-body of a nested loop is executed is called the **iteration space** of the loop. Under the assumption that only integer-valued variables are used as loop indices, we have $IS \subseteq \mathcal{Z}^d$.

Example: Consider the following double loop:

```
      DO I1 = 1, 2
        DO I2 = 1, 3
S1:       A(I1,I2) = 10.0
S2:       B(I1,I2) = B(I1,I2) - 1
        ENDDO
      ENDDO
```

The index vector of this loop is $\vec{\mathtt{I}} = (\mathtt{I}_1, \mathtt{I}_2)^T$. The loop-body consists of the two assignment statements S_1 and S_2, appearing at nesting depth two. For this loop, $S_1(\vec{\mathtt{I}})$ denotes the indexed statement '`A(I1,I2)=10.0`' and $S_1(1,2)$ denotes the instance '`A(1,2)=10.0`' of this statement executed in iteration $\vec{\mathtt{I}} = (1,2)^T$. The iteration space $IS \subseteq \mathcal{Z}^2$ of this loop is shown below:

$$IS = \{(1,1)^T, (1,2)^T, (1,3)^T, (2,1)^T, (2,2)^T, (2,3)^T\}$$

3.1.2 Loop Bounds

Because **DO-loop normalization** [234, p174–177] can be used to enforce unit strides, usually we assume that each DO-loop is stride one. For such a DO-loop, the loop index $\mathtt{I}_i$ iterates over all integers in the closed interval $[\mathtt{L}_i, \mathtt{U}_i]$, called the **execution set** of the $\mathtt{I}_i$-loop. This execution set is defined by the **loop bounds** $\mathtt{L}_i$ and $\mathtt{U}_i$, which may depend on the indices of outer DO-loops.

Admissible Loop Bounds

A single lower bound $\mathtt{L}_i$ or upper bound $\mathtt{U}_i$ is called an **admissible loop bound** if it can be expressed as follows, where all $l_{ij}, u_{ij} \in \mathcal{Z}$ and $l_{ii} > 0$ and $u_{ii} > 0$:

$$\mathtt{L}_i = \left\lceil \frac{l_{i0} + \sum_{j=1}^{i-1} l_{ij} \cdot \mathtt{I}_j}{l_{ii}} \right\rceil \quad \text{and} \quad \mathtt{U}_i = \left\lfloor \frac{u_{i0} + \sum_{j=1}^{i-1} u_{ij} \cdot \mathtt{I}_j}{u_{ii}} \right\rfloor$$

Because $\mathtt{I}_i$ is an integer-valued variable and $l_{ii} > 0$, the inequality $\mathtt{L}_i \leq \mathtt{I}_i$ can be expressed in terms of the index vector $\vec{\mathtt{I}}$ as follows:

$$(l_{i1} \ldots l_{i,i-1} \quad -l_{ii} \quad \underbrace{0 \ldots 0}_{d-i}) \cdot \vec{\mathtt{I}} \quad \leq \quad -l_{i0} \tag{3.1}$$

Likewise, the inequality $\mathtt{I}_i \leq \mathtt{U}_i$ can be expressed as shown below:

$$(-u_{i1} \ldots - u_{i,i-1} \quad u_{ii} \quad \underbrace{0 \ldots 0}_{d-i}) \cdot \vec{\mathtt{I}} \quad \leq \quad u_{i0} \tag{3.2}$$

Furthermore, a lower and upper bound is also admissible if it consists of the maximum or minimum of a number of admissible bounds respectively, as illustrated below:

$$\mathtt{L}_i = \mathtt{MAX}(\mathtt{L}_i^1, \mathtt{L}_i^2, \ldots) \quad \text{and} \quad \mathtt{U}_i = \mathtt{MIN}(\mathtt{U}_i^1, \mathtt{U}_i^2, \ldots)$$

Because all inequalities in such compound bounds must be satisfied simultaneously, these bounds give rise to several inequalities of the form (3.1) and (3.2) respectively. Consequently, the iteration space $IS \subseteq \mathcal{Z}^d$ of a loop in which all loop bounds are admissible can be represented by a system of linear inequalities $A\vec{\mathtt{I}} \leq \vec{b}$, where A is an integer matrix and $\vec{b}$ an integer vector:

$$IS = \{\vec{\mathtt{I}} \in \mathcal{Z}^d \mid A\vec{\mathtt{I}} \leq \vec{b}\}$$

Because in FORTRAN only finite execution sets may be used, for integer-valued loop indices the iteration space $IS \subseteq \mathcal{Z}^d$ consists of all discrete points in a bounded polyhedral set $PS \subseteq \mathcal{R}^d$. Usually, programmers use a *single* lower and upper bound where $l_{ii} = 1$ or $u_{ii} = 1$ in (3.1) and (3.2), which we will refer to as **simple loop bounds**. However, the more general loop bounds discussed in this section may arise after loop transformations.

Example: In the following triple loop, a compound upper bound is used in the $\mathtt{I}_2$-loop:

```
DO I1 = 0, 7
  DO I2 = I1, MIN(I1+3, 8)
    DO I3 = 0, 7-I1
      B(I1,I2,I3)
    ENDDO
  ENDDO
ENDDO
```

The following system of inequalities represents the iteration space of this loop:

$$\begin{array}{rcl} 0 & \leq & \mathtt{I}_1 \\ \mathtt{I}_1 & \leq & 7 \\ \mathtt{I}_1 & \leq & \mathtt{I}_2 \\ \mathtt{I}_2 & \leq & \mathtt{I}_1 + 3 \\ \mathtt{I}_2 & \leq & 8 \\ 0 & \leq & \mathtt{I}_3 \\ \mathtt{I}_3 & \leq & 7 - \mathtt{I}_1 \end{array} \qquad \begin{pmatrix} -1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & -1 & 0 \\ -1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \\ 1 & 0 & 1 \end{pmatrix} \vec{\mathtt{I}} \leq \begin{pmatrix} 0 \\ 7 \\ 0 \\ 3 \\ 8 \\ 0 \\ 7 \end{pmatrix}$$

Each inequality defines a half-space in $\mathcal{R}^3$. The last inequality, for instance, defines the half-space $H = \{\vec{\mathtt{I}} \in \mathcal{R}^3 \mid \mathtt{I}_1 + \mathtt{I}_3 \leq 7\}$. In figure 3.1, the convex polyhedron defined by the intersection of all these half-spaces is shown. Taking the intersection of this polyhedron and $\mathcal{Z}^3$ yields the iteration space of the loop. Note that half-space defined by $\mathtt{I}_1 \leq 7$ is redundant with respect to the polyhedron. Indeed, we could even 'simplify' the bounds of the outermost DO-loop into '`DO` $\mathtt{I}_1$`=1,`∞' without introducing additional iterations since for $\mathtt{I}_1 > 7$ only zero-trip loops are executed. For obvious reasons, however, we are only interested in simplifications that keep execution sets bounded (cf. section 2.2.4).

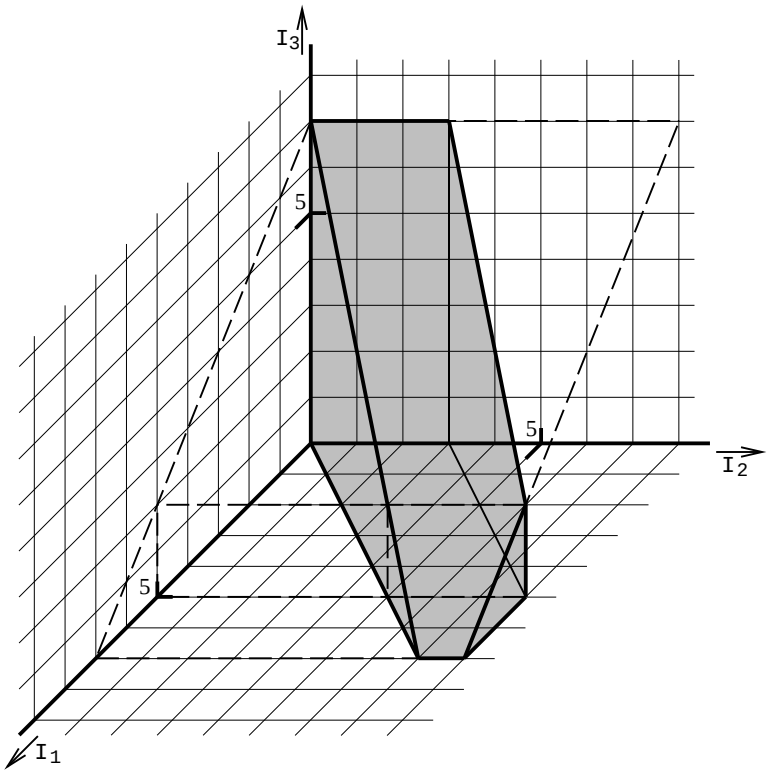


Figure 3.1: Iteration Space

Inadmissible Loop Bounds

All loop bounds that are not admissible are referred to as **inadmissible loop bounds**. Loop transformations that rely on the representation of loop bounds as a system of linear inequalities may become disabled in the presence of inadmissible bounds. However, transformations may still be applicable to a sub-loop consisting of a few DO-loops with admissible bounds. Moreover, a system of inequalities in which some loop indices are unbounded may be used as a conservative representation of the iteration space of a loop with inadmissible loop bounds. In this case, the approximated iteration space consists of all discrete points in an *unbounded* polyhedral set.

3.1.3 Subscript Functions

We can distinguish between individual elements of arrays by means of **subscript functions**, also called subscripts for short. A c-dimensional array A may be accessed by any c-tuple $(f_1, \ldots, f_c)$ of subscripts, where each subscript f_i must evaluate to an integer value satisfying the subscript bounds of the ith dimension.

Admissible Subscripts

A subscript f_i of an occurrence of a c-dimensional array A at nesting depth d is called an **admissible subscript** if it can be expressed as an affine transformation that is restricted to $f_i : \mathcal{Z}^d \to \mathcal{Z}$ because all $v_i, w_{ij} \in \mathcal{Z}$ and only integer-valued variables are used as loop indices:

$$f_i(\vec{\mathtt{I}}) = v_i + \sum_{j=1}^{d} w_{ij} \cdot \mathtt{I}_j \tag{3.3}$$

Hence, if all subscript functions are admissible, these subscripts can be represented by a single affine transformation $F : \mathcal{Z}^d \rightarrow \mathcal{Z}^c$ that can be expressed in matrix form $F(\vec{\mathtt{I}}) = \vec{v} + W\vec{\mathtt{I}}$ as follows, where W is a $c \times d$ integer matrix and $\vec{v}$ is an integer vector with c components:

$$F(\vec{\mathtt{I}}) = \begin{pmatrix} v_1 \\ \vdots \\ v_c \end{pmatrix} + \begin{pmatrix} w_{11} & \dots & w_{1d} \\ \vdots & \ddots & \\ w_{c1} & & w_{cd} \end{pmatrix} \vec{\mathtt{I}} \tag{3.4}$$

We assume that subscript bounds are not violated, i.e. if the bounds of the corresponding array are declared as '$\mathtt{A}(\mathtt{L}_1 : \mathtt{U}_1, \dots, \mathtt{L}_c : \mathtt{U}_c)$', then $F(\vec{\mathtt{I}}) \in [\mathtt{L}_1, \mathtt{U}_1] \times \dots \times [\mathtt{L}_c, \mathtt{U}_c]$ for all $\vec{\mathtt{I}} \in IS$, where $IS \subseteq \mathcal{Z}^2$ denotes the iteration space of the loop.

Example: The prototype restructuring compiler MT1 [37, 24, 45] uses constant folding and some simple algebraic equivalences to detect admissible subscripts and loop bounds. Each admissible subscript or loop bound is prompted to the programmer between angle brackets. The subscript functions belonging to the occurrence of array X in following loop, for instance, are prompted by MT1 to the programmer as '<2*I+6*J+17>,<K>,<1>':

```
      DO 10 I = 1, 100
        DO 5 J = 1, 100
          DO 1 K = 1, 100
            X(5*(I+J)-(I-5)*3+J+2,K,K+1-K) = 10.0
 1        CONTINUE
 5      CONTINUE
10    CONTINUE
```

These subscripts are represented by an affine transformation $F : \mathcal{Z}^2 \rightarrow \mathcal{Z}^3$ that can be expressed in matrix form as follows, where $\vec{\mathtt{I}} = (\mathtt{I}, \mathtt{J}, \mathtt{K})^T$:

$$F(\vec{\mathtt{I}}) = \begin{pmatrix} 17 \\ 0 \\ 1 \end{pmatrix} + \begin{pmatrix} 2 & 6 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix} \vec{\mathtt{I}}$$

Inadmissible Subscripts

A subscript that cannot be expressed as an affine transformation in surrounding loop indices is referred to as an **inadmissible subscript**. Usually, programs containing inadmissible subscripts cannot be analyzed very accurately. Conventional transformations, like constant propagation, scalar forward substitution and induction variable substitution [234, ch3][3, ch10] can be used to increase the number of admissible subscripts and loop bounds. Moreover, although we cannot always expect admissible subscripts in general programs [192], most subscript functions used in numerical applications are admissible.

3.1.4 Execution Order

The relation '$\prec_k$' on $\mathcal{Z}^d$ for $1 \leq k \leq d$ and the **lexicographical order** '$\prec$' on $\mathcal{Z}^d$ are defined as follows, where $\vec{\imath}, \vec{\jmath} \in \mathcal{Z}^d$:

$$\begin{cases} \vec{\imath} \prec_k \vec{\jmath} & \Leftrightarrow & i_1 = j_1, \dots, i_{k-1} = j_{k-1}, i_k < j_k \\ \vec{\imath} \prec \vec{\jmath} & \Leftrightarrow & \exists 1 \leq k \leq d : \vec{\imath} \prec_k \vec{\jmath} \end{cases}$$

We have $\vec{\imath} \preceq \vec{\jmath}$ if either $\vec{\imath} \prec \vec{\jmath}$ or $\vec{\imath} = \vec{\jmath}$. The relations '$\succ_k$' for $1 \leq k \leq d$, '$\succ$' and '$\succeq$' are defined similarly. Moreover, we say that a vector $\vec{\imath} \in \mathcal{Z}^d$ is **lexicographically positive** if $\vec{\imath} \succ \vec{0}$ holds, which means that the first nonzero component is positive.

The following notation is used to isolate some consecutive components of a vector $\vec{i} \in \mathcal{Z}^d$, where $1 \leq c_1 \leq c_2 \leq d$:

$$\vec{i}[c_1 : c_2] = (i_{c_1}, \ldots, i_{c_2})^T$$

Let $IS \subseteq \mathcal{Z}^{d_i}$ and $JS \subseteq \mathcal{Z}^{d_j}$ denote the iteration spaces of two loops in which respectively the statements S_i and S_j occur. We define the **common nesting depth** $d \leq \min(d_i, d_j)$ of S_i and S_j as the nesting depth of the *innermost* DO-loop that is still shared by both statements. The **execution order** '$<_o$' on instances of S_i and S_j, induced by the sequential semantics of FORTRAN DO-loops with *positive* strides, can be defined as follows, where $\vec{i} \in IS$ and $\vec{j} \in JS$:

$$S_i(\vec{i}) <_o S_j(\vec{j}) \quad \Leftrightarrow \quad \begin{cases} \vec{i}[1:d] \prec \vec{j}[1:d] \\ \text{or} \\ \vec{i}[1:d] = \vec{j}[1:d] \text{ and } i < j \end{cases}$$

If the stride of a particular DO-loop is negative, the value of the corresponding component of the iteration vector decreases in successive iterations of that DO-loop. Although it is straightforward to deal with this subtlety in the definition of '$<_o$' [228], usually we assume that all strides are positive. In fact, most DO-loops are stride-1 [123], whereas, as stated before, DO-loop normalization can be used to enforce unit strides.

3.1.5 Data Dependences

In this section, we define data dependences. This relation on statement instances arises from the flow of data in a program and, in contrast with the execution order, induces a partial order on statement instances.

Input and Output Sets

The **input set** and the **output set** of a statement S_i, denoted by $\text{IN}(S_i)$ and $\text{OUT}(S_i)$ respectively, consist of all variables that may be read or written to by this statement. Substituting an iteration $\vec{i} \in IS$ for the index vector of the loop with iteration space $IS \subseteq \mathcal{Z}^d$ in which S_i appears yields the sets $\text{IN}(S_i(\vec{i}))$ and $\text{OUT}(S_i(\vec{i}))$ consisting of the actual elements that are read or written to by the statement instance $S_i(\vec{i})$.
Example: Consider the following assignment statement:

```
S1:  X(K+1) = A(I,J) * S - 4.0
```

The input and output set of this statement are shown below:

$$\begin{cases} \text{IN}(S_1) &= \{ \quad \texttt{A(I,J)}, \texttt{I}, \texttt{J}, \texttt{K}, \texttt{S} \quad \} \\ \text{OUT}(S_1) &= \{ \quad \texttt{X(K+1)} \quad \} \end{cases}$$

If the variables `I`, `J` and `K` are used as loop indices, then the input and output set of the instance $S_1(1,2,3)$ have the following form (the loop indices vanish):

$$\begin{cases} \text{IN}(S_1(1,2,3)) &= \{ \quad \texttt{A(1,2)}, \texttt{S} \quad \} \\ \text{OUT}(S_1(1,2,3)) &= \{ \quad \texttt{X(4)} \quad \} \end{cases}$$

The input and output sets of statements enables us to define data dependences.

Flow, Anti, and Output Dependences

Consider two statements S_i and S_j that appear two possibly different loops with the iteration spaces $IS \subseteq \mathcal{Z}^{d_i}$ and $JS \subseteq \mathcal{Z}^{d_j}$ respectively. If $S_i(\vec{\imath}) <_o S_j(\vec{\jmath})$ for $\vec{\imath} \in IS$ and $\vec{\jmath} \in JS$, then the following memory-based[1] **data dependences** may arise between these two instances, respectively called a **flow**, **anti**, and **output-dependence**:

$$\begin{array}{lcllcllcl}
S_i(\vec{\imath}) & \delta & S_j(\vec{\jmath}) & \text{if} & \text{OUT}(& S_i(\vec{\imath}) &) & \cap & \text{IN}(\quad S_j(\vec{\jmath}) \quad) \neq \emptyset \\
S_i(\vec{\imath}) & \overline{\delta} & S_j(\vec{\jmath}) & \text{if} & \text{IN}(& S_i(\vec{\imath}) &) & \cap & \text{OUT}(\quad S_j(\vec{\jmath}) \quad) \neq \emptyset \\
S_i(\vec{\imath}) & \delta^o & S_j(\vec{\jmath}) & \text{if} & \text{OUT}(& S_i(\vec{\imath}) &) & \cap & \text{OUT}(\quad S_j(\vec{\jmath}) \quad) \neq \emptyset
\end{array}$$

The notation $S_i(\vec{\imath})\ \delta^*\ S_j(\vec{\jmath})$ is used to indicate that there is an arbitrary data dependence between two statement instances, and we say that the instance $S_j(\vec{\jmath})$ depends on the instance $S_i(\vec{\imath})$, or we say that there is a data dependence from instance $S_i(\vec{\imath})$ to the instance $S_j(\vec{\jmath})$.

If $d \leq \min(d_i, d_j)$ denotes the common nesting depth of S_i and S_j, then the **dependence distance vector** of such a data dependence is defined as the following vector:

$$\vec{d} = \vec{\jmath}[1:d] - \vec{\imath}[1:d]$$

It is not difficult to see that, because $S_i(\vec{\imath}) <_o S_j(\vec{\jmath})$, all dependence distance vectors are zero or lexicographically positive. The data dependence is called **loop-independent** if $\vec{d} = \vec{0}$, and **loop-carried** if $\vec{d} \succ \vec{0}$. In particular, in the latter case we say that the loop is carried by the $\texttt{I}_k$-loop if $\vec{d} \succ_k \vec{0}$. Furthermore, the data dependence is called **lexically forward** if $i > j$, **lexically backward** if $i < j$, or a **self-dependence** if $i = j$ holds.

Data dependences impose an ordering constraint on the execution of statement instances because, if there is a data dependence between two statement instances, then changing the execution order of these instances could change the semantics of the program. Anti and output dependences arise from the re-use of memory and, in principle, could be removed by the introduction of new variables [5]. Therefore, these data dependences are also referred to as false dependences, whereas flow dependences are sometimes called true dependences. If the intersection of the input sets of two statement instances is non-empty, then this gives rise to an **input dependence**. No ordering constraint are imposed by input dependences, which are not further considered in this dissertation.

If the execution of a statement instance depends on the outcome of a particular test, a so-called **control dependence** arises. Although control dependences also impose an ordering constraint on the execution of statement instances, we usually do not explicitly consider control dependences. In fact, there are methods to convert control dependences into data dependences [6][234, p238-249].

Static Data Dependences

In the presence of loops, it is usually infeasible to record all data dependences, because the compiler cannot represent each statement instance individually. Therefore, some abstraction is required [p139-140][229]. We say that there is a **static** flow, anti or output dependence between a **source statement** S_i and a **sink statement** S_j, denoted by $S_i \delta S_j$, $S_i \overline{\delta} S_j$ and $S_i \delta^o S_j$ respectively, if there is at least one such a data dependence between instances of S_i and S_j. Again, the notation $S_i \delta^* S_j$ is used to indicate an arbitrary static data dependence, and we say that S_j depends on S_i or that there is a dependence from S_i to S_j. A convenient graphical representation of static data dependences consists of a **data dependence graph**, in which the vertices and edges correspond to statements and the different static data dependences, respectively.

[1] In contrast, data dependences are called value-based if there are no intermediate writes to the data elements causing the data dependence [174][229, ch.5].

One way to represent the data dependence structure of a program is to annotate each *static* data dependence with dependence distance vectors of the underlying data dependences. However, even representing the data dependence structure in terms of a set of dependence distance vectors is not always feasible. If some of the loop bounds are very large or inadmissible, an infeasible or infinite number of distance vectors may arise. In such cases, a static data dependence can be annotated with a **dependence direction vector**, which simply records the sign of each component of the corresponding dependence distance vectors. Each component of a dependence direction vector is an element in $\{*, <, >, =\}$, corresponding to an unknown sign, positive or negative sign, or a zero component respectively. In this manner, a possibly infinite set of dependence distance vectors can be recorded. For example, the dependence distance vector $(<, =)^T$ represents the following infinite set of dependence distance vectors:

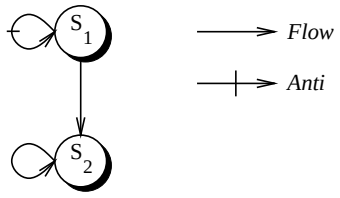
$$\{(1,0)^T, (2,0)^T, (3,0)^T, (4,0)^T, \ldots\}$$

An annotated static data dependence is denoted with the corresponding dependence distance or direction vector as subscript (e.g. $S_1\delta_{(+1,0,-4)}S_2$ or $S_1\delta_{(<,=,>)}S_2$).

If a perfectly nested loop with iteration space $IS \subseteq \mathcal{Z}^d$ is considered in isolation, we may represent the data dependence structure of this loop as a set $\mathcal{D} \subseteq \mathcal{Z}^d$ of dependence distance (or direction) vectors, where $\vec{d} \in \mathcal{D}$ simply implies that a statement instance executed in an iteration $\vec{j} \in IS$ depends on an instance executed in iteration $\vec{i} \in IS$, where $\vec{j} = \vec{i} + \vec{d}$. In this manner, we abstract from data dependences that involve statements outside the loop-body of this loop, and from components of dependence distance vectors caused by DO-loops that appear within the loop-body.

Example: Consider the following fragment:

```
   DO I1 = 1, N-1
S1:  X(I1) = X(I1+1) * 5.0
   ENDDO
   DO I1 = 2, N
     DO I2 = 1, N-1
S2:    A(I1,I2) = A(I1-1,I2+1) * X(I1)
     ENDDO
   ENDDO
```



As depicted in the data dependence graph, in this fragment the static anti dependence $S_1\overline{\delta}_{(1)}S_1$ and the static flow dependences $S_1\delta S_2$ and $S_2\delta_{(+1,-1)}S_2$ hold. The data dependence structure of the first loop in isolation may be represented by $\mathcal{D} = \{(+1)\}$, whereas the dependence structure of the second loop in isolation can be represented by $\mathcal{D} = \{(+1,-1)\}$.

3.1.6 Data Dependence Analysis

Data dependence analysis consists of determining the data dependence structure of a program. In general, the problem of computing all data dependences at compile-time is undecidable. However, many methods have been developed to determine the data dependences between arrays with admissible subscripts.

Data Dependence System

Let $IS \subseteq \mathcal{Z}^{d_i}$ and $JS \subseteq \mathcal{Z}^{d_j}$ denote the iteration spaces of two loops in which respectively statements S_i and S_j occur. Let $d \leq \min(d_i, d_j)$ denote the common nesting depth of S_i and S_j. If both statements appear in the loop-body of the same loop, then we have $IS = JS$. Otherwise, the d outermost DO-loops are shared by both statements, whereas the other DO-loops are not.

Now suppose that both statements access a particular c-dimensional array A. If a write operation to this array occurs in S_i and a read operation in S_j, then there is a potential flow dependence from S_i to S_j and a potential anti dependence from S_j to S_i, depending on the execution order of instances that access the same element. In the opposite case, there is a potential flow dependence from S_j to S_i or potential anti dependence from S_i to S_j, whereas there is a potential output dependence in either direction if write operations occur in both statements.

If the subscripts of A in both statements are admissible and represented by affine transformations $F : \mathcal{Z}^{d_i} \to \mathcal{Z}^2$ and $F' : \mathcal{Z}^{d_j} \to \mathcal{Z}^2$ which are denoted in matrix form as $F(\vec{\mathtt{I}}) = \vec{v} + W\vec{\mathtt{I}}$ and $G(\vec{\mathtt{J}}) = \vec{x} + Y\vec{\mathtt{J}}$ respectively, where $\vec{\mathtt{I}}$ and $\vec{\mathtt{J}}$ denote the index vectors of the two loops, then both statements may access the same element if there exists integer vectors $\vec{\imath} \in \mathcal{Z}^{d_i}$ and $\vec{\jmath} \in \mathcal{Z}^{d_j}$ that satisfy $F(\vec{\imath}) = G(\vec{\jmath})$:[2]

$$\begin{pmatrix} W & -Y \end{pmatrix} \begin{pmatrix} \vec{\imath} \\ \vec{\jmath} \end{pmatrix} = \vec{x} - \vec{v} \tag{3.5}$$

Additional constraints arise from the fact that we must search for solutions $\vec{\imath} \in IS$ and $\vec{\jmath} \in JS$. All admissible loop bounds give rise to linear inequalities. Other inequalities arise if we test for a data dependence with a particular dependence direction vector. Eventually, these additional constraints can be expressed as an integer system of linear inequalities.

$$A \begin{pmatrix} \vec{\imath} \\ \vec{\jmath} \end{pmatrix} \leq \vec{b} \tag{3.6}$$

Together, the systems (3.5) and (3.6) form the **data dependence system**. If no integer solutions exist, then the statements S_i and S_j are independent (at least, with respect to the two occurrences of array A). Otherwise, there is a data dependence between the two statements. Hence, in essence data dependence analysis is equivalent to integer linear programming [61, 151]. If some subscripts or loop bounds are inadmissible, we may conservatively omit the corresponding equations or inequalities from the data dependence system.

Data Dependence System Solvers

Because solving linear integer programming programs exactly may be infeasible in practice, many data dependence systems solvers have been developed for special and more general cases, varying from exact tests (providing a necessary and sufficient condition for data dependence) to approximate tests (only providing a necessary condition for data dependence). Although solvers may have a different trade-off between accuracy and efficiency, all solvers must be *conservative*, i.e. if independence cannot be proven, then data dependence must be assumed.

One way to solve a data dependence system, for example, is to use proposition 2.2 to solve the system of equations with an integer echelon reduction algorithm (generalized GCD-test), followed by using Fourier-Motzkin elimination to solve the system of inequalities obtained by substituting the general form of the solution for the variables in the original system. In general, this reduces the number of inequalities and variables [151].

Other solvers, such as the GCD- or bounds-test consider equations separately, possibly accounting for the inequalities. Note that if solvers only deal with one equation at the time, dependence analysis for multi-dimensional arrays can be handled by considering all admissible subscripts separately after which the solution sets are intersected, or by linearizing the subscripts first.

In [47], a method to reduce the number of applications of a solver is presented. Rather than, for instance, calling the solver for every plausible dependence direction vector in two directions,

[2] Note that although the index vectors $\vec{\mathtt{I}}$ and $\vec{\mathtt{J}}$ have the first d loop indices in common, the actual values of the corresponding components in $\vec{\imath}$ and $\vec{\jmath}$ may differ.

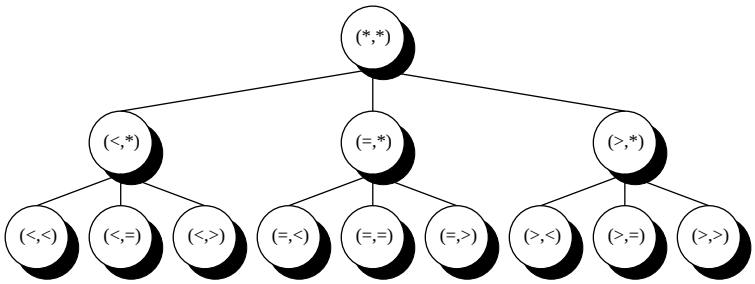


Figure 3.2: Hierarchical Dependence Testing

i.e. testing for dependences from S_i to S_j and from S_j to S_i, we test for dependences in both directions simultaneously in a hierarchical manner by successively refining dependence direction vectors. First, we start with the dependence direction vector $(*, \ldots, *)$, which simply imposes no additional constraints on the variables. If independence cannot be proven, one of the components is refined into the directions '$<$', '$=$', and '$>$'. In this manner, a tree of dependence direction vectors is constructed, as illustrated in figure 3.2 for a dependence direction vector with two components. If independence can be shown for a particular dependence direction vector, however, no further refinements are required, which effectively prunes the subtree rooted at that dependence direction vector. If we arrive at a leaf, the appropriate static data dependence is recorded, possibly after reversing an implausible dependence direction vector from S_i to S_j into a plausible dependence direction vector from S_j to S_i and reversing the nature of the data dependence that would result from S_i to S_j according to flow $\rightarrow$ anti, anti $\rightarrow$ flow, and output $\rightarrow$ output.

This approach becomes even more efficient, if information required by the solver algorithm can be constructed incrementally during traversal of the tree. Moreover, in some cases the way in which the tree is expanded may affect the number of times the solver is called.

Example: Consider the following double loop:

```
   DO I = 1, 100
S1:  A(I) = ...
S2:  ...  = A(I+1)
   ENDDO
```

In this loop, there is a potential data dependence between the statement instances $S_1(i)$ and $S_2(j)$, because these instances may refer to the same element of array A as implied by the following dependence system:

$$\begin{cases} i & = & j+1 & & \\ 1 & \leq & i & \leq & 100 \\ 1 & \leq & j & \leq & 100 \end{cases}$$

Therefore, we consider the three dependence systems that result after adding the constraints $i < j$, $i = j$ and $i > j$ respectively. Obviously, only the last dependence system has solutions. Because the write occurs in S_1, we may say that there is a static flow dependence $S_1 \delta_> S_2$ with an implausible dependence direction vector, which is reversed into the static anti dependence $S_2 \overline{\delta}_< S_1$ to account for the fact that $S_2(j) <_o S_1(i)$.

Effective data dependence analysis should also account for the flow of control in a program. For example, loop-independent data dependences between instances of statements appearing in different branches of a multi-way IF-statements cannot occur. In addition, in the presence of subroutines and functions, interprocedural data dependence analysis is required.

A more detailed presentation of data dependence analysis and data dependence system solvers, however, is beyond the scope of this dissertation but can be found in the literature (see e.g. [15, 16, 17, 23, 47, 82, 109, 127, 142, 151, 155, 166, 170, 228, 229, 234])

3.2 Exploitation of Implicit Parallelism

Although the sequential semantics of loops induce a total order on statement instances, the execution order of individual DO-loops may be relaxed without changing the semantics as long as all data dependences are preserved. Since numerical programs spend most execution time inside loops, a substantial speed-up may be expected from exploiting such implicit parallelism.

In this section, we briefly discuss how a compiler can convert implicit parallelism into explicit parallel constructs by loop vectorization and loop concurrentization. A more detailed discussion of this topic and other restructuring transformations can be found in the literature (see e.g. [4, 5, 41, 47, 48, 64, 101, 86, 127, 128, 132, 135, 147, 152, 158, 165, 166, 170, 217, 228, 227, 229, 234]).

3.2.1 Loop Vectorization

The automatic conversion of serial DO-loops into semantically equivalent vector statements is referred to as **vectorization**.

Vector Statements

The main transformation for vectorization is the conversion of a DO-loop in which a single assignment appears into a vector statement, which is made explicit in the text using subscript triplets. Each vector statement is subject to **FS-semantics** (fetch before store-semantics), which means that all right-hand side elements are fetched before any of the left-hand side elements are stored. Hence, any self-anti dependence may be ignored.

Under the assumption that self-output dependences also may be ignored because stores are executed deterministically, vectorization of a single assignment statement in a DO-loop is valid, *if no flow dependence is carried by the DO-loop*.

Example: Because below, static dependences $S_1 \delta_< S_1$ and $S_2 \overline{\delta}_< S_2$ hold, only statement S_2 may be vectorized (we always assume that the final value of a loop index is not used after the DO-loop, making a last-value-assignment to restore the value of the loop index unnecessary):

```
   DO I = 2, N
S1:  A(I) = A(I-1) * 2.0
   ENDDO
   DO I = 1, N-1
S2:  B(I) = B(I+1) * 2.0
   ENDDO
```

→

```
DO I = 2, N
  A(I) = A(I-1) * 2.0
ENDDO
B(1:N-1:1) = B(2:N:1) * 2.0
```

Generation of Vector Statements

Two basic forms of loop transformations are essential for the effective generation of vector statements. The first transformation, called **loop distribution**, converts a single DO-loop with a loop-body that is partitioned into adjacent blocks `B1` and `B2` into two adjacent DO-loops with the loop-bodies `B1` and `B2` respectively.

Distribution of a DO-loop is valid, *if no lexically backward data dependence from an instance of a statement in* `B2` *to an instance of a statement in* `B1` *is carried by the DO-loop*:

```
DO I = 1, N
  B1(I)
  B2(I)
ENDDO
```

→

```
DO I = 1, N
  B1(I)
ENDDO
DO I = 1, N
  B2(I)
ENDDO
```

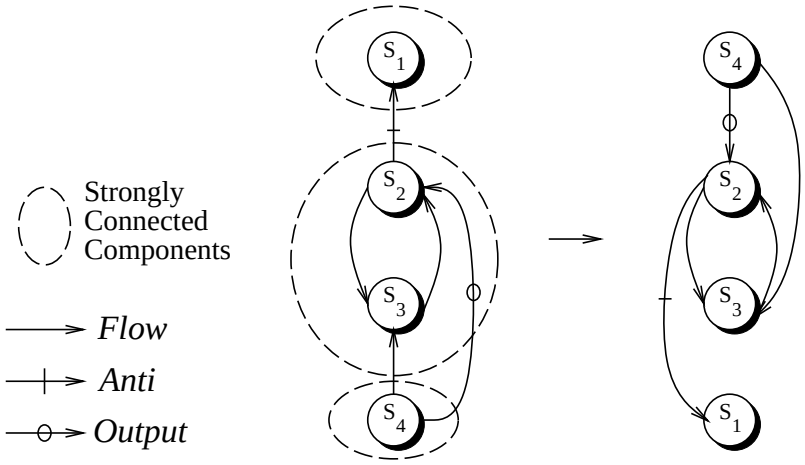


Figure 3.3: Data Dependence Graph

The second transformation is referred to as **statement reordering**. Given two adjacent statements, simply reordering statements S_i and S_{i+1} textually in the program is valid, *if there is no loop-independent data dependence from an instance of S_i to an instance of S_{i+1}*.

Given the dependence graph of a loop, vectorization may proceed as follows. First, the acyclic condensation of the dependence graph is constructed, in which vertices correspond to strongly connected components. Subsequently, the statements in the loop-body are reordered according to a topological sort of the acyclic condensation, where statements in the same strongly connected components become adjacent. Now, there are only lexically forward dependences between instances of statements in different strongly connected components, and loop distribution can be used to isolate these strongly connected components.

Statements involved in a multi-statement data dependence cycle remain in a serial loop. If valid, all other statements are vectorized, or possibly recognized as particular idiom that can be efficiently implemented in some manner, such as a reduction [228, ch7][234, p235-237]. For example, although a flow dependence is carried by the following DO-loop, the whole construct may be recognized as a summation:

```
DO I = 1, N
  S = S + A(I)      →    S = S + SUM( A(1:N) )
ENDDO
```

Vectorizing arithmetic reductions is only valid if roundoff errors, which are due to inexact computer arithmetic, are allowed to accumulate in a different order [135, ch4][228, ch7].

Example: Based on a topological sort of the acyclic condensation of the dependence graph shown in figure 3.3, the following loop is vectorized as shown below:

```
   DO I = 1, N-1                                  S4:B(2:N:1) = 5.0
S1:  A(I)   = 10.0                                   DO I = 1, N-1
S2:  B(I)   = A(I+1) * C(I)          →           S2:  B(I)   = A(I+1) * C(I)
S3:  C(I+1) = B(I)                               S3:  C(I+1) = B(I)
S4:  B(I+1) = 5.0                                    ENDDO
   ENDDO                                         S1:A(1:N-1:1) = 10.0
```

Nested loops can be vectorized in a similar way by ignoring data dependences that are carried by more outer DO-loops during vectorization of inner DO-loops. Details about vectorization can be found in [5, 135, 166, 170, 227][228, ch3][229, ch10][234, ch6].

3.2.2 Loop Concurrentization

Executing the different iterations of a DO-loop on different processors of a shared memory multiprocessor may result in a substantial reduction of execution time. Two kinds of concurrent DO-loops can be distinguished [59, 60, 166, 171][170, p13][228, ch4][234, ch7]. A DOALL-loop is used if all iterations are independent and can be executed concurrently. A DOACROSS-loop is used if a partial execution order on some (parts) of the iterations must be imposed. In the latter case, synchronization between the execution of different iterations is required to meet this ordering constraint.

DOALL-Loop

A DO-loop can be converted into a DOALL-loop if all iterations of the DO-loop are independent, i.e. *if no data dependence is carried by this DO-loop*. Issues related to the automatic concurrentization of loops are addressed in [48, 66][228, ch4][229, ch11][234, ch7].
Example: Since only $S_1\delta_{<,<}S_1$ holds in the following double loop, no data dependence is carried by the innermost DO-loop and all iterations of this DO-loop may be executed in parallel:

```
   DO I = 2, N                                DO I = 2, N
     DO J = 2, N                                DOALL J = 2, N
S1:  A(I,J) = A(I-1,J-1) + 5.0      →             A(I,J) = A(I-1,J-1) + 5.0
    ENDDO                                      ENDDO
   ENDDO                                      ENDDO
```

DOACROSS-Loop

To impose a partial execution order on some (parts) of the iterations of a DOACROSS-loop, synchronization between the execution of different iterations is required. In [59, 60], this synchronization is modeled under the assumption that processors operate synchronously by using a particular delay $d \geq 0$ between consecutive iterations of the DOACROSS-loop, i.e. the ith iteration is executed after a delay of $(i-1) \cdot d$. Alternatively, synchronization can be enforced using the primitives **testset**/**test** (or **advance**/**await**) [155, ch6][156, 157][229, p393-395] which can be implemented with a single synchronization variable [228, p84-86].

Here, we focus on more general **random synchronization** with the primitives **post**/**wait** [228, p75-83][234, p289-295]. In a busy-waiting implementation, each **post** is a non-blocking operation that sets a unique bit on which completion of a corresponding **wait** depends.

If data dependences are carried by a DO-loop that is converted into a DOACROSS-loop, a **post**-statement is placed directly *after* the source statement of the data dependence, while a corresponding **wait**-statement is placed *before* the sink statement of the data dependence. Different synchronization variables are used for the synchronization of different static data dependences, while other parameters are used to distinguish between the different underlying data dependences of each individual static data dependence. The automatic generation of synchronization and the elimination of redundant synchronization is addressed in [141, 155, 156, 157, 234].
Example: If the following DO-loop is converted into a DOACROSS-loop, then all underlying data dependences of $S_1\delta_{<}S_2$ with fixed dependence distance 4 can be enforced by instances of the given **wait**- and **post**-statements (**wait** does not block on out-of-bounds iterations):

```
   DO I = 1, N-4                          DOACROSS I = 1, N-4
S1:  A(I+4) = ...                           A(I+4) = ...
S2:  ...    = A(I)          →               post(ASYNC,I)
   ENDDO                                    wait(ASYNC,I-4)
                                            ...    = A(I)
                                          ENDDOACROSS
```

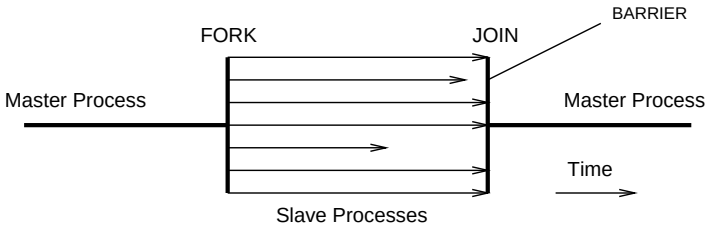


Figure 3.4: Execution of a Concurrent-Loop

The synchronization variable ASYNC has one parameter, so it can be implemented as a one-dimensional bit array with one bit for each iteration.

In general, however, several parameters may be necessary to distinguish between different instances of a single source statement. In these cases, the synchronization variable must be implemented as a multi-dimensional bit array. However, because we assume that several instances of a **wait**-statement may test the same bit, it is not necessary to distinguish between different instances of a single sink statement.

Concurrent Loop Scheduling

We assume that concurrent loops are executed using fork/join-like parallelism, illustrated in figure 3.4, where a master process executes the serial part of the program and initiates a number of slave processes when a concurrent loop is reached [229, 385-387]. After all iterations of this loop have been executed, the slave processes synchronize using barrier synchronization, and the master process continues execution of the serial code after the concurrent loop. Whether each slave process is actually executed on a physical processor and whether slaves are terminated or simply parked at the barrier depends on the operating system used.

The way in which iterations of a concurrent loop are assigned to the slave processes depends on the scheduling policy. We can use **pre-scheduling**, where either a block of consecutive iterations is assigned statically to each slave process (block-scheduling), or iterations are assigned statically in a cyclic fashion to the slave processes (cyclic scheduling). To reduce the potential of load imbalance, we can also use **self-scheduling**, where each slave processes enters a critical section to dynamically obtain a next chunk of iterations to be executed. A small chunk size probably yields good load balancing at the expense of much synchronization overhead, whereas a large chunk size decreases synchronization overhead at the expense of potential load imbalance. A good comprise is to vary the chunk size dynamically, such as assigning $1/p$ of the remaining iterations to each next slave process in case there are p slave processes (guided self-scheduling). These scheduling policies are discussed in more detail in [170, ch4][228, p73-74][229, p387-392][234, 296-298].

3.3 Unimodular Loop Transformations

A major step forward in solving the phase ordering problem has been accomplished by the observation that any combination of the iteration-level loop transformations loop interchanging, loop skewing [226] and loop reversal can be represented by a unimodular transformation [18, 19, 71, 224, 225]. The advantage of this approach is that the order and validity of individual transformations becomes irrelevant, because a unimodular transformation can be constructed directly for a particular goal provided that data dependence constraints are accounted for.

3.3.1 Iteration-Level Loop Transformations

An iteration-level loop transformation transforms a perfectly nested loop with stride-1 DO-loops, index vector $\vec{\mathtt{I}} = (\mathtt{I}_1, \ldots, \mathtt{I}_d)^T$, and iteration space $IS \subseteq \mathcal{Z}^d$ into another perfectly nested loop having index vector $\vec{\mathtt{I}}' = (\mathtt{I}'_1, \ldots, \mathtt{I}'_d)^T$ and iteration space $IS' \subseteq \mathcal{Z}^d$.

The former loop and $IS \subseteq \mathcal{Z}^d$ are referred to as the **original loop** and the **original iteration space** respectively. The latter loop and $IS' \subseteq \mathcal{Z}^d$ are called the **target loop** and the **target iteration space**. Each iteration-level loop transformation consisting of a combination of loop interchanging, loop skewing, and loop reversal can be represented by a linear transformation $F : IS \rightarrow IS'$ that is defined by a $d \times d$ unimodular matrix U. An iteration $\vec{\imath} \in IS$ in the original iteration space is mapped to an iteration $\vec{\imath}' \in IS'$ in the target iteration space as follows:

$$\vec{\imath}' = F(\vec{\imath}) = U\vec{\imath}$$

Because iterations in both the original and target iteration space are traversed in lexicographical order, a unimodular transformation changes the order in which iterations are executed. In the original loop, a particular iteration $\vec{\imath} \in IS$ is executed before another iteration $\vec{\jmath} \in IS$ if we have $\vec{\imath} \prec \vec{\jmath}$, whereas in the target loop the iteration corresponding to the former is executed before the iteration corresponding to the latter if we have $U\vec{\imath} \prec U\vec{\jmath}$. Application of an elementary *integer* row operation (cf. section 2.2.3) to the $d \times d$ identity matrix yields an elementary matrix that defines a single loop reversal, loop interchanging, or loop skewing.
Example: Consider the following double loop:

```
DO I1 = 1, 100
  DO I2 = 1, 50
    B(I1,I2)
  ENDDO
ENDDO
```

The target loops that result after application of the unimodular transformations defined by the following 2×2 elementary matrices are shown below:

Reversal:

$$\begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}$$

```
DO I'1 = -100, -1
  DO I'2 = 1, 50
    B(-I'1,I'2)
  ENDDO
ENDDO
```

Interchanging:

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

```
DO I'1 = 1, 50
  DO I'2 = 1, 100
    B(I'2,I'1)
  ENDDO
ENDDO
```

Skewing:

$$\begin{pmatrix} 1 & 0 \\ p & 1 \end{pmatrix}$$

```
DO I'1 = 1, 100
  DO I'2 = 1+p*I'1, 50+p*I'1
    B(I'1,I'2-p*I'1)
  ENDDO
ENDDO
```

Any combination of loop interchanging, loop skewing and loop reversal can be represented by a single unimodular transformation. Conversely, each unimodular matrix can be decomposed into a finite number of elementary matrices [19, p40-45] and, hence, loop transformations. Consequently, this approach offers more flexibility than the traditional step-wise application of loop transformations, where the usefulness and validity of each individual transformation must be considered separately.

3.3.2 Validity of Application

Application of a unimodular transformation is valid if the semantics of the original loop are preserved. Therefore, we must verify whether the data dependences arising in the original loop are still satisfied in the target loop (where, as usual, we assume that the final value of loop indices is immaterial).

Dependence Distance Vectors

Let the set $\mathcal{D} \subseteq \mathcal{Z}^d$ of dependence distance vectors represent the data dependence structure of the original loop, i.e. if a statement instance executed in iteration $\vec{j} \in IS$ depends on a statement instance executed in iteration $\vec{i} \in IS$, then for $\vec{d} = \vec{j} - \vec{i}$, we have $\vec{d} \in \mathcal{D}$.

Because U defines a linear transformation, applying a transformation defined by a unimodular matrix U to this loop affects the dependence distance between the previous two iterations as shown below:

$$\vec{d}' = U\vec{i}' - U\vec{i} = U(\vec{i}' - \vec{i}) = U\vec{d}$$

Consequently, application of a unimodular transformation defined by U to a loop of which the data dependence structure is represented by a set of dependence distance vectors $\mathcal{D} \subseteq \mathcal{Z}^d$ is valid *if and only if* $U\vec{d} \succeq \vec{0}$ holds for all $\vec{d} \in \mathcal{D}$.

Dependence Direction Vectors

To deal with the general case, both dependence distance and direction vectors are represented by **dependence vectors** [224, 225]. Each component of a dependence vector $\vec{d}$ consists of a range $[d_i^{\min}, d_i^{\max}]$ that is described by two bounds $d_i^{\min}, d_i^{\max} \in \mathcal{Z} \cup \{-\infty, +\infty\}$. Components of a dependence direction vector are translated to components of a dependence vector as follows:

$$\begin{array}{lcllcl} > & \equiv & [-\infty, -1] & * & \equiv & [-\infty, +\infty] \\ < & \equiv & [+1, +\infty] & = & \equiv & [0, 0] \end{array}$$

A component of a dependence distance vector is represented by a degenerate interval $[d_i, d_i]$.

Given a dependence vector $\vec{d}$ with d components, then $\vec{d} \succ \vec{0}$ holds if there is a $1 \leq i \leq d$ such that $d_i^{\min} > 0$ and $d_j^{\min} = 0$ for all $1 \leq j < i$. Moreover, $\vec{d} \succeq \vec{0}$ holds if either $\vec{d} \succ \vec{0}$ or $d_i^{\min} \geq 0$ for all $1 \leq i \leq d$. Two ranges are added as follows, where $\infty + s = \infty$ for any $s \neq -\infty$ and $-\infty + s = -\infty$ for any $s \neq \infty$:

$$[l, u] + [l', u'] = [l + l', u + u']$$

Multiplication of a range by a scalar $s \in \mathcal{Z}$ is defined below, where $s \cdot \pm\infty = 0$ if $s = 0$, and $s \cdot \pm\infty = \pm\infty$ has the appropriate sign for $s \neq 0$:

$$s \cdot [l, u] = \begin{cases} [s \cdot l, s \cdot u] & \text{if } s \geq 0 \\ [s \cdot u, s \cdot l] & \text{otherwise} \end{cases}$$

Applying a unimodular transformation defined by U to a loop of which the data dependence structure is represented by a set $\mathcal{D}$ of dependence vectors with $\vec{d} \succeq \vec{0}$ for all $\vec{d} \in \mathcal{D}$ is valid, if $U\vec{d} \succeq \vec{0}$ also holds for all $\vec{d} \in \mathcal{D}$ under the previous defined arithmetic. The converse implication does not hold, because loop skewing may cause loss of data dependence information.

3.3.3 Code Generation

The application of a loop transformation defined by a unimodular matrix U to a loop nest with index vector $\vec{\mathtt{I}}$ is implemented by replacing the original loop with the target loop. Effectively, this replacement is implemented by (i) rewriting the loop-body of the original loop, and (ii) generating new loops with index vector $\vec{\mathtt{I}}'$ that induce a lexicographical traversal of the target iteration space. Here, we assume that all loop bounds of the original loop are *admissible*.

Replacement of Original Loop with Target Loop

Because the index vectors of the target loop and original loop are related according to $\vec{\mathtt{I}}' = U\vec{\mathtt{I}}$, step (i) is simply performed by replacing each loop index in the original loop-body according to the equation $\vec{\mathtt{I}} = U^{-1}\vec{\mathtt{I}}'$.

Moreover, since the original iteration space can be defined in terms of a system of linear inequalities and all discrete points in the image of a polyhedral set under a unimodular transformation uniquely correspond to a discrete point in that set (see section 2.1.5), a representation of the target iteration space is obtained as shown below:

$$IS = \{\vec{\mathtt{I}} \in \mathcal{Z}^d \mid A\vec{\mathtt{I}} \leq \vec{b}\} \quad \stackrel{U}{\rightarrow} \quad IS' = \{\vec{\mathtt{I}}' \in \mathcal{Z}^d \mid AU^{-1}\vec{\mathtt{I}}' \leq \vec{b}\}$$

Hence, step (ii) consists of generating loops that induce a lexicographical traversal of all discrete points $\vec{\imath}' \in \mathcal{Z}^d$ that satisfy $AU^{-1}\vec{\imath}' \leq \vec{b}$.

Example: Consider application of loop interchanging to the following double loop:

```
DO I1 = 1, 3
  DO I2 = I1+1, 4
    B(I1,I2)
  ENDDO
ENDDO
```

$$U = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

$$\rightarrow$$

```
DO I1' = 2, 4
  DO I2' = 1, I1'-1
    B(I2',I1')
  ENDDO
ENDDO
```

The loop-body of the target loop is obtained by replacing the loop indices according to the equation $\vec{\mathtt{I}} = U^{-1}\vec{\mathtt{I}}'$:

$$\begin{pmatrix} \mathtt{I}_1 \\ \mathtt{I}_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} \mathtt{I}'_1 \\ \mathtt{I}'_2 \end{pmatrix} \tag{3.7}$$

Unfortunately, generating the loop bounds of the target loop is less straightforward. A first step towards finding these bounds is to apply the substitution defined by equation (3.7), that is $\mathtt{I}_1 = \mathtt{I}'_2$ and $\mathtt{I}_2 = \mathtt{I}'_1$, to the system of inequalities representing the loop bounds of the original loop:

$$\begin{array}{rcl} 1 \leq & \mathtt{I}_1 & \leq 3 \\ 1 + \mathtt{I}_1 \leq & \mathtt{I}_2 & \leq 4 \end{array} \quad \rightarrow \quad \begin{array}{rcl} 1 \leq & \mathtt{I}'_2 & \leq 3 \\ 1 + \mathtt{I}'_2 \leq & \mathtt{I}'_1 & \leq 4 \end{array}$$

The bounds of the *innermost* loop can be determined directly:

$$1 \leq \mathtt{I}'_2 \leq \mathtt{MIN}(3, \mathtt{I}'_1 - 1)$$

However, inequality $1 + \mathtt{I}'_2 \leq \mathtt{I}'_1$ cannot be used directly to determine the lower bound of index $\mathtt{I}'_1$ because this bound is given in terms of the innermost loop index $\mathtt{I}'_2$. First, index $\mathtt{I}'_2$ must be eliminated from the system. This is performed by replacing all inequalities involving $\mathtt{I}'_2$ by inequalities in which each lower bound of this index is less than or equal to each upper bound of this index. In the example, we obtain:

$$\begin{array}{rcl} 1 \leq & \mathtt{I}'_2 & \\ & & \end{array} \quad \begin{array}{rcl} \mathtt{I}'_2 & \leq & 3 \\ \mathtt{I}'_2 & \leq & \mathtt{I}'_1 - 1 \\ \mathtt{I}'_1 & \leq & 4 \end{array} \quad \rightarrow \quad \begin{array}{rcl} 1 & \leq & 3 \\ 1 & \leq & \mathtt{I}'_1 - 1 \\ \mathtt{I}'_1 & \leq & 4 \end{array}$$

Consequently, the lower and upper bound of $\mathtt{I}'_1$ can be expressed as 2 and 4 respectively, which is the appropriate form for the bounds of an outermost loop. At this point the valid range for index $\mathtt{I}'_1$ is known, and the upper bound of index $\mathtt{I}'_2$ can be simplified into $\mathtt{I}'_1 - 1$. By enumerating all instances of the loop-body that are executed in the original and new loop, we can easily verify that both loops execute the same instances, but only in a different order:

Original Loop:

```
B(1,2) B(1,3) B(1,4)
       B(2,3) B(2,4)
              B(3,4)
```

Target Loop:

```
B(1,2)
B(1,3) B(2,3)
B(1,4) B(2,4) B(3,4)
```

Fourier-Motzkin Elimination

In general, the system of inequalities $AU^{-1}\vec{\mathtt{I}} \leq \vec{b}$ describing the target iteration space is unsuited to generate the loop bounds of the target loop directly, because the bounds of a particular index may be defined in terms of indices of more inner DO-loops. However, as advocated in [10], Fourier-Motzkin elimination can be used to convert the system in the appropriate form.

Starting with the column augmented integer matrix representation $(A^{(d)} \mid \vec{b}^{\,(d)})$ of the target iteration space, where $A^{(d)} = AU^{-1}$ and $\vec{b}^{\,(d)} = \vec{b}$, the sequence (2.10) is generated. Each column augmented matrix $(A^{(k)} \mid \vec{b}^{\,(k)})$ defines the lower and upper bounds of $\mathtt{I}'_k$ according to the inequalities (2.11). If only one lower or upper bound results (viz. $p^{(k)} = 1$ or $q^{(k)} = p^{(k)} + 1$), then the maximum or minimum function is omitted. Ceiling and floor functions are omitted for all lower or upper bounds having $a_{ik}^{(k)} = 1$.

The ad-hoc and exact simplification method can be used to eliminate redundant inequalities and, hence, redundant loop bounds, which improves the efficiency of the generated code by reducing evaluation overhead.

Example: Applying Fourier-Motzkin elimination to the system of inequalities that define the target iteration space of the example presented in the previous section yields the following sequence of matrices:

$$\left(\begin{array}{rr|r} -1 & 1 & -1 \\ 0 & 1 & 3 \\ 0 & -1 & -1 \\ 1 & 0 & 4 \end{array}\right) \rightarrow \left(\begin{array}{r|r} 1 & 4 \\ -1 & -2 \\ 0 & 2 \end{array}\right) \rightarrow \left(\begin{array}{c} 2 \\ 2 \end{array}\right)$$

The ad-hoc simplification method can eliminate all redundant inequalities:

$$\left(\begin{array}{rr|r} -1 & 1 & -1 \\ 0 & -1 & -1 \end{array}\right) \rightarrow \left(\begin{array}{r|r} 1 & 4 \\ -1 & -2 \end{array}\right)$$

Effectively, these simplifications correspond to the following program transformations:

```
DO I'1 = 2, 4                          DO I'1 = 2, 4
  DO I'2 = 1, MIN(I'1-1, 3)              DO I'2 = 1, I'1-1
    B(I'2,I'1)                    →          B(I'2,I'1)
  ENDDO                                  ENDDO
ENDDO                                  ENDDO
```

Example: Consider, as another example, applying the transformation represented by the following unimodular matrix U to the loop nest shown below:

```
DO I1 = 10, 15
  DO I2 = 1, 3
    DO I3 = 1, 50
      B(I1,I2,I3)
    ENDDO
  ENDDO
ENDDO
```

$$U = \begin{pmatrix} 0 & 6 & 1 \\ 1 & -3 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

The loop-body of the target loop is obtained by replacing the original loop indices according to the equation $\vec{\mathtt{I}} = U^{-1}\vec{\mathtt{I}}'$. The target iteration space is represented by the column augmented matrix $(AU^{-1} \mid \vec{b})$, where $(A \mid \vec{b})$ represents the original iteration space:

$$AU^{-1} = \begin{pmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 & 3 \\ 0 & 0 & 1 \\ 1 & 0 & -6 \end{pmatrix} \quad \vec{b} = \begin{pmatrix} -10 \\ -1 \\ -1 \\ 15 \\ 3 \\ 50 \end{pmatrix}$$

At page 22, the sequence of column augmented matrices arising from Fourier-Motzkin elimination has already been presented. Eventually, the following code is generated:

```
DO I'1 = 7, 68
  DO I'2 = MAX(1, ⌈(21-I'1)/2⌉), MIN(12, ⌊80-I'1)/2⌋)
     DO I'3 = MAX(1, ⌈(10-I'2)/3⌉, ⌈(I'1-50)/6⌉), MIN(3, ⌊(15-I'2)/3⌋, ⌊(I'1-1)/6⌋)
      B(I'2+3*I'3,I'3,I'1-6*I'3)
    ENDDO
  ENDDO
ENDDO
```

Example: In [225], application of the unimodular transformation defined by the U shown below to the following triple loop is considered:

```
DO I1 = 1, 100
  DO I2 = 2*I1, 100
    DO I3 = 2*I1+I2-1, MIN(I2, 100)
      B(I1,I2,I3)
    ENDDO
  ENDDO
ENDDO
```

$$U = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

Applying Fourier-Motzkin elimination to the column augmented matrix $(AU^{-1} \mid \vec{b})$, representing the target iteration space, results in the following sequence:

$$\left(\begin{array}{ccc|c} 0 & 0 & 1 & 100 \\ 0 & -1 & 2 & 0 \\ -1 & 1 & 2 & 1 \\ 0 & 0 & -1 & -1 \\ 0 & 1 & 0 & 100 \\ 1 & -1 & 0 & 0 \\ 1 & 0 & 0 & 100 \end{array}\right) \rightarrow \left(\begin{array}{cc|c} -1 & 1 & -1 \\ 0 & 1 & 100 \\ 0 & -1 & -2 \\ 1 & -1 & -2 \\ 1 & -1 & 0 \\ 0 & 0 & 99 \\ 1 & 0 & 100 \end{array}\right) \rightarrow \left(\begin{array}{c|c} 1 & 100 \\ 1 & 100 \\ -1 & -3 \\ 0 & 98 \\ 0 & 99 \\ 0 & -1 \end{array}\right) \rightarrow \begin{pmatrix} 97 \\ 97 \\ 98 \\ 99 \\ -1 \end{pmatrix}$$

Because one of the components of the column vector is negative, the system of inequalities is inconsistent. This implies that both the target and original iteration space are empty. Indeed, careful examination reveals that the original nest is a zero-trip loop.

3.3.4 Construction of a Unimodular Loop Transformation

Applying an iteration-level loop transformation may change the order in which iterations are executed. In this section, constructing a unimodular transformation that affects this order in some desired manner is explored.

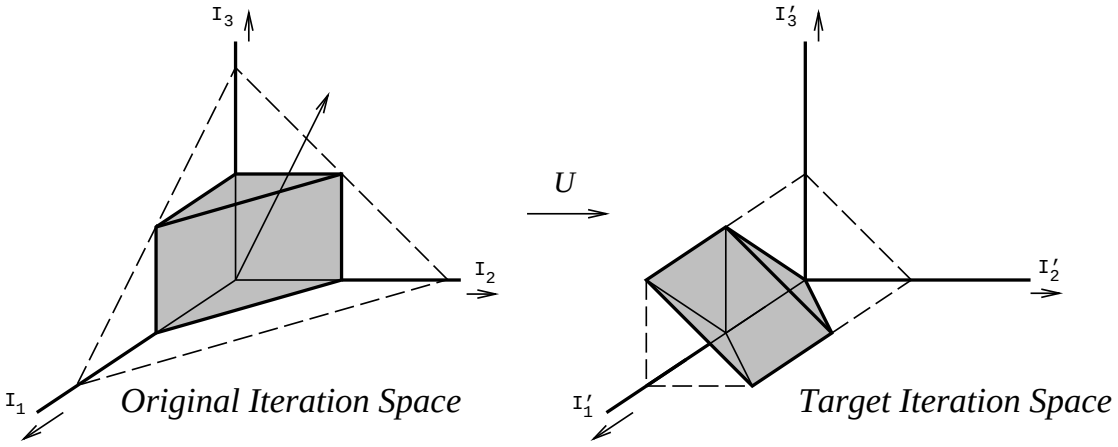


Figure 3.5: Application of a Unimodular Transformation

Hyperplane Traversal

Applying a linear transformation $F : IS \rightarrow IS'$ defined by a $d \times d$ unimodular matrix U gives rise to the system of equations $\vec{\mathtt{I}}' = U\vec{\mathtt{I}}$:

$$\left\{ \begin{array}{rcl} \mathtt{I}'_1 & = & u_{11} \cdot \mathtt{I}_1 + \ldots + u_{1d} \cdot \mathtt{I}_d \\ & \vdots & \\ \mathtt{I}'_d & = & u_{d1} \cdot \mathtt{I}_1 + \ldots + u_{dd} \cdot \mathtt{I}_d \end{array} \right. \tag{3.8}$$

The ith row of matrix U forms the normal vector of a hyperplane in $\mathcal{R}^d$

$$H_i^U(\mathtt{I}'_i) = \{\vec{\mathtt{I}} \in \mathcal{R}^d \mid u_{i1} \cdot \mathtt{I}_1 + \ldots + u_{id} \cdot \mathtt{I}_d = \mathtt{I}'_i\}$$

Consequently, for fixed $\mathtt{I}'_1 = i'_1, \ldots, \mathtt{I}'_k = i'_k$, the other DO-loops of the target loop execute all iterations in IS' that are in the image of the following set under F:

$$H_1^U(i'_1) \cap \ldots \cap H_k^U(i'_k) \cap IS$$

Because the rows of a unimodular matrix are linearly independent, the intersection of hyperplanes $H_1^U(i'_1) \cap \ldots \cap H_k^U(i'_k)$ forms a $(d-k)$-dimensional affine subspace of $\mathcal{R}^d$. All discrete points in this affine subspace that belong to IS are mapped by F to iterations in IS' that are executed for fixed $\mathtt{I}'_1 = i'_1, \ldots, \mathtt{I}'_k = i'_k$.

Example: Consider the following unimodular transformation:

```
DO I1 = 0, 50
  DO I2 = 0, 50 - I1
    DO I3 = 0, 50
      B(I1,I2,I3)
    ENDDO
  ENDDO
ENDDO
```

$$U = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

$\rightarrow$

```
DO I'1 = 0, 100
  DO I'2 = 0, MIN(50,I'1)
    DO I'3 = MAX(0,I'1-I'2-50),
+                MIN(50-I'2,I'1-I'2)
      B(I'2,I'3,I'1-I'2-I'3)
    ENDDO
  ENDDO
ENDDO
```

The transformation of the original iteration space IS into the target iteration space IS' is illustrated in figure 3.5. The rows of U give rise to three planes of the following form:

$$\left\{ \begin{array}{rcl} H_1^U(\mathtt{I}'_1) & = & \{ \ \vec{\mathtt{I}} \in \mathcal{R}^3 \mid \mathtt{I}_1 + \mathtt{I}_2 + \mathtt{I}_3 = \mathtt{I}'_1 \ \} \\ H_2^U(\mathtt{I}'_2) & = & \{ \ \vec{\mathtt{I}} \in \mathcal{R}^3 \mid \mathtt{I}_1 = \mathtt{I}'_2 \ \} \\ H_3^U(\mathtt{I}'_3) & = & \{ \ \vec{\mathtt{I}} \in \mathcal{R}^3 \mid \mathtt{I}_2 = \mathtt{I}'_3 \ \} \end{array} \right.$$

All iterations in $H_1^U(i'_1)$ are mapped to the iterations in IS' that are executed for $\mathtt{I}'_1 = i'_1$. Hence, as illustrated in the first picture of figure 3.6, the original iteration space is traversed along planes defined by $\mathtt{I}_1 + \mathtt{I}_2 + \mathtt{I}_3 = i'_1$ that are moved in the direction $(1,1,1)^T$ in successive iterations of the $\mathtt{I}'_1$-loop. Likewise, all iterations in $H_1^U(i'_1) \cap H_2^U(i'_2)$ are mapped to iterations executed for $\mathtt{I}'_1 = i'_1$ and $\mathtt{I}'_2 = i'_2$. As depicted in the second picture of figure 3.6, this intersection forms a straight line with direction $(0,1,-1)$ (cf. last column of U^{-1}). Finally, the single iteration in $H_1^U(i'_1) \cap H_2^U(i'_2) \cap H_3^U(i'_3)$, illustrated in the last picture of figure 3.6, is mapped to the iteration $\vec{\mathtt{I}}' = (i'_1, i'_2, i'_3)^T$.

On account of these observations, we identify two methods to construct a unimodular matrix that affects the order in which iterations are executed in some desired manner [25].

Outermost DO-loop of the Target Loop

Suppose that we want to map all iterations of $IS \subseteq \mathcal{Z}^d$ lying in a hyperplane defined by the equation $\vec{\alpha} \cdot \vec{\mathtt{I}} = i'_1$, where $\vec{\alpha} \in \mathcal{Z}^d$ and $\gcd(\alpha_1, \ldots, \alpha_d) = 1$, to iterations in $IS' \subseteq \mathcal{Z}^d$ that are executed for $\mathtt{I}'_1 = i'_1$. The desired transformation is defined by any unimodular matrix U having $(\alpha_1, \ldots, \alpha_d)$ as first row.

The extended completion method presented in section 2.2.2 can be used to construct such a matrix U and corresponding inverse.

Example: Inner loop concurrentization methods [18, 20, 130, 228] are based on this observation. Given a set $\mathcal{D} \subseteq \mathcal{Z}^d$ of distance vectors representing the data dependence structure of a loop, first a vector $\vec{\alpha} \in \mathcal{Z}^d$ of which the components are relative prime is determined such that for all $\vec{d} \in \mathcal{D}$, the inequality $\vec{\alpha} \cdot \vec{d} \geq 1$ holds. Thereafter, a unimodular matrix U having $\vec{\alpha} \in \mathcal{Z}^d$ as first row is constructed and the linear transformation defined by this matrix is applied. For instance, in the following double loop, the static data dependences $S_1\delta_{(1,0)}S_1$ and $S_1\delta_{(0,1)}S_1$ are represented by the set $\mathcal{D} = \{(1,0)^T, (0,1)^T\}$:

```
      DO I1 = 0, 4
        DO I2 = 0, 4
S1:       A(I1,I2) = A(I1-1,I2) + A(I1,I2-1)
        ENDDO
      ENDDO
```

For $\vec{\alpha} = (1,1)^T$, for example, the inequality $\vec{\alpha} \cdot \vec{d} = 1$ holds for all $\vec{d} \in \mathcal{D}$. This reflects independence of all iterations along straight lines defined by the equation $\mathtt{I}_1 + \mathtt{I}_2 = i'_1$, as illustrated in the first picture of figure 3.7. Application of the extended completion method to construct a unimodular matrix U with $(1,1)$ as first row yields the following matrices:

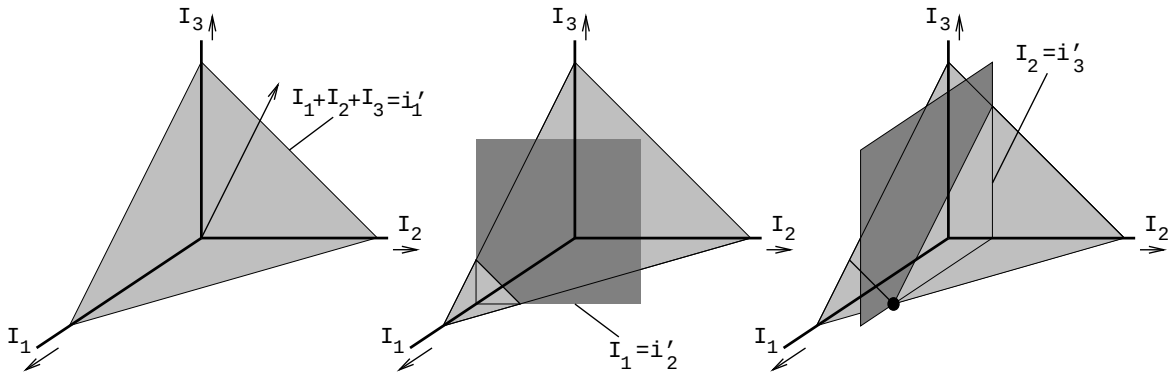


Figure 3.6: Traversal of Original Iteration Space

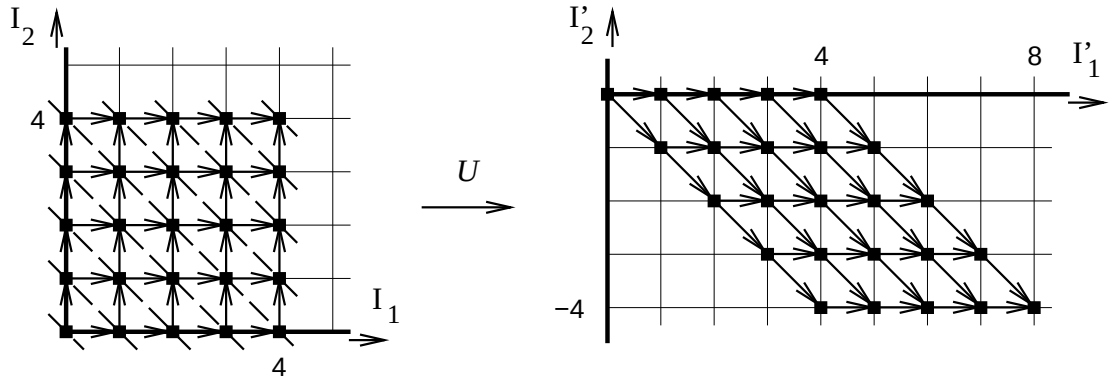


Figure 3.7: Hyperplane Traversal

$$U = \begin{pmatrix} 1 & 1 \\ -1 & 0 \end{pmatrix} \qquad U^{-1} = \begin{pmatrix} 0 & -1 \\ 1 & 1 \end{pmatrix}$$

By construction, application of the transformation defined by U is valid, and yields the following target loop, of which the iteration space is illustrated in the second picture of figure 3.7:

```
DO I'1 = 0, 8
  DO I'2 = MAX(-4, -I'1), MIN(0, 4-I'1)
    A(-I'2,I'1+I'2) = A(-I'2-1,I'1+I'2) + A(-I'2,I'1+I'2-1)
  ENDDO
ENDDO
```

Obviously, all iterations of the $\mathtt{I}'_2$-loop can be executed in parallel. We are not restricted to use this particular matrix, however. In fact, any unimodular matrix having $(1, 1)$ as first row can be used, which illustrates that in some cases, different loop transformations can be used to achieve the same objective.

Innermost DO-loop of the Target Loop

In other cases, we want to map all iterations in $IS \subseteq \mathcal{Z}^d$ that are along a single straight line with direction $\vec{\alpha} \in \mathcal{Z}^d$, where $\gcd(\alpha_1, \ldots, \alpha_d) = 1$, to iterations in $IS' \subseteq \mathcal{Z}^d$ in successive iterations of the innermost DO-loop of the target loop for fixed $\mathtt{I}'_1 = i'_1, \ldots, \mathtt{I}'_{d-1} = i'_{d-1}$. Obviously, the equation $\vec{\mathtt{I}} = U^{-1}\vec{\mathtt{I}}'$ implies that any transformation defined by a matrix U where the last column of U^{-1} consists of $\vec{\alpha} \in \mathcal{Z}^d$ can be used for this purpose.

There is no need to develop another completion method to construct such a matrix, because any unimodular matrix U with $\vec{\alpha} \in \mathcal{Z}^d$ as first row together with U^{-1} can be converted into the desired matrices $\tilde{U}$ and $\tilde{U}^{-1}$ as follows:

$$\tilde{U}^{-1} = (PU)^T \qquad \tilde{U} = (U^{-1}P^T)^T \qquad \text{where } P = \begin{pmatrix} & 1 & & \\ & & \ddots & \\ & & & 1 \\ 1 & & & \end{pmatrix}$$

Example: Given a set $\mathcal{D} \subseteq \mathcal{Z}^d$ of dependence distance vectors representing the data dependence structure of a loop, enforcing a traversal along straight lines with the direction $\vec{\alpha} \in \mathcal{Z}^d$ enables concurrentization of all DO-loops except the innermost DO-loop if for all $\vec{d} \in \mathcal{D}$, we have $\vec{d} = \lambda \cdot \vec{\alpha}$ holds for some $\lambda > 0$. For example, consider the following double loop:

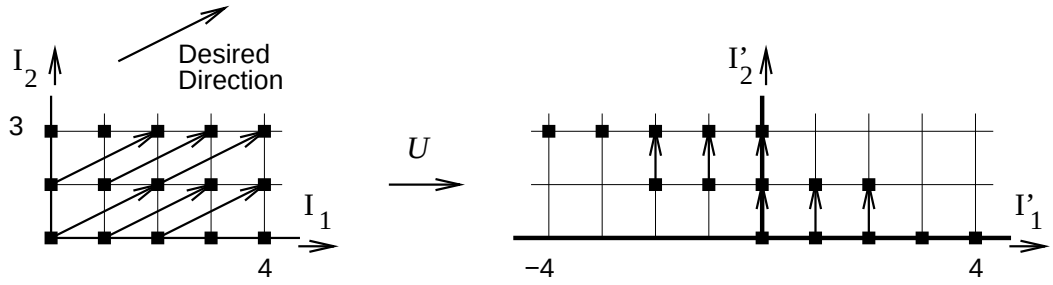


Figure 3.8: Enforcing a Direction

```
      DO I1 = 0, 4
        DO I2 = 0, 2
S1:       A(I1,I2) = A(I1-2,I2-1)
        ENDDO
      ENDDO
```

In this fragment, the static data dependence $S_1\delta_{(2,1)}S_1$ holds. Concurrentization of the outermost DO-loop becomes valid if we traverse the iteration space of this loop along straight lines with the direction $\vec{\alpha} = (2,1)^T$. Hence, a unimodular matrix U for which $(2,1)^T$ forms the last column of U^{-1} is constructed:

$$U = \begin{pmatrix} 1 & -2 \\ 0 & 1 \end{pmatrix} \qquad U^{-1} = \begin{pmatrix} 1 & 2 \\ 0 & 1 \end{pmatrix}$$

The target loop shown below results:

```
DO I'1 = -4, 4
  DO I'2 = MAX(0, ⌈-I'1/2⌉), MIN(2, ⌊(4-I'1)/2⌋)
    A(I'1+2*I'2,I'2) = A(I'1+2*I'2-2,I'2-1)
  ENDDO
ENDDO
```

In figure 3.8, the transformation of the original iteration space into the target iteration space is illustrated. This figure clearly shows that all dependences become parallel to the $\mathtt{I}'_2$-axis (viz. $U\vec{d} = (0,1)^T$), which implies that the $\mathtt{I}'_1$-loop can be converted into a DOALL-loop. Obviously, the restriction that all data dependences must be along the same direction is rather restrictive. More advanced outermost loop concurrentization methods are described in [19, 225].

3.3.5 Extensions to Unimodular Loop Transformations

It is relatively easy to deal with loop bounds in which symbolic constants appear, which are loop-invariant variables of which the actual values are unknown at compile-time. Given a perfectly nested loop with indices $\mathtt{I}_1, \ldots, \mathtt{I}_d$ and a number of symbolic constants $\mathtt{C}_1, \ldots, \mathtt{C}_m$ used in the loop bounds of this loop, we construct the following representation of loop bounds:

$$A(\mathtt{C}_1, \ldots, \mathtt{C}_m, \mathtt{I}_1, \ldots, \mathtt{I}_d)^T \leq \vec{b}$$

Since symbolic constants are unbounded in this system, conceptually a new perfectly nested loop having '`DO` $\mathtt{C}_k = -\infty, +\infty$' for $1 \leq k \leq m$ as outermost DO-loops results. Applying a loop transformation defined by a unimodular matrix U to the original loop is done using the following unimodular matrix $\hat{U}$, where the index vectors of the target and original loop are related according to $(\mathtt{C}_1, \ldots, \mathtt{C}_m, \mathtt{I}'_1, \ldots, \mathtt{I}'_d)^T = \hat{U}(\mathtt{C}_1, \ldots, \mathtt{C}_m, \mathtt{I}_1, \ldots, \mathtt{I}_d)^T$:

$$\hat{U} = \begin{pmatrix} I & \\ & U \end{pmatrix} \quad \hat{U}^{-1} = \begin{pmatrix} I & \\ & U^{-1} \end{pmatrix}$$

The loop-body of the target loop is still obtained by replacing all original loop indices according to $\vec{\mathtt{I}} = U^{-1}\vec{\mathtt{I}}'$. The new loop bounds are obtained by applying Fourier-Motzkin elimination to $A\tilde{U}^{-1} \leq \vec{b}$, where symbolic constants appear *before* loop indices (cf. [196]). During this elimination, unbounded variables may appear.

Example: Consider, for example, application of loop interchanging to the following double loop:

```
DO I1 = 1, C1+6                          DO I1' = -3, MIN(C2+6,100)
  DO I2 = I1-4,MIN(C2-C1+I1,100)           DO I2' = MAX(1,C1-C2+I1'),MIN(C1+6,I1'+4)
    B(I1,I2)                       →         B(I2',I1')
  ENDDO                                    ENDDO
ENDDO                                    ENDDO
```

For appropriate $\hat{U}$, applying Fourier-Motzkin elimination to $A\hat{U}^{-1}(\mathtt{C}_1, \mathtt{C}_2, \mathtt{I}'_1, \mathtt{I}'_2)^T \leq \vec{b}$ that represents the loop bounds of the target loop, results in a sequence of column augmented matrices that is initiated with the following two column augmented matrices from which the target loop bounds can be derived:

$$\left(\begin{array}{cccc|c} -1 & 0 & 0 & 1 & 6 \\ 0 & 0 & -1 & 1 & 4 \\ 1 & -1 & 1 & -1 & 0 \\ 0 & 0 & 0 & -1 & -1 \\ 0 & 0 & 1 & 0 & 100 \end{array}\right) \rightarrow \left(\begin{array}{ccc|c} 0 & -1 & 1 & 6 \\ 0 & 0 & 1 & 100 \\ 0 & 0 & -1 & 3 \\ 1 & -1 & 0 & 4 \\ -1 & 0 & 0 & 5 \end{array}\right) \rightarrow \ldots$$

The inequalities defined by the remaining column augmented matrices in this sequence define constraints under which execution sets are non-empty (e.g. $-5 \leq \mathtt{C}_1$ or $-9 \leq \mathtt{C}_2$):

$$\ldots \rightarrow \left(\begin{array}{cc|c} 0 & -1 & 9 \\ 1 & -1 & 4 \\ 0 & 0 & 103 \\ -1 & 0 & 5 \end{array}\right) \rightarrow \left(\begin{array}{c|c} -1 & 5 \\ 0 & 103 \end{array}\right) \rightarrow \begin{pmatrix} 103 \end{pmatrix}$$

A similar technique can be used to enable the application of a unimodular loop transformation to a perfectly nested *sub*-loop with admissible bounds appearing in a non-perfectly nested loop with possibly inadmissible bounds. All loop indices of DO-loops that are not directly involved in the loop transformation are also handled as symbolic constants. Note that in all these cases, an appropriate padding with zero components must be applied to the data dependence vectors to enable the validity test.

More advanced extensions can be found in the literature. For example, the framework can be extended to non-singular matrices [19, 22, 139, 140, 230], which allows for more freedom during construction of a suited loop transformation. However, some complications in code generation arise because not all points within the convex polytope defined by the image of the original bounds belong to the target iteration space, whereas the use of conditionals in the loop-body that exclude such points must be avoided. The framework can also be extended to deal with the transformation of non-perfectly nested loops [119]. Applying different loop transformations to different statements in a loop-body has been studied in [13, 14, 121]. In this case, it is useful to extend the framework to affine transformations (rather than just linear transformations) as well, to incorporate loop alignment [48].

3.4 Iteration Space Partitioning

In this section, we present a method to isolate a loop-body for all iterations within a polyhedral set defined by a system of linear inequalities [36]. This method differs from directly generating loop bounds from a given system of inequalities in the sense that loops iterating over the remaining iterations must also be generated. First, we present a simple loop transformation, referred to as **execution set partitioning**. Subsequently, we discuss how repetitive application of this basic transformation can be used to solve the problem.

Our iteration space partitioning method has as advantage that if the original iteration space consists of all discrete points within a bounded polyhedral set, then all generated loops have this property. In this manner, subsequent iteration space partitioning and other transformations relying on the representation of an iteration space as a system of linear inequalities remain feasible. Moreover, the method avoids redundant code duplication.

Iteration space partitioning can be used for a number of purposes. First, it can be used to simplify programs [50, 120]. For instance, after we have isolated the part of an iteration space in which a conditional is either true or false, a conditional statement that is controlled by that conditional can be eliminated. Although such simplifications are not likely to be applicable to ordinary programs, in many cases the code resulting after compiler transformations provides many opportunities for simplification. In other cases, isolating certain parts of the iteration space in which particular data dependences do not hold increases the opportunities for concurrentization and vectorization. Obviously, having a general method to isolate an iteration space in combination with an advanced data dependence analysis tool provides the compiler with sufficient functionality to enhance the performance of existing codes. Finally, the method can be very useful for a compiler that performs data structure transformations, such as the prototype sparse compiler presented in the second part of this dissertation.

3.4.1 Execution Set Partitioning

Execution set partitioning transforms a single DO-loop into two DO-loops, thereby partitioning the execution set into two disjoint sets.

Transformation

Given a stride-1 DO-loop with index `I` and an inequality $\texttt{I} \leq \texttt{T}$, application of execution set partitioning transforms this DO-loop into two new DO-loops with the same loop-body, such that all original iterations satisfying this inequality are executed in the first DO-loop, while all other iterations for which $\texttt{T} < \texttt{I}$ are executed in the second:

```
DO I = L, U                    DO I = L, MIN(U, T)
  B(I)                           B(I)
ENDDO                  →       ENDDO
                               DO I = MAX(L, T+1), U
                                 B(I)
                               ENDDO
```

Likewise, given the inequality $\texttt{T} \leq \texttt{I}$, we partition the execution set of the DO-loop as follows, so that all iterations satisfying the inequality are executed in the second loop, while all remaining iterations for which $\texttt{I} < \texttt{T}$ are executed in the first loop:

```
DO I = L, U                    DO I = L, MIN(U, T-1)
  B(I)                           B(I)
ENDDO                  →       ENDDO
                               DO I = MAX(L, T), U
                                 B(I)
                               ENDDO
```

Execution set partitioning resembles other loop transformations such as index set splitting, loop unrolling, and loop peeling [147, 166, 170, 228, 229, 234]. Obviously, the semantics of the original DO-loop are preserved because the use of minimum and maximum functions prevents the erroneous introduction of additional iterations. However, in many cases the efficiency of the generated code can be improved by eliminating redundant bounds. Furthermore, a zero-trip loop is eliminated and a one-trip loop is unrolled by replacing the complete DO-loop by the loop-body in which the appropriate value is substituted for I (recall that we always assume that the final value of a loop index is not used after a DO-loop).

Implementation of Execution Set Partitioning

Execution set partitioning can be implemented as follows. Suppose that we want to isolate the loop-body appearing at nesting depth d of an arbitrary loop for all iterations of the iteration space $IS \subseteq \mathcal{Z}^d$ in the following half-space, where all $a_i \in \mathcal{Z}$ and $b \in Z$:

$$H = \{\vec{\mathtt{I}} \in \mathcal{R}^d \mid a_1 \cdot \mathtt{I}_1 + \ldots + a_d \cdot \mathtt{I}_d \leq b\}$$

Let $1 \leq d' \leq d$ denote the index of the last nonzero coefficient in the equation (i.e. $a_{d'} \neq 0$ and $a_i = 0$ for $d' < i \leq d$).

Furthermore, let the system $A(\mathtt{I}_1, \ldots, \mathtt{I}_{d'})^T \leq \vec{b}$, where A is a $c \times d'$ integer matrix and $\vec{b} \in Z^c$, form a (conservative) representation of the iteration space of the loop consisting of the d' outermost DO-loops. We require that the $\mathtt{I}_{d'}$-loop is a stride-1 DO-loop with admissible bounds. However, if some inadmissible bounds appear in the more outer DO-loops, then the corresponding index is simply left unbounded. Furthermore, these DO-loops may have arbitrary strides, provided that the role of the lower and upper bound is interchanged if the stride is negative.

The sets $IS \cap H$ and $IS \cap \overline{H}$ restricted to the first d' loop indices are represented by the systems of linear inequalities of which the column augmented matrix representations are shown below (using the method (2.13) to negate an inequality for integer-valued variables):

$$M_1 = \left(\begin{array}{c|c} A & \vec{b} \\ a_1 \ldots a_{d'} & b \end{array} \right) \qquad M_2 = \left(\begin{array}{c|c} A & \vec{b} \\ -a_1 \ldots - a_{d'} & -b-1 \end{array} \right) \tag{3.9}$$

Transforming the $\mathtt{I}_{d'}$-loop into two DO-loops, gives rise to two new nested loops, sharing some outer DO-loops, where a copy of the loop-body of the original $\mathtt{I}_{d'}$-loop is used as loop-body:

```
DO I1 = L1, U1
  ...
    DO Id' = ..., ...
      ...                          ← first loop
    ENDDO
    DO Id' = ..., ...
      ...                          ← second loop
    ENDDO
  ...
ENDDO
```

If $a_{d'} > 0$, then the iteration space of the first and second loop are represented by M_1 and M_2 respectively. If $a_{d'} < 0$, then these iteration spaces are represented by M_2 and M_1 respectively In the former case, all iterations in $IS \cap H$ are executed by the first loop, whereas these iterations are executed by the second loop in the latter case.

Adding the bounds of indices of more outer DO-loops enables us to test the consistency of each system and to simplify the bounds of the resulting DO-loops. First, the consistency of both systems is tested using Fourier-Motzkin elimination. If a system M_i is inconsistent, the corresponding DO-loop is a zero-trip loop and does not have to be generated. Otherwise, the system may be simplified as follows.

While several non-examined upper bounds of index $\mathtt{I}_{d'}$ remain, the consistency of the system obtained by negating one of the upper bounds in M_i is tested using Fourier-Motzkin elimination. If the resulting system is inconsistent, the corresponding upper bound is eliminated (cf. proposition 2.1), while the original upper bound is restored otherwise. Lower bounds are simplified in a similar manner.

The bounds of the corresponding $\mathtt{I}_{d'}$-loop are generated according to the inequalities (2.11). If the lower bound is identical to the upper bound, the DO-loop may be unrolled.

Example: Suppose that we want to isolate the loop-body of the following double loop for all iterations in the half-plane defined by the linear inequality $-\mathtt{I} + \mathtt{J} \leq \mathtt{0}$:

```
DO I = 1, 100
  DO J = I, I+10
    B(I,J)
  ENDDO
ENDDO
```

The column augmented matrix representation of the iteration space is shown below:

$$(A \mid \vec{b}) = \left(\begin{array}{rr|r} -1 & 1 & 10 \\ 1 & -1 & 0 \\ 1 & 0 & 100 \\ -1 & 0 & -1 \end{array}\right) \begin{array}{l} \left.\begin{array}{l} \\ \\ \end{array}\right\} \text{bounds of the J-loop} \\ \left.\begin{array}{l} \\ \\ \end{array}\right\} \text{bounds of the I-loop} \end{array}$$

Adding the appropriate inequalities to this system and application of Fourier-Motzkin elimination to these systems reveals that both systems are consistent. Because two lower bounds are defined on J by the last system, consistency of this system where the second inequality is negated is tested (i.e. $\mathtt{I} \leq \mathtt{J}$ is replaced by $\mathtt{J} \leq \mathtt{I} - \mathtt{1}$). Since this system is inconsistent, this upper bound is eliminated.

Similar simplifications are applied to the first system:

$$\left(\begin{array}{rr|r} A & & \vec{b} \\ \hline -1 & 1 & 0 \end{array}\right) \qquad \left(\begin{array}{rr|r} A & & \vec{b} \\ \hline 1 & -1 & -1 \end{array}\right)$$

$$\downarrow \qquad\qquad \downarrow$$

$$\left(\begin{array}{rr|r} -1 & 1 & 0 \\ 1 & -1 & 0 \end{array}\right) \qquad \left(\begin{array}{rr|r} -1 & 1 & 10 \\ 1 & -1 & -1 \end{array}\right)$$

Consequently, the following code results, in which the first loop-body is isolated for iterations in the polyhedral set. Loop overhead is reduced by unrolling the first J-loop:

```
DO I = 1, 100                          DO I = 1, 100
  DO J = I, I                            B(I,I)
    B(I,J)                               DO J = I+1, I+10
  ENDDO                        →           B(I,J)
  DO J = I+1, I+10                       ENDDO
    B(I,J)                             ENDDO
  ENDDO
ENDDO
```

3.4.2 Partitioning an Iteration Space

Repetitive application of execution set partitioning can be used to isolate a loop-body of a nested loop with iteration space $IS \subseteq \mathcal{Z}^d$ for all iterations in a polyhedral set $PS \subseteq \mathcal{R}^d$ defined by $A\vec{\mathtt{I}} \leq \vec{b}$. For each inequality in the system, we partition the execution set of the appropriate DO-loop as explained in the previous section.

This gives rise to one of the following situations:

1. One loop is generated for which either
   (a) all iterations satisfy the current inequality, or
   (b) no iteration satisfies the current inequality.
2. Two loops are generated such that one loop executes all iterations satisfying the current inequality and the other loop executes the remaining iterations.

In case 1(b), the process can be terminated because none of the iterations satisfies all inequalities simultaneously, i.e. $IS \cap PS = \emptyset$. In the other cases, however, there is a loop in which the loop-body at nesting depth d is isolated for all iterations satisfying all previous considered inequalities. The process is applied recursively to this loop with the next inequality in the system until all inequalities have been considered.

Note that *considering inequalities that partition the execution set of outer DO-loops before inequalities partitioning the execution set of inner loops reduces the size of the generated code*. A preceding simplification of the system $A\vec{\mathtt{I}} \leq \vec{b}$ may reduce overhead in the resulting code.
Example: Consider the following nested loop:

```
DO I = 1, 10
  DO J = 1, 10
    B(I,J)
  ENDDO
ENDDO
```

If we want to isolate the loop-body for all iterations in the unbounded polyhedral set defined by $\mathtt{J} \leq 5$ and $-\mathtt{I} + \mathtt{J} \leq 0$, a naive approach would be to generate two J-loops with the execution sets $[1, \mathtt{MIN}(5, \mathtt{I})]$ and $[1 + \mathtt{MIN}(5, \mathtt{I}), 10]$. The disadvantage of this approach is that the iteration space of the second loop is not convex and, hence, cannot be represented by a system of inequalities.

However, using the iteration space partitioning method, we first partition the execution set of the J-loop according to the inequality $\mathtt{J} \leq 5$, yielding two DO-loops with the execution sets $[1, 5]$ and $[6, 10]$. Subsequently, we partition the execution set of the first DO-loop according to the half-space defined by $-\mathtt{I} + \mathtt{J} \leq 0$. These transformations eventually result in the following code, where the loop-body marked with a '$(*)$' is executed for the iterations in the polyhedral set:

```
DO I = 1, 10
  DO J = 1, MIN(5,I)
    B(I,J)                (*)
  ENDDO
  DO J = I+1, 5
    B(I,J)
  ENDDO
  DO J = 6, 10
    B(I,J)
  ENDDO
ENDDO
```

In this case, the iteration spaces of the loops executing the remaining iterations still consists of all discrete points in a convex polygon. This has been achieved by one additional partitioning of the remaining iteration space.
Example: Consider, as another example, the following triple loop:

```
DO I = 0, 50
  DO J = 0, 50
    DO K = 0, 50
      B(I,J,K)
    ENDDO
  ENDDO
ENDDO
```

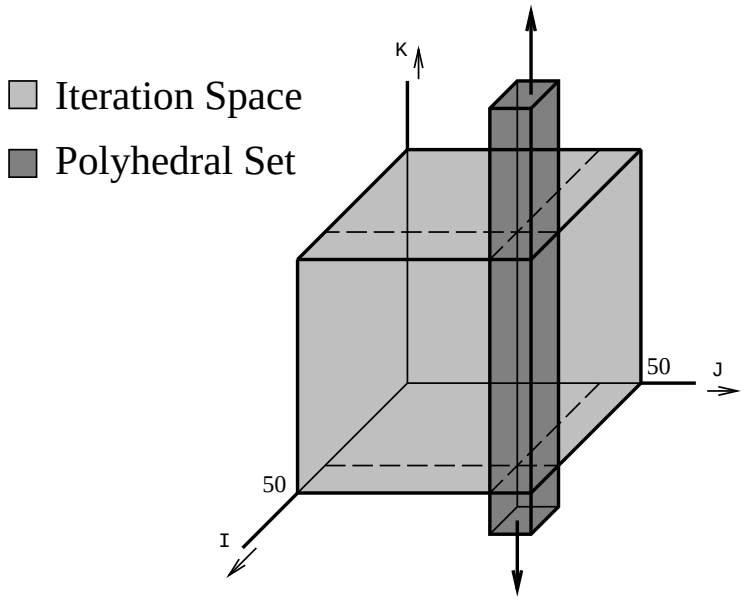


Figure 3.9: Desired Isolation

Now, suppose that we want to to isolate the loop-body of this loop for all iterations within the polyhedral set defined by the given inequalities:

$$\begin{array}{lcl} 40 & \leq & \texttt{I} \leq 50 \\ 40 & \leq & \texttt{J} \leq 50 \end{array}$$

In figure 3.9, the iteration space of this loop and the polyhedral set for which the loop-body must be isolated are shown. Obviously, starting with the inequalities that induce a partitioning of the execution set of the `I`-loop, which results in the generation of two DO-loops with the execution sets $[0, 39]$ and $[40, 50]$, followed by a similar partitioning of the execution set of the `J`-loop in the second loop results in the least increase of code size.

The code shown below results, where the loop-body marked with a '$(*)$' is executed for iterations in the polyhedral set:

```
DO I = 0, 39
  DO J = 0, 50
    DO K = 0, 50
      B(I,J,K)
    ENDDO
  ENDDO
ENDDO
DO I = 40, 50
  DO J = 0, 39
    DO K = 0, 50
      B(I,J,K)
    ENDDO
  ENDDO
  DO J = 40, 50
    DO K = 0, 50
      B(I,J,K)              (*)
    ENDDO
  ENDDO
ENDDO
```

If we would have partitioned the execution set of the `J`-loop first, the resulting `J`-loops with execution sets $[0, 39]$ and $[40, 50]$ would become unnecessary duplicated by application of execution set partitioning to the `I`-loop.

Example: Below, we give an example in which iteration space partitioning can be used to eliminate an IF-statement from the loop-body to reduce run-time overhead:

```
DO I = 2, 50                                    DO I = 2, 50
  DO J = 1, I-1                                   DO J = 1, I-1
    DO K = 1, 100                                   DO K = 1, ⌊(J+19)/2⌋
      IF ((2 * K - J) .EQ. 20) THEN                   B2(I,J,K)
        B1(I,J,K)                                   ENDDO
      ELSE                                  →       DO K = ⌈(J+20)/2⌉, ⌊(J+20)/2⌋
        B2(I,J,K)                                     B1(I,J,K)
      ENDIF                                         ENDDO
    ENDDO                                           DO K = ⌈(J+21)/2⌉, 100
  ENDDO                                               B2(I,J,K)
ENDDO                                               ENDDO
                                                  ENDDO
                                                ENDDO
```

Isolating the loop-body for all iterations lying within the hyperplane $-\mathtt{J} + 2 \cdot \mathtt{K} = 20$ can be done by slicing the iteration space according to the inequalities $20 \leq -\mathtt{J} + 2 \cdot \mathtt{K} \leq 20$.

Construction of the appropriate systems, followed by a test for consistency and elimination of redundant bounds eventually results in the following three systems:

$$\left(\begin{array}{ccc|c} \multicolumn{3}{c|}{A} & \vec{b} \\ \hline 0 & -1 & 2 & 19 \end{array}\right) \quad \left(\begin{array}{ccc|c} \multicolumn{3}{c|}{A} & \vec{b} \\ \hline 0 & -1 & 2 & 20 \\ 0 & 1 & -2 & -20 \end{array}\right) \quad \left(\begin{array}{ccc|c} \multicolumn{3}{c|}{A} & \vec{b} \\ \hline 0 & 1 & -2 & -21 \end{array}\right)$$

$$\downarrow \qquad\qquad \downarrow \qquad\qquad \downarrow$$

$$\left(\begin{array}{ccc|c} 0 & -1 & 2 & 19 \\ 0 & 0 & -1 & -1 \end{array}\right) \quad \left(\begin{array}{ccc|c} 0 & -1 & 2 & 20 \\ 0 & 1 & -2 & -20 \end{array}\right) \quad \left(\begin{array}{ccc|c} 0 & 0 & 1 & 100 \\ 0 & 1 & -2 & -21 \end{array}\right)$$

Although in the resulting fragment the lower bound of the second K-loop resembles the upper bound, unrolling is not allowed because this would erroneously introduce additional iterations in case the value of J is odd. The partitioned iteration space is shown in figure 3.10.

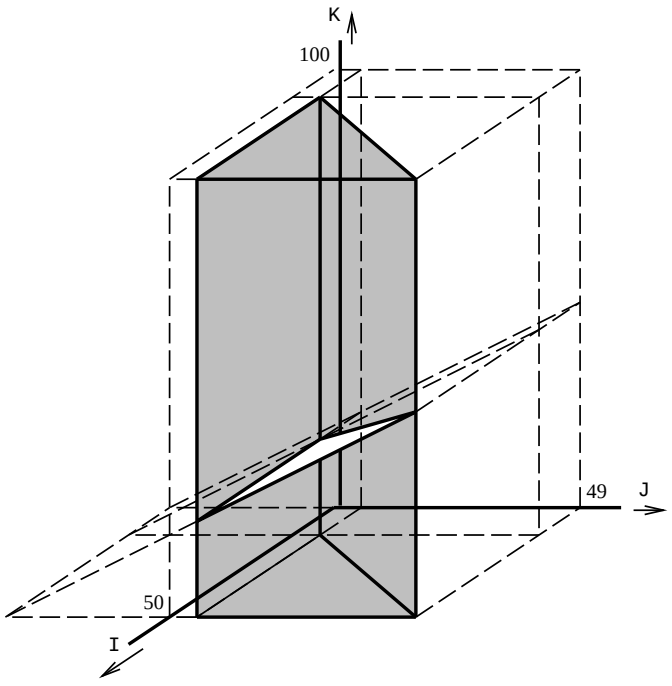


Figure 3.10: Partitioned Iteration Space

Part II

A Sparse Compiler

Chapter 4

A Sparse Compiler

A significant part of scientific codes consists of sparse matrix computations that are difficult to develop and that show notoriously bad efficiency on today's supercomputers (see e.g. [186]). In most cases, only a small fraction of the computing power of these computers can be utilized. Many reasons can be given for these effects. First, the codes usually suffer from a lack of spatial locality caused by irregular data accesses induced by sparse computations. This prohibits efficient cache utilization and reduces memory bandwidth. Another reason is that temporal locality is not as high as in most dense codes, because the amount of possible reuse of data is limited due to the elimination of many operations. Not only does this prevent data locality optimizations, but the communication overhead in message-passing architectures can be substantial. Finally, problems arise because sparse matrices need to be represented in a compact way to keep the storage requirements and computational time to reasonable levels. This causes the representation of a sparse code in either FORTRAN, with the occurrence of subscripted subscripts, or in another language with pointer structures, to be complex. This is probably the most important problem, because it complicates both the development and maintenance of sparse codes which are more complex than dense applications [73], and because it disables most compiler optimizations. In addition, because different architectural features or properties of the nonzero structures may favor different sparse storage schemes, a sparse program that has been developed for a particular target architecture or class of sparse matrices may perform poorly on another machine or for another class of sparse matrices.

To tackle the sparse data structure problem, examination of the following generic definition is useful: *sparse matrix computations are computations that compute on sparse data structures* and *sparse data structures are data structures that are logically contained in enveloping data structures*. The underlying problem for sparse matrix computations now is where to deal with the fact that only part of the enveloping data structure is computed on. The common approach is to deal with sparsity at the programming level. However, it is also possible to deal with this issue at a lower level, i.e. at the compilation level. This implies that all programming can be done as for dense computations, i.e. all sparse matrices are stored in simple two-dimensional arrays. Obviously, this greatly reduces the complexity of developing and maintaining the original dense program. Thereafter, a **sparse compiler** selects an appropriate sparse storage scheme for each matrix that is actually sparse and applies the corresponding data structure transformations to the original dense program. Hence, the output of the sparse compiler consists of semantically equivalent code operating on sparse data structures to take advantage of sparse matrices to reduce both storage requirements and computational time of the original dense program. The resulting sparse program is supplied to a conventional compiler for a particular target machine.

An advantage of this approach is that the sparse compiler does not need to extract program knowledge from an obscured code, but is presented with a much cleaner program on which regular data dependence checking and standard optimizations can be performed.

This frequently increases the amount of concurrency that can be detected and exploited automatically. In addition, because the sparse compiler performs the data structure selection and transformation, this selection can be based on the actual operations performed, possibly in combination with standard program transformations if the data structure selection cannot be resolved efficiently. Because the sparse compiler can account for both the characteristics of the target machine and the data operated on, one original dense program can be converted into several sparse versions that are specifically suited for a particular instance of the same problem, which implies that is it important to supply nonzero structure information to the sparse compiler. Finally, just as traditional restructuring compilers enable the re-use of existing serial software on parallel target architectures, a sparse compiler enables the re-use of parts of existing dense codes to develop sparse applications.

This approach has potential limitations though. The sparse compiler must rely on powerful strategies to prevent the generation of sparse codes with poor performance. Since much effort has already been put in the development of efficient sparse packages solving a particular problem, it will be extremely difficult to be competitive with such heavily specialized codes, even if all peculiarities of the sparse matrices could be supplied to the compiler and sophisticated reordering methods would be incorporated. In any case, a sparse compiler enables inexperienced programmers to generate reasonably efficient sparse code in a relatively simple way, whereas it can assist more experienced programmers to develop advanced sparse code, because the output of a sparse compiler can be further extended and hand-optimized.

These observations gave rise to the development and implementation of a prototype sparse compiler that is presented in this dissertation. In this chapter, we first discuss some issues related to sparse matrices. Subsequently, we give an overview of the organization of the prototype sparse compiler. Furthermore, we briefly discuss the data structure selection and transformation method used by this sparse compiler to automatically convert a dense program into semantically equivalent sparse code. The different phases of this method are discussed in more detail in subsequent chapters.

4.1 Sparse Matrices

In this section, we give definitions related to sparse matrices and identify some important nonzero structures. Next, an overview of sparse storage schemes is given. A brief overview of some issues related to direct methods to solve dense, symmetric, and sparse systems of linear equations can be found in appendix A.

4.1.1 Definitions

If many elements in a matrix are zero, then this matrix is called a **sparse matrix**. Usually, no attempts are made to obtain a more formal definition and we simply say that a matrix is sparse if it contains sufficient zero elements to enable the exploitation of these zero elements. Any other matrix is referred to as a **dense matrix**.

For an $m \times n$ sparse matrix A, the **nonzero structure** is defined as follows:

$$\mathit{Nonz}(A) = \{(i,j) \in [1,m] \times [1,n] \mid a_{ij} \neq 0\}$$

In figure 4.1, the nonzero structures of two sparse matrices taken from the Harwell-Boeing Sparse Matrix Collection [79] are illustrated. The 183×183 sparse matrix 'fs_183_1' with 1069 nonzero elements has a rather arbitrary nonzero structure. In contrast, the 1005 nonzero elements of the 185×185 sparse matrix 'gre_185' are clustered around the main diagonal.

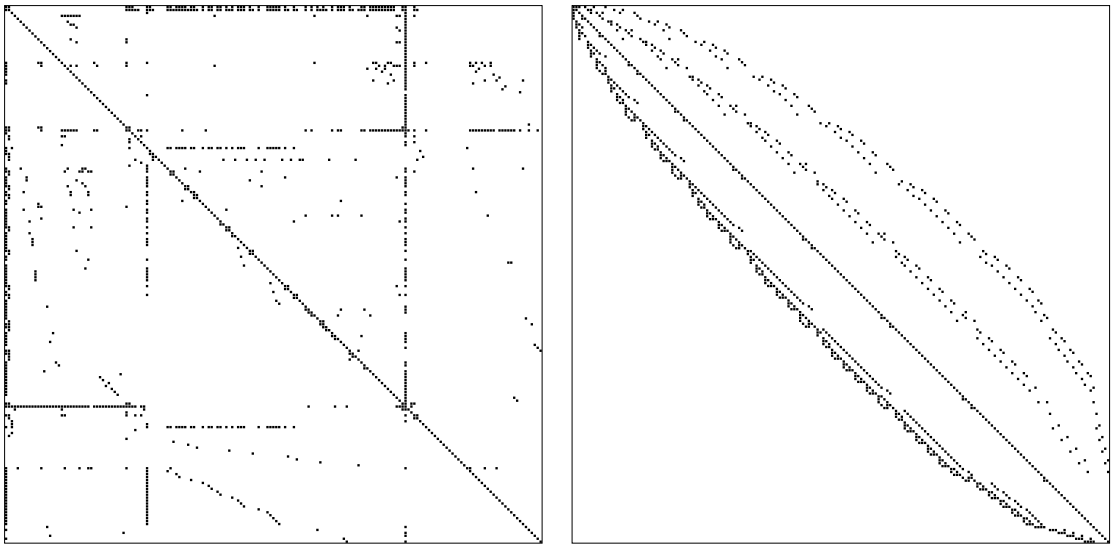


Figure 4.1: Nonzero Structure of 'fs_183_1' and 'gre_185'

The number τ of nonzero elements in A is defined as $\tau = |\mathit{Nonz}(A)|$. The **density** ρ of A is defined as follows (giving rise to a **sparsity** of $1 - \rho$):

$$\rho = \frac{\tau}{m \cdot n}$$

In many fields of science and engineering, applications arise that operate on sparse matrices. Both the *storage requirements* and *computational time* of these applications can be reduced substantially if advantage of the zero elements in these matrices is taken.

Reduction of Storage Requirements

Many sparse storage schemes have been developed to reduce the storage requirements of a sparse matrix. Which sparse storage scheme is the most efficient heavily depends on peculiarities of the nonzero structure of the sparse matrix and the kind of operations to be applied to this matrix.

Storage required to store numerical values is called **primary storage**. Storage necessary to reconstruct the underlying matrix is referred to as **overhead storage**. In some cases it is practical to store some zero elements too, because the use of a simpler storage scheme with less overhead storage compensates for the increase in the amount of primary storage and results in less run-time overhead.

Elements that are stored explicitly are called **entries**. The set $E(A)$ is used to indicate the index set of all entries of a sparse $m \times n$ matrix A:

$$\mathit{Nonz}(A) \subseteq E(A) \subseteq [1, m] \times [1, n]$$

Hence, $(i, j) \notin E(A)$ implies that $a_{ij} = 0$, but the converse implication does not necessarily hold. Moreover, if elements of the matrix may change during a computation, we must be ready to deal with situations in which zero elements become nonzero, which is referred to as **fill-in**. Usually, we ignore the opposite situation in which nonzero elements become zero. Depending on whether a fixed $E(A)$ can be chosen such that $\mathit{Nonz}(A) \subseteq E(A)$ will always hold during program execution, or whether unpredictable alterations to $E(A)$ must be possible at run-time, we distinguish between **static storage schemes** and **dynamic storage schemes**.

For static storage schemes, we can further distinguish between cases where the fixed set $E(A)$ is already known at compile-time, because all changes are confined to fixed regions known in advance, or where this fixed set $E(A)$ is determined at run-time before initialization of the storage scheme by computing a conservative approximation of elements that may fill-in. In a dynamic storage scheme, we can alter the set $E(A)$ at run-time to account for the insertion of a new entry, which is referred to as **creation**. If initially all zero elements in the matrix are exploited (viz. we start with $E(A) = \mathit{Nonz}(A)$), then all fill-in induces creation. This can contribute substantially to the computational time because data movement and occasionally a left compression may occur, as further explained in section 4.1.3.

Reduction of Computational Time

The actual computational time of an algorithm operating on a sparse matrix can be reduced if we account for the fact that certain operations on zero elements can be skipped. Usually, such a reduction of the actual computational time can only be achieved if an appropriate storage scheme is used, because, in general, skipping operations by means of conditionals does not yield a satisfactory reduction in computational time. Only if we can keep the work proportional to the number of nonzero elements in a matrix, sparsity has been fully exploited. Sparse storage schemes and related operations are further discussed in section 4.1.3.

4.1.2 Nonzero Structures

We can distinguish between general sparse matrices and sparse matrices that have a particular nonzero structure. In the following sections some important nonzero structures of *square* matrices are identified (see e.g. [78, 173, 198, 199, 214]). Note that a matrix in X-form may also be referred to as an X-matrix (e.g. a matrix in lower triangular form may also be called a lower triangular matrix).

Band Forms

The **lower semi-bandwidth** b_l and **upper semi-bandwidth** b_u of an $n \times n$ matrix A are defined as the smallest integers $b_l \geq 0$ and $b_u \geq 0$ for which the following constraint is still satisfied:

$$(a_{ij} \neq 0) \Rightarrow (-b_u \leq i - j \leq b_l) \tag{4.1}$$

Minimum values reveal the most information about the nonzero structure, because (4.1) is trivially satisfied for $b_l = n - 1$ and $b_u = n - 1$. Allowing for negative semi-bandwidths would enable the specification of an arbitrary band in which the main diagonal is not necessarily included. However, usually we assume that all matrices have a full transversal (i.e. all elements on the main diagonal are nonzero).

If the semi-bandwidths are relatively small, we say that the matrix is in **band form**, which means that all nonzero elements are confined to a small band. The value $b_l + b_u + 1$ is referred to as the **bandwidth**. Some special classes of band matrices can be distinguished. A band matrix is in **diagonal form** if both b_l and b_u are zero, and in **tridiagonal form** if $b_l = b_u = 1$. A band matrix A is in **full band form** if the following constraint is satisfied:

$$(-b_u \leq i - j \leq b_l) \Leftrightarrow (a_{ij} \neq 0)$$

The **lower skyline** l_i and **upper skyline** u_i of an $n \times n$ matrix A with a full transversal are defined as the following two sequences for $1 \leq i \leq n$ and $1 \leq j \leq n$:

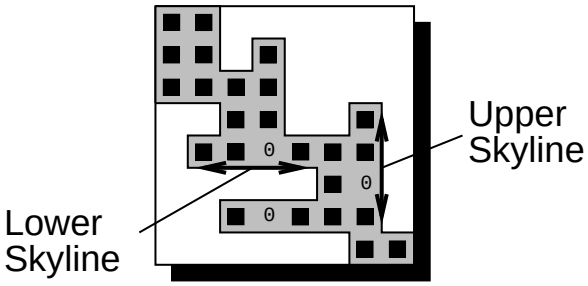


Figure 4.2: Variable Band Matrix

$$\begin{cases} l_i & = & i & - & \min\{j \mid a_{ij} \neq 0\} \\ u_j & = & j & - & \min\{i \mid a_{ij} \neq 0\} \end{cases} \tag{4.2}$$

Each l_i and u_j indicates the lower and upper semi-bandwidth in the ith row and jth column respectively:

$$(a_{ij} \neq 0) \Rightarrow (-u_j \leq i - j \leq l_i)$$

The **variable band form** of a matrix is defined by the lower and upper skyline. For instance, in figure 4.2 an 8×8 variable band matrix is shown. Although some zero elements still appear within the variable band, the nonzero structure is described more accurately by a variable band than by a band with fixed semi-bandwidths.

For a symmetric matrix A, i.e. a matrix that satisfies $A = A^T$, the lower and upper skyline are identical. The **envelope** of a symmetric matrix A consists of all elements in the variable band that are below the main diagonal. The **envelope size** or **profile** p of A is defined as follows [52, 97, 169]:

$$p = \sum_{i=1}^{n} l_i$$

Triangular Forms

A matrix satisfying the following constraint is in **lower triangular form**:

$$(a_{ij} \neq 0) \Rightarrow (j \leq i)$$

If additionally, the equation $a_{ii} = 1$ holds for all $1 \leq i \leq n$, then the matrix is in **unit lower triangular form**. If the inequality is strict (viz. $j < i$, which implies that the transversal is empty), then the matrix is in **strictly lower triangular form**. A lower triangular matrix is, in fact, a special band matrix with $b_u = 0$ and relatively large $b_l > 0$. For a relatively small $b_l > 0$ the matrix is in so-called **band lower triangular form**. Similar definitions can be given for matrices in **(unit/strictly) upper triangular form** and **band upper triangular form**.

Block Forms

Consider a block partition of a square matrix A into sub-matrices A_{ij}:

$$A = \begin{pmatrix} A_{11} & \dots & A_{1p} \\ \vdots & \ddots & \\ A_{p1} & & A_{pp} \end{pmatrix}$$

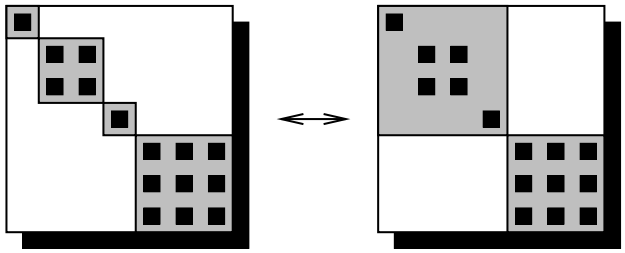


Figure 4.3: Two Different Block Partitions into Block Diagonal Form

Each sub-matrix A_{ii}, referred to as a **diagonal block**, is a square $n_i \times n_i$ sub-matrix. Hence, each sub-matrix A_{ij} with $i \neq j$, referred to as an **off-diagonal block**, is necessarily an $n_i \times n_j$ sub-matrix. If a block consists of zero elements only, this is denoted by $A_{ij} = 0$. Such blocks are referred to as **zero blocks**.

Given this block partition, a **block band form** is defined by the block lower and upper semi-bandwidths $B_l \geq 0$ and $B_u \geq 0$, which are the *minimum* values for which the following constraint is still satisfied:

$$(A_{ij} \neq 0) \Rightarrow (-B_u \leq i - j \leq B_l)$$

If $B_l = B_u = 1$, then the matrix is in **block tridiagonal form** and we have a **block diagonal form** if $B_l = B_u = 0$. For $B_u = 0$ and a relatively large B_l, the matrix is in **block lower triangular form**. Likewise, for $B_l = 0$ and a relatively large B_u, the matrix is in **block upper triangular form**. The off-diagonal blocks A_{pi} and A_{ip} for $1 \leq i < p$ are referred to as the **lower border** and **upper border** respectively. If, except for some nonzero blocks in the lower or upper border, a matrix is in block diagonal form, then the matrix is in **doubly bordered block diagonal form**. Likewise, there are matrices in **singly bordered block lower triangular form** or **singly bordered block upper triangular form**.

Although, depending on which blocks are nonzero, a particular block form of a matrix is defined once a block partition of that matrix is given, it is possible that *similar* block forms defined by *different* block partitions differ in the accuracy of describing the nonzero structure (viz. a matrix is in any block form using the trivial block partition $A = A_{11}$). In figure 4.3, for example, two different block partitions of a matrix into block diagonal form are shown with respectively 15 and 25 elements in the nonzero blocks. Therefore, we say the most accurate description for a particular block form is given by the block form defined by a *minimum block partition into that block form*, which means that there are no other block partitions of the matrix into the same block form with fewer elements in the nonzero blocks (although the number of elements in the nonzero block is still likely to exceed the actual number of nonzero elements, since the nonzero blocks are not necessarily full).

We state the following obvious properties about block diagonal forms (similar propositions hold for block lower or upper triangular forms).

Proposition 4.1 *A square matrix has a unique minimum block partition into block diagonal form.*

PROOF Assume that a matrix has two *different* minimum block partitions into block diagonal form. This implies that at least two diagonal blocks of these different block partitions partially overlap or one is properly contained in the other. Hence, there must be a non-trivial block partition into block diagonal form of at least one of these diagonal blocks. This gives rise to a block partition into block diagonal form of the whole matrix with fewer elements, contradicting the assumption. □

Proposition 4.2 *A block partition of a square matrix into block diagonal form is minimum if and only if there is no diagonal block with a non-trivial block partition into block diagonal form.*

PROOF '$\Rightarrow$' A non-trivial block partition of a diagonal block into block diagonal form would contradict the minimality of the block partition. '$\Leftarrow$' Consider an arbitrary block partition into block diagonal form where no diagonal block can be further partitioned into block diagonal form. An arbitrary diagonal block of this block partition cannot be properly contained in an overlapping diagonal block of the minimum block partition into block diagonal form (since this would contradict minimality), nor can it partially overlap with a diagonal block of that block partition (since this would give rise to a further non-trivial block partition of at least one of these diagonal blocks into block diagonal form). Hence, the two block partitions are equal. □

4.1.3 Sparse Storage Schemes

In this section, we present some storage schemes for sparse matrices. The overview is by no means exhaustive, because many other sparse storage schemes exist and, in addition, there are many variants of the presented storage schemes.

General Discussion

Because most numerical applications are written in FORTRAN, many storage schemes are based on arrays rather than on more advanced data structures using records/structures, pointers and dynamic memory allocation. One way to group logically related information together in the absence of such features is to use **parallel one-dimensional arrays**. Given a number of parallel arrays `A1`, `A2`, `A3`, and so on, all data stored at the `I`th location, i.e. the elements `A1(I)`, `A2(I)`, `A3(I)`, etc., are related. In this manner, a linked list of at most 7 values of type `REAL`, for example, can be implemented as follows (cf. [78, p25-28] and [169, p8-10]):

```
REAL    VAL(7)
INTEGER LNK(7), HD, FREE
```

The first element of the list can be found in the parallel arrays `VAL` and `LNK` at the location indicated by `HD`. The value of this element is stored in `VAL(HD)`, while the next element can be found at location `LNK(HD)`. We can follow the links stored in `LNK` until a null pointer is encountered, for which usually the value zero or a negative value is used. Likewise, all elements of the parallel arrays which are not used are linked together in a free-list. The first element of this list can be found through `FREE`. For example, possible contents of these arrays for a linked list containing $(3.0, 8.0, 12.0)$ are illustrated below for `HD=2` and `FREE=1`:

	1	2	3	4	5	6	7
VAL	-	3.0	-	-	12.0	8.0	-
LNK	4	6	7	3	0	5	0

In the following sections, we assume that the constant `N` contains the order of the matrix to be stored. For dynamic data structures, we assume that the value of a constant `MAXSZ` is at least the maximum number of entries that can appear in this matrix. Furthermore, we assume that the value of each entry in this matrix is of type `REAL`, although other types may be used in case double precision or complex numbers must be stored. Moreover, the type `INTEGER` is used for all integers, although usually less bytes are required to store a row or column index of a matrix than the storage required for an integer that is used as a pointer. The value of the former cannot exceed the order of the matrix, whereas the value of the latter may be as large as the number of entries.

Band and Diagonal Schemes

In a band scheme [78, p200-203][97, p48-51][169, 185, p13-14], all elements in the band of a band matrix with the semi-bandwidths b_l and b_u are stored in a rectangular array declared as either '`REAL BND1(N,BW)`' or '`REAL BND2(BW,N)`', where `BW`$= b_l + b_u + 1$. Which of these declarations is used and the way in which entries are stored in this array both depend on whether consecutive storage of the elements along rows, columns or diagonals is desirable.
Example: Consider the following 6×6 band matrix A having $b_l = 2$ and $b_u = 1$:

$$A = \begin{pmatrix} a_{11} & a_{12} & & & & \\ a_{21} & a_{22} & a_{23} & & & \\ a_{31} & a_{32} & a_{33} & a_{34} & & \\ & a_{42} & a_{43} & a_{44} & a_{45} & \\ & & a_{53} & a_{54} & a_{55} & a_{56} \\ & & & a_{64} & a_{65} & a_{66} \end{pmatrix}$$

Two ways of storing the elements within the band of this matrix are illustrated below:

`BND1`

$\perp$	$\perp$	a_{11}	a_{12}
$\perp$	a_{21}	a_{22}	a_{23}
a_{31}	a_{32}	a_{33}	a_{34}
a_{42}	a_{43}	a_{44}	a_{45}
a_{53}	a_{54}	a_{55}	a_{56}
a_{64}	a_{65}	a_{66}	$\perp$

`BND2`

$\perp$	$\perp$	a_{31}	a_{42}	a_{53}	a_{64}
$\perp$	a_{21}	a_{32}	a_{43}	a_{54}	a_{65}
a_{11}	a_{22}	a_{33}	a_{44}	a_{55}	a_{66}
a_{12}	a_{23}	a_{34}	a_{45}	a_{56}	$\perp$

For column-major storage, elements along one diagonal are stored consecutively in the first rectangular array. The rows of A can be accessed along the rows of `BND1` because diagonals in the lower triangular part of A are down-justified, whereas all diagonals above the main diagonal are up-justified in the array. Likewise, rows of A can be accessed along the columns of `BND2`, whereas diagonals of A are stored along the rows of this array. However, other layouts are also possible.

Only the zero elements outside the band in the matrix are exploited using a band scheme, because all elements within the band are stored explicitly (for symmetric band matrices only the elements in the lower or upper triangular part of the band have to be stored):

$$E(A) = \{(i,j) \in [1,m] \times [1,n] \mid -b_u \leq i - j \leq b_l\} \supseteq \mathit{Nonz}(A)$$

However, an advantage of this storage scheme is that, during LU-factorization without pivoting (see appendix A), all fill-in is confined to the band. Hence, the band scheme can be used as *static* data structure in which creation does not have to be accounted for. The relative simplicity of band schemes and the code operating on this data structure together with the high performance that can be achieved on pipelined vector processors have made band methods rather popular.

A slightly more complex variant of a band scheme that allows for storing a few nonzero diagonals is formed by a diagonal scheme [185][129, ch11], where for each nonzero diagonal an offset to the main diagonal is recorded in a one-dimensional array `OFF`, while the diagonals are stored along the columns of a two-dimensional array `VAL`.
Example: Consider the following 4×4 matrix A:

$$A = \begin{pmatrix} a_{11} & a_{12} & & \\ & a_{22} & a_{23} & \\ a_{31} & & a_{33} & a_{34} \\ & a_{42} & & a_{44} \end{pmatrix}$$

A diagonal scheme for this matrix is illustrated below:

$$\texttt{OFF}\ \boxed{\begin{array}{ccc} 2 & 0 & -1 \end{array}} \qquad \texttt{VAL}\ \boxed{\begin{array}{ccc} \perp & a_{11} & a_{12} \\ \perp & a_{22} & a_{23} \\ a_{31} & a_{33} & a_{34} \\ a_{42} & a_{44} & \perp \end{array}}$$

Envelope Schemes

Alternative sparse storage schemes for *symmetric* band matrices that preserve most of the simplicity of band schemes but at the same time offer more potential to exploit sparsity, are formed by envelope schemes [112, 113]. These schemes are based on storage of all elements in the lower triangular part that are within a *variable band*, i.e. $E(A)$ is defined as follows:

$$E(A) = \{(i,j) \in [1,m] \times [1,n] \mid -u_j \leq i - j \leq l_i\} \supseteq \mathit{Nonz}(A)$$

All elements in a row from the first nonzero element up to the diagonal element are stored consecutively in a one-dimensional array, declared as `REAL VAL(MAXSZ)` (the main sequence), in which the different row segments are stored contiguously. An additional one-dimensional integer array, declared as '`INTEGER PTR(N)`' (the address sequence), is used to locate the diagonal elements.

Example: Consider the following lower triangular part of a symmetric and sparse 5×5 matrix A, where only the nonzero elements are shown:

$$A = \begin{pmatrix} a_{11} & & & & \\ a_{21} & a_{22} & & & \\ a_{31} & & a_{33} & & \\ & & a_{43} & a_{44} & \\ & a_{52} & & a_{54} & a_{55} \end{pmatrix}$$

The corresponding envelope storage scheme is illustrated in figure 4.4, where each `PTR(I)` contains the location of the `I`th diagonal element in the main sequence. An element in the variable band with row index `I` and column index `J` is stored at location `PTR(I)-I+J` in the main sequence. Conversely, the *column index* of the first entry in row `I` can be determined as follows:

```
I - (PTR(I) - PTR(I-1) - 1)
```

Many variants of this storage scheme exist (see e.g [78, p151-153,p204-205][97, p79-80][146] [169, p14-16][185]). The upper triangular part of the matrix can be stored by columns, which corresponds to storing A^T according to the previous method. Separate storage can be used for the main diagonal. An advantage that is shared by all versions is that, because during LU-factorization without pivoting, fill-in is confined to the variable band, an envelope scheme can be used as *static* data structure. In fact, the envelope even becomes completely full if a nonzero element appears before each diagonal element after the first row [93].

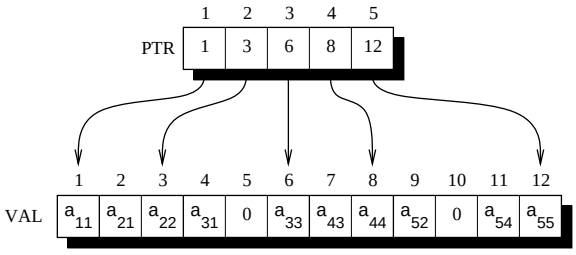


Figure 4.4: Envelope Scheme

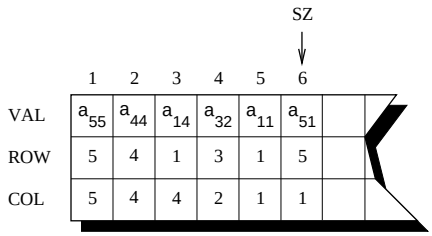


Figure 4.5: Coordinate Scheme

Coordinate Schemes

The most convenient way to store a general sparse matrix is using a coordinate scheme, in which all entries are stored as an unordered set of triples (a_{ij}, i, j) in three parallel arrays [78, p23-24][129, 185, 219][235, ch2].
Example: Consider the following sparse 5×5 matrix A:

$$A = \begin{pmatrix} a_{11} & & a_{14} & & \\ & a_{32} & & & \\ & & a_{44} & & \\ a_{51} & & & & a_{55} \end{pmatrix} \tag{4.3}$$

The six nonzero elements of this matrix are stored in arbitrary order in the first six elements of the parallel arrays `VAL`, `ROW`, and `COL` of size `MAXSZ`, as illustrated in figure 4.5. A scalar `SZ` is used to record the number of explicitly stored elements. A new entry can be easily inserted at the first free location. A given entry can be easily deleted by moving the last stored entry to the location of the deleted entry. However, in order to search for a particular entry or to fetch an entire row or column, all entries must be scanned, making this storage scheme less convenient for most numerical applications. Due to its simplicity, coordinate schemes are used as input scheme by several applications [74, 80, 94, 164]. In this way, little constraints are imposed on the input sets. The coordinate scheme is transformed into an efficient storage scheme before the actual computations are performed.

Linked List Schemes

A linked list scheme [122, p298-302] provides efficient access from each entry to the next entry in its row as well as to the next entry in its column. Furthermore, pointers to the first element in each list are stored. In figure 4.6 this idea is illustrated for the matrix (4.3). A possible implementation of the linked list scheme [169, p16-20], illustrated in the same figure, is shown below, where `FREE` can be used as a pointer to the first location of a free-list:

```
REAL    VAL(MAXSZ)
INTEGER ROW(MAXSZ), COL(MAXSZ), LNKR(MAXSZ), LNKC(MAXSZ)
INTEGER HDR(N), HDC(N), FREE
```

For each entry, the value, row and column index together with links to the next entry in the same row and column are stored in five parallel arrays. Pointers to the location of the first entry in each row and column can be found through the elements of arrays `HDR` and `HDC`, respectively. For example, because in figure 4.6 we have `HDR(5)=2`, `LNKR(2)=4`, and `LNKR(4)=0`, the entries in the 5th row can be found at locations 2 and 4 of the parallel arrays. Obviously, because four integers are associated with each entry, this storage scheme suffers from substantial overhead storage.

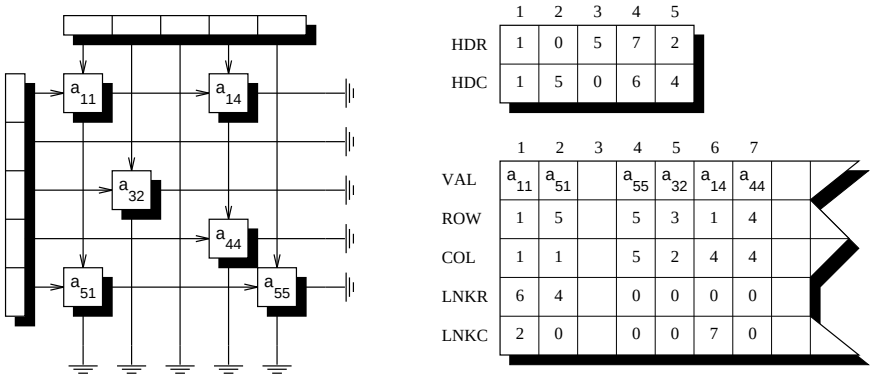


Figure 4.6: Linked List Scheme

Some savings are obtained by dropping the row and column index associated with each entry and replacing the null pointer at the end of each row and column list by the negation of these indices [56][78, 77, p31-32][235, p34-36]. This so-called Curtis and Reid scheme is illustrated in figure 4.7. The row or column index of each entry is obtained by scanning to the end of the row or column list. For sparse matrices having a small number of entries in each row and column, the storage savings are obtained at the expense of only a relatively small increase in computational time. Alternatively, storage can be saved if only the entries in a row or a column are linked together, yielding a row-linked or column-linked list [78, p28-29][185][199, ch1].

The use of linked list scheme has as advantage that creation can be implemented without any data movement, while only a few links are affected. Accessing the links, however, may contribute substantially to the computational time, while locality may be disturbed in case the elements in a linked list are scattered through the parallel arrays.

General Sparse Row- or Column-wise Schemes

Another sparse storage scheme for general sparse matrices is based on storing either the rows or columns as a set of sparse vectors.

Example: Consider the 5×5 sparse matrix given below:

$$A = \begin{pmatrix} a_{11} & & a_{13} & a_{14} & \\ & a_{22} & & & a_{25} \\ a_{31} & & a_{33} & a_{34} & \\ a_{41} & & & a_{44} & \\ & & a_{53} & & a_{55} \end{pmatrix} \tag{4.4}$$

Sparse row-wise storage of A is illustrated in figure 4.8. The value of all entries in a row together with the corresponding column indices are stored consecutively in the parallel arrays VAL and IND, where entries in one row are not necessarily sorted on column index.

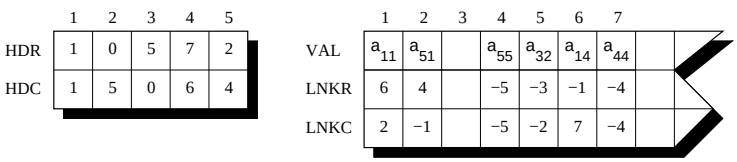


Figure 4.7: Curtis and Reid Scheme

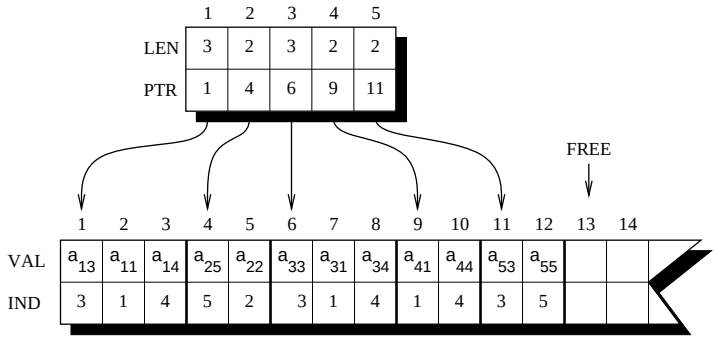


Figure 4.8: Sparse Row-wise Scheme

The location of the first entry in a row `I` can be found through `PTR(I)`, while the number of entries in this row is defined by `LEN(I)`. A scalar `FREE` contains the first unused location at the end of all rows:

```
REAL    VAL(MAXSZ)
INTEGER IND(MAXSZ), PTR(N), LEN(N), FREE
```

Inserting an element requires some data movement if there is no free space adjacent to the corresponding row. In this case, all entries of the row are moved to free space at the end of all rows, after which the new element is added. For example, in figure 4.9 we show the data structure of figure 4.8 after element a_{23} has been inserted in the second row. The previously occupied locations are marked as free by resetting the associated indices. Free space can be used by subsequent insertions. In figure 4.9, for instance, a new element can be inserted in row 1 or row 3 without any data movement. If, however, data movement is required but cannot be done because insufficient free space is available at the end of all rows, a left-compression is performed to make all rows contiguous again [74, 80][164, p25-33][235, p16-25]. Since such a left-compression is relatively expensive, sufficient working space (or 'elbow room') must be supplied to prevent the situation in which a left-compression has to be applied many times.

There are different kinds of sparse row- and column-wise storage schemes (see e.g. [2][78, p24-25,p31-32][77, 80][129, ch11][105, 185][199, ch1][164][235, ch2]). For example, in *ordered* variants, the entries are sorted on index information, making creation slightly more expensive. An additional pointer can be used to separate entries in the lower and triangular part, whereas the main diagonal can be kept in separate storage. Because sparse row- or column-wise storage only supports fast generation of entries along a row or column, the column- or row-structure of the matrix may be stored as well.

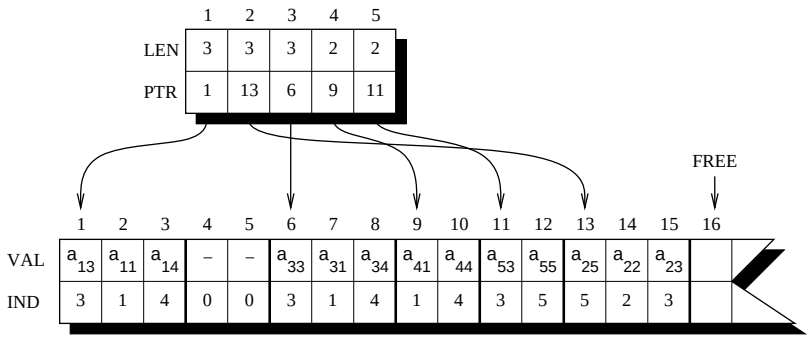


Figure 4.9: Data Movement

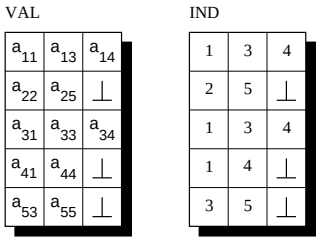


Figure 4.10: Extended Column Scheme

A representation of a sparse matrix in either a linked list scheme or the general sparse scheme of this section can be efficiently converted into a similar representation of the *transposed* matrix [106][169, p236-239]. In fact, since entries are ordered on index information afterwards, applying such an algorithm twice can be used to converted an unordered representation of a sparse matrix into an ordered representation [8, 106][169, p239-240].

Extended Column or Row Schemes

In the extended column or ITPACK scheme [162], every kth entry in the ith row of a matrix is stored in the ith row and kth column of a two-dimensional array '`REAL VAL(N,MAXROW)`', while the column index of this entry is stored at the same position in a two-dimensional integer array `IND` with the same shape. Here, `MAXROW` denotes the maximum number of entries in a row of the matrix. In figure 4.10, we illustrate the extended column scheme for the matrix (4.4), where an appropriate padding must be used for all rows with less than `MAXROW` entries (denoted by '$\perp$').

Again, many variants of this scheme are possible [2, 11, 84, 184, 185, 186, 219][235, p39-40]. The entries in each row may be unordered, while column-oriented schemes are also possible. Reordering the rows in the matrix in decreasing order of the number of entries per row can be used to move all unused locations in the arrays to the lower right corner. In this manner, redundant operations can be avoided by recording the number of entries stored in each column of `VAL` as well. A one-dimensional variant of this scheme is usually called a jagged diagonal scheme. All these storage schemes have been specifically developed to enhance vector performance on pipelined vector processors by accessing the entries along columns of `VAL`, which increases the average length of vector instructions.

Quad-Tree Schemes

As advocated in [1, 220, 221, 222, 223], so-called quad-trees, well-known from the fields of image processing and computer graphics (see e.g. [110, ch10][191]) can be used to represent sparse matrices. An $n \times n$ matrix is embedded in a $2^{\lceil \ln n \rceil} \times 2^{\lceil \ln n \rceil}$ matrix, where an appropriate padding with zero elements is applied. A zero matrix is represented by a NULL-pointer, whereas a 1×1 nonzero matrix is simply represented by a scalar. All other matrices are represented as a quadruple of sub-matrices consisting of the left-upper-, right-upper-, left-lower-, and right-lower-quadrant.

An example of a quad-tree representation of a 4×4 sparse matrix is shown in in figure 4.11. The quad-tree representation provides a uniform way of representing both dense as well as sparse matrices, while it also simplifies the implementation of algorithms based on matrix partitioning. For example, the sum of two matrices can be assembled recursively, where the recursion terminates if either one of the operands is the nil pointer (yielding the other operand as result), or if two scalars are encountered (yielding the sum of the these scalars). Likewise, matrix multiplication can be formulated recursively using eight recursive multiplications of quadrants followed by four additions, where the nil pointer acts as multiplicative cancellator and additive identity.

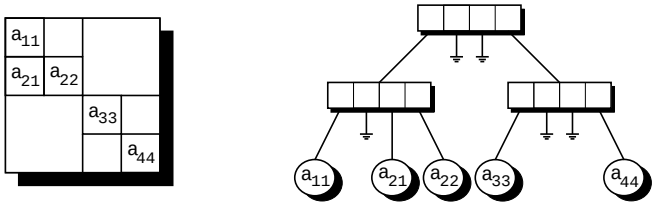


Figure 4.11: Quad-Tree Representation

4.2 Organization of the Sparse Compiler

In this section, a brief overview of sparse code generation is given and some sparse methods are reviewed to determine the issues that should be dealt with by a sparse compiler.

4.2.1 Terminology of the Sparse Compiler

For each matrix, the sparsity of which is not explicitly dealt with in the original dense program, the following three concepts can be distinguished:

- An $m \times n$ **implicitly sparse matrix** A, used at a logical level.
- An array `REAL A(M,N)`, used as **enveloping data structure** of A.
- A **sparse storage scheme** for A, selected by the sparse compiler.

The concept of an implicitly sparse matrix is introduced to reason about programs at a logical level (perform the operation $\vec{b} = A\vec{x}$, consider the nonzero structure of A, etc.). At the programming level, all operations on an implicitly sparse matrix are defined on the enveloping data structure, for which a two-dimensional array of appropriate size is used. Hence, an implicitly sparse matrix is, in fact, an ordinary sparse matrix for which a simple storage scheme is used to reduce the complexity of developing and maintaining the original dense program. The burden of sparse code generation is placed on the sparse compiler, which selects a suitable sparse storage scheme for each implicitly sparse matrix and transforms all occurrences of the corresponding enveloping data structure in the original dense program accordingly.

Hence, eventually semantically equivalent sparse code is generated in which the sparsity of each implicitly sparse matrix is accounted for to reduce the storage requirements and computational time of the original dense program. To emphasize the correspondence between an implicitly sparse matrix and its enveloping data structure, identical names will be used for both (i.e. two-dimensional arrays `A`, `B`, `C`, etc., denote the enveloping data structures of implicitly sparse matrices A, B, C, etc.).

4.2.2 The Sparse Compiler

As illustrated in figure 4.12, the input of the sparse compiler consists of an ordinary FORTRAN program stored in, for instance, the file '`prg.f`'. In this program, two-dimensional arrays are used as enveloping data structures of all implicitly sparse matrices. The use of arrays simplifies both the development and maintenance of the code. In addition, regular data dependence checking and standard restructuring techniques can be applied to the original dense program.

Information that cannot be expressed in the dense description of an algorithm is supplied to the sparse compiler by means of **annotations**. There are, for instance, annotations to identify the enveloping data structures or to incorporate techniques that are specific for sparse applications.

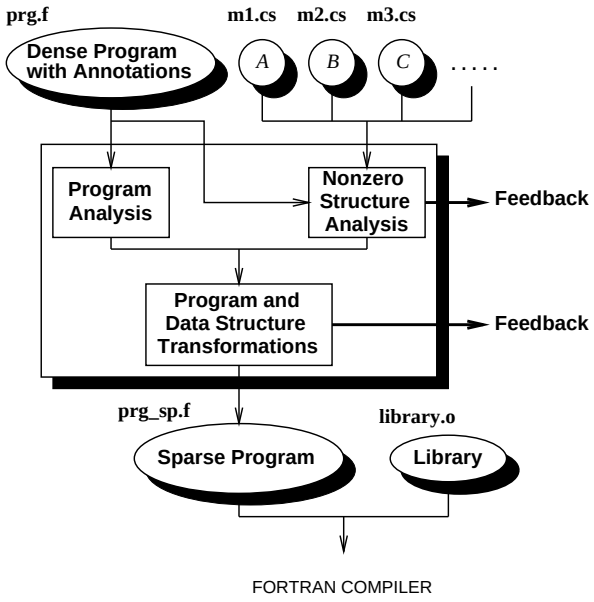


Figure 4.12: Organization of the Sparse Compiler

All annotations have the form of comments, which enables direct compilation and testing of the original dense program (cf. [92]). The original dense code is analyzed by the sparse compiler to detect statements that can exploit sparsity and to determine the way in which the enveloping data structures are accessed. As indicated by annotations, the input of the sparse compiler may also consist of some implicitly sparse matrices that are available at compile-time on file (cf. '`m1.cs`', '`m2.cs`', and so on). To impose little constraints on programmers not familiar with sparse applications, the compiler expects all matrices in a very simple storage format, namely the coordinate scheme (cf. section 4.1.3). The files are automatically analyzed to determine characteristics of the nonzero structures. These characteristics are supplied to the transformation phase. If desired, the results can also be prompted to the programmer to provide some feedback. In a realistic application, however, not all sparse matrices will be available at compile-time. Even in these cases, characteristics of the nonzero structure may be known in advance. Therefore, annotations to supply nonzero structure information to the sparse compiler have been made available.

Program and data structure transformations are applied to the original dense program in order to obtain semantically equivalent code in which the sparsity of all implicitly sparse matrices is exploited. Restructuring techniques required include procedure cloning [54, 55], access pattern reshaping, iteration space partitioning and actual sparse code generation. Information about this restructuring phase may be prompted to the programmer. This enables the programmer to fix the parts of the original dense program that are transformed into inefficient sparse code.

Finally, the resulting sparse program is saved in a file with the additional extension '`_sp`' to indicate the sparse character of this program (cf. '`prg_sp.f`'). This file is supplied to a conventional FORTRAN compiler that produces machine code for a particular target machine. A library containing some useful primitives, which only has to be compiled once for every possible target machine, is linked with the generated sparse program. Moreover, to keep the sparse storage schemes selected for the implicitly sparse matrices completely transparent to the programmer, the sparse compiler also generates appropriate initialization code at the beginning of the main program. This implies that no initialization code has to be defined in the original dense code (except for some temporarily initialization code for the enveloping data structures that is removed after testing).

4.2.3 Incorporation of Sparse Methods

In addition to methods for dense matrix computations, a vast amount of methods have been developed specifically for sparse matrix computations. In particular, these sparse matrix computations differ from dense applications by the use of sparse storage schemes (see section 4.1.3) and sparsity preserving reordering methods (see appendix A). Both these issues should be addressed by a sparse compiler to enable the automatic generation of efficient sparse codes.

Sparse Storage Schemes

The most efficient sparse storage scheme that can be selected for a sparse matrix A heavily depends on the operations performed on this matrix, and the peculiarities of the nonzero structure of A. To enable the selection of an appropriate sparse storage scheme, the original dense program is analyzed to obtain information about the kind of operations performed, and nonzero structure information is obtained either from annotations or from automatic analysis of matrices on file. The sparse compiler allows for the compile-time selection of a hybrid storage scheme, with static dense storage of regions which are (or become) rather dense, and dynamic storage in a pool of sparse vectors of entries in sparse regions. The layout of vectors over these regions may be different for each region and depends on the most frequently used access direction in that region. However, the actual position of the entries only becomes known at run-time. Primitives in the library support the run-time manipulation of the sparse vectors in this pool.

To support this kind of data structure selection, annotations are available to specify a file that should be analyzed at compile-time, or to identify the dense or sparse regions in a matrix. Regions that are completely zero, and will remain so at run-time, can also be identified. No storage is allocated by the compiler for these regions. Moreover, an attempt is made to remove code performing redundant operations on these regions at compile-time. If zero regions are detected by automatic nonzero structure analysis, the compiler first inquires the programmer whether these regions actually remain zero at run-time.

Sparsity Preserving Reordering Methods

Although reordering methods are occasionally used to enable the use of certain data structures, to increase the amount of exploitable parallelism, or to enhance data locality or vector performance [2, 11, 84, 184], most reordering methods are aimed at preserving sparsity. In the context of solving a sparse system of linear equations, both local strategies as well as a priori reordering methods are used (see appendix A). The use of a reordering method may be essential to keep solving a sparse problem feasible. For example, factorization of the matrix shown in figure 4.13 without pivoting causes complete fill-in, whereas application of minimum degree [96, 98] or reverse Cuthill-Mckee yields a factorization in which no fill-in occurs (cf. [52][78, p96-98,p153-157]).

Because such reordering methods improve the efficiency of a sparse application and reduce storage requirements, a mechanism must be available to incorporate sparsity preserving reordering methods in the automatically generated sparse code. One possibility would be to let the programmer deal with permutations explicitly by means of e.g. masks, permutation arrays or physical data movement. In fact, this approach is taken in dense applications, where, for instance, partial or complete pivoting are explicitly implemented in the code (see e.g. [90, p58-67][173, 176]). However, this solution is unsuited for the automatic generation of sparse codes, since it obscures the functionality of the code, disables regular data dependence analysis, and reduces the flexibility of the program since only one reordering method can be implemented. Moreover, it is difficult to express sparsity related decisions in the dense code, and much programming effort is wasted since a completely different implementation is required in the resulting sparse code.

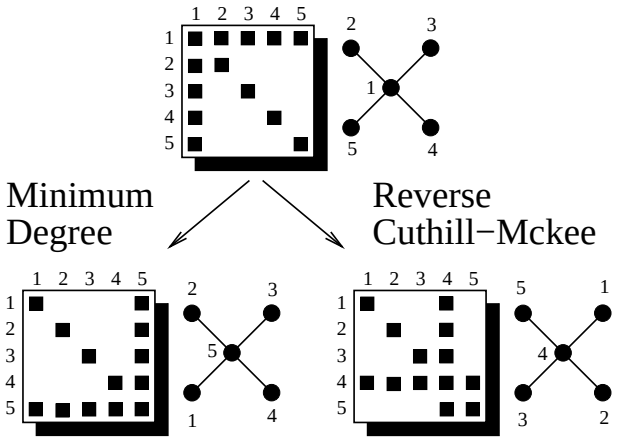


Figure 4.13: The Importance of Reordering Methods for Factorization

Therefore, the sparse compiler provides some elementary support for the incorporation of reordering methods, suited for the incorporation of local strategies and a priori reordering methods The implementation of permutations is kept completely transparent to the programmer and *the compiler is responsible for the generation of sparse code in which suitable permutations are possibly selected, applied, and recorded*. As far as the programmer is concerned, *all programming can be done on the enveloping data structure as if the permutation is performed by physically moving elements in this two-dimensional array*. In addition, rather than specifying a particular method directly (e.g. reverse Cuthill-Mckee), the programmer merely uses some annotations to inform the sparse compiler about the kind of permutations that may be applied to an implicitly sparse matrix at particular positions in the code. After analyzing the program, the sparse compiler selects a suitable reordering method. However, we will see that permutation annotations alone are not sufficient, but we also need annotations to deal with the *mathematical consequences* of permutations.

4.3 Automatic Data Structure Selection and Transformation

The automatic data structure selection and transformation method of the sparse compiler is based on a bottom-up approach and consists of a three phase process. In the first phase, the enveloping data structures and the instructions in the code affected by the sparsity of these data structures are identified. Some preparatory program transformations are applied to simplify subsequent data structure transformations and to make fully use of statements affected by sparsity. In the second phase, a sparse storage scheme is selected for each implicitly sparse matrix, possibly in combination with loop transformations to resolve conflicts. In the third and final phase, the actual data structure transformations are applied and sparse code is generated.

4.3.1 Intuition behind the Automatic Exploitation of Sparsity

If in the original dense program, a two-dimensional array `REAL A(M,N)` is used as the enveloping data structure of an $m \times n$ implicitly sparse matrix matrix A, then this is indicated using the following annotation:

```
      REAL A(M,N)
C_SPARSE(A)
```

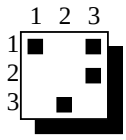
Obviously, the storage requirements of the program can be reduced by converting the array `A` into a sparse storage scheme for the matrix A.

Although there are many ways to store sparse matrices, the most efficient of which depends on the problem considered, some general sparse storage schemes exist. One possible approach to automatic dense into sparse conversion would be to select one of these data structures for every implicitly sparse matrix directly, followed by a corresponding transformation of the program. This approach is too simple, however, as it does not give any control over the efficiency of the resulting code.

To achieve a reasonable level of control, data structure selection must be based on the nonzero structure of the matrix and the actual operations performed by the code. Regarding the latter aspect, this control may be realized by first identifying statements where sparsity can be exploited to save computational time. We can observe that *all instances of an assignment statement in which a zero is assigned to a non-entry or where an arbitrary variable is updated with a zero can be eliminated*, provided that this statement does not call functions with side-effects. Usually, this observation is only exploited for non-entries of sparse data structures, and accidentally stored zero entries or zero elements arising in dense data structures are ignored.

Example: Suppose that below, array A is used as enveloping data structure of an implicitly sparse 3×3 matrix A, of which the nonzero structure is illustrated in the picture:

```
  DO I = 1, 3
    DO J = 1, 3
S1:    ACC    = ACC + A(I,J)
S2:    A(I,J) = A(I,J) * 2.0
    ENDDO
  ENDDO
```



It is clear that, independent of the actual numerical value of each entry, executing only the following statement instances preserves the semantics of the original dense loop:

```
S1(1,1): ACC    = ACC + A(1,1)   S1(2,3): ACC    = ACC + A(2,3)
S2(1,1): A(1,1) = A(1,1) * 2.0   S2(2,3): A(2,3) = A(2,3) * 2.0
S1(1,3): ACC    = ACC + A(1,3)   S1(3,2): ACC    = ACC + A(3,2)
S2(1,3): A(1,3) = A(1,3) * 2.0   S2(3,2): A(3,2) = A(3,2) * 2.0
```

Frequently, statement instances that can exploit sparsity may be executed in arbitrary order. This is certainly true if no cross-iteration data dependences hold (e.g. a loop with only S_2). If cross-iteration data dependences exist, the original execution order must be preserved, although data dependences caused by a simple accumulation (cf. S_1) can be ignored if roundoff errors, due to inexact computer arithmetic, are allowed to accumulate in a different way [228].

A similar observation can be made for IF-statements, since *all statement instances under control of a condition that cannot hold may be skipped*, provided that evaluating the condition is free of any side-effects. For example, only instances of a one-way IF-statement with the condition '(A(I.J).NE.0.0)' that refer to an entry, have to be executed, independently of the statements that actually appear in the body of the IF-statement.

The identification of statements that can exploit sparsity enables the sparse compiler to take a bottom-up approach consisting of three phases. In the first phase, every statement in the program is checked whether it contains occurrences of enveloping data structures. If so, this statement is conceptually an IF-statement, in which a distinction is made between code operating on entries and code that operates on zero elements. For example, a statement in which an occurrence A(I,J) of an enveloping data structure occurs can be thought of as the following IF-statement, where $E(A) \subseteq [1,m] \times [1,n]$ denotes the index set of the entries of the implicitly sparse matrix A:

```
IF ( (I,J) ∈ E(A) ) THEN
  ... A(I,J) ...      ← operation on an entry
ELSE
  ... 0.0    ...      ← operation on a zero element
ENDIF
```

Due to the previous two observations, the ELSE-branch can be eliminated in some cases. Because the presence of **guards** like '$(\mathtt{I},\mathtt{J}) \in E(A)$' reflects the run-time overhead that is inherent to sparse codes, it is desirable to eliminate this overhead. An important source of overhead reduction is the encapsulation of a guard that controls a one-way IF-statement in the execution set of a surrounding loop, as is illustrated below:

```
DO I ∈ V                                  DO I ∈ {V | φ}
  IF ( φ(I) ) THEN          →               ...
    ...                                   ENDDO
  ENDIF
ENDDO
```

Such a conversion is only possible if all elements in the execution set $\{V \mid \phi\}$ can be generated efficiently at run-time. Therefore, information about the way in which enveloping data structures are accessed is collected and propagated to the next phase.

In the second phase, constraints are imposed on the organization of the sparse storage scheme of each implicitly sparse matrix to enable the encapsulation of guards and other overhead reducing techniques, while each data structure is kept as compact as possible to limit the storage requirements. If possible, access patterns are reshaped and execution sets are partitioned to obtain a number of non-overlapping regions in each implicitly sparse matrix that are accessed in a consistent manner. Thereafter, the sparse compiler selects a suitable sparse storage scheme for each implicitly sparse matrix.

Finally, in the third and final phase, the corresponding data structure transformations are applied by converting the dense code into a form that operates on the selected sparse storage schemes.

In the following sections, we briefly glance at each of these phases. A more detailed discussion of the phases can be found in chapters 5, 7, and 8. Nonzero structure analysis and more advanced topics, such as concurrentization and incorporating reordering methods are presented in chapters 6 and 10 respectively.

4.3.2 Phase 1: Program Analysis

In the first phase, conditions are associated with the statements of a program and information is collected about the way in which enveloping data structure are accessed. Moreover, some preparatory program transformations are applied to improve these conditions and to simplify subsequent data structure transformations.

Associating Conditions with Statements

Suppose that an array A is used as enveloping data structure of an $m \times n$ implicitly sparse matrix A, and that the index set of the entries of this matrix (usually not known at compile-time) is denoted by $E(A) \subseteq [1,m] \times [1,n]$. If an occurrence `A(I,J)` appears in a statement, then conceptually this statement is a two-way IF-statement that distinguishes between entries and zero elements.

If we use A' as an abstraction of a sparse storage scheme of A, where a bijective storage function $\sigma_A : E(A) \rightarrow AD'_A$ maps the indices of an entry to the corresponding address in A', we can view a statement in which the enveloping data structure occurs at the right-hand side of an assignment statement as follows, since either a value must be fetched from A' or a zero must be used:

```
                          IF (I,J) ∈ E(A) THEN
                            X = A'[σ_A(I,J)]
X = A(I,J)          →     ELSEIF (I,J) ∉ E(A) THEN
                            X = 0.0
                          ENDIF
```

Making the distinction seems useless in this case, and we rather use the convention that we have $\sigma_A(i,j) = \perp$ if $(i,j) \notin E(A)$ holds, where $\mathtt{A}'[\perp] = 0$, to avoid the distinction shown above. In other cases, however, branches may be eliminated due to the observations made in the previous section. An example is shown below:

```
ACC = ACC + A(I,J)      →      IF (I,J) ∈ E(A) THEN
                                 ACC = ACC + A'[σ_A(I,J)]
                               ENDIF
```

Another, less obvious example where one branch can be eliminated if zero constants are handled as non-entries is shown below:[1]

```
A(I,J) = 0.0      →      IF (I,J) ∈ E(A) THEN
                           A'[σ_A(I,J)] = 0.0
                         ENDIF
```

If an occurrence of an enveloping data structure appears at the left-hand side of an assignment statement in which an arbitrary expressions appears at the right-hand side, then the two branches must handle the change in value for an entry or creation respectively:

```
                         IF (I,J) ∈ E(A) THEN
                           A'[σ_A(I,J)] = X
A(I,J) = X      →        ELSEIF (I,J) ∉ E(A) THEN
                           A'[new_A(I,J)] = X
                         ENDIF
```

We have used the function new_A to insert a new entry in A. This function returns the address of a new entry a_{ij}, and adapts the storage function σ_A, the index set of the entries $E(A)$ and the set of addresses AD'_A accordingly as side-effects.

The '$\in E(A)$'- and '$\notin E(A)$'-tests are referred to as **positive** and **negative guards** respectively. Every *different* occurrence of an enveloping data structures gives rise to an additional positive and negative guard. Hence, statements with k different occurrences of enveloping data structures can be thought of as a multi-way IF-statements with 2^k branches for every possible conjunction of the corresponding guards. An example is given below, where we assume that array B is also used as enveloping data structure. The last branch has been eliminated to exploit sparsity:

```
X = X + A(I,J) + B(I,J)      →      IF (I,J) ∈ E(A) ∧ (I,J) ∈ E(B) THEN
                                      X = X + A'[σ_A(I,J)] + B'[σ_B(I,J)]
                                    ELSEIF (I,J) ∈ E(A) ∧ (I,J) ∉ E(B) THEN
                                      X = X + A'[σ_A(I,J)]
                                    ELSEIF (I,J) ∉ E(A) ∧ (I,J) ∈ E(B) THEN
                                      X = X + B'[σ_B(I,J)]
                                    ENDIF
```

We define the **condition** of a statement as the disjunction of all conditions (consisting of conjunctions of guards) that appear in the remaining branches of the multi-way IF-statement, i.e. the branches that cannot exploit sparsity. For the previous example, for instance, the associated condition is:

$$\begin{array}{lll} (& ((\mathtt{I},\mathtt{J}) \in E(A) \wedge (\mathtt{I},\mathtt{J}) \in E(B)) & \vee \\ & ((\mathtt{I},\mathtt{J}) \in E(A) \wedge (\mathtt{I},\mathtt{J}) \notin E(B)) & \vee \\ & ((\mathtt{I},\mathtt{J}) \notin E(A) \wedge (\mathtt{I},\mathtt{J}) \in E(B)) &) \quad \equiv \quad (\mathtt{I},\mathtt{J}) \in E(A) \vee (\mathtt{I},\mathtt{J}) \in E(B) \end{array}$$

This condition, which has been simplified into a single disjunction of guards, reflects the fact that all instances referring to at least one entry must be executed so that only the instances that access two non-entries can fully exploit sparsity (if we ignore the fact that some entries may accidentally be zero). In general, a condition of a statement defines the instances of the statement that cannot exploit sparsity and, hence, must be executed. Another example is shown below:

[1] Note that in this case, the entry could be eliminated from the sparse storage scheme in the remaining branch. Usually, however, the situation in which an entry becomes zero is simply ignored.

```
X = X + A(I,K) * B(K,J)      →      IF (I,K) ∈ E(A) ∧ (K,J) ∈ E(B) THEN
                                      X = X + A'[σ_A(I,J)] * B'[σ_B(I,J)]
                                    ENDIF
```

Condition '$(\texttt{I},\texttt{K}) \in E(A) \wedge (\texttt{K},\texttt{J}) \in E(B)$' is associated with this statement, which indicates that only the instances in which two entries are accessed have to be executed. In some cases, we can also associate conditions with IF-statements. Condition '$(\texttt{J},\texttt{I}) \in E(A)$', for example, may be associated with the following one-way IF-statement, since the boolean expression '`(A(J,I).GT.ABS(X))`' necessarily evaluates to false for all non-entries:

```
IF (A(J,I) .GT. ABS(X)) THEN          IF (J,I) ∈ E(A) THEN
  ...                          →         IF (A'[σ_A(J,I)] .GT. ABS(X)) THEN
ENDIF                                      ...
                                         ENDIF
                                       ENDIF
```

Obviously, it would be extremely cumbersome to first generate multi-way-IF statements, followed by eliminating branches that can exploit sparsity and computing the condition for each statement according to the remaining branches. Fortunately, it is not necessary to explicitly construct multi-way IF-statements to compute the condition of each statement. In chapter 5, we present a simple attributed grammar that directly computes the condition for each statement. The conditions computed by this attributed grammar only exploit zero constants and non-entries, i.e. the conditions conservatively assume that the contents of dense variables and entries are always nonzero. Although this implies that accidentally stored zeros are not exploited, it prevents the generation of additional tests on the value of such expressions which are likely to fail.

Dominating Guards

For some statements, the value of the associated condition is completely dependent on the value of one particular positive guard. Such situations frequently occur in linear algebra operations that are classified as static or simply dynamic [235, p10-12]. For example, implementing the operation $A \leftarrow \alpha \cdot A$, which is called a simply dynamic operation, involves the following assignment statement:

```
A(I,J) = ALPHA * A(I,J)      →      IF (I,J) ∈ E(A) THEN
                                      A'[σ_A(I,J)] = ALPHA * A'[σ_A(I,J)]
                                    ENDIF
```

We say that a positive guard ψ **dominates** a condition ϕ, if $\phi \Rightarrow \psi$ holds. Informally speaking, if the guard ψ does not hold, then the whole condition ϕ does not hold. In this previous example, the guard '$(\texttt{I},\texttt{J}) \in E(A)$' dominates the identical condition. As another example, both the guards in the condition '$(\texttt{I},\texttt{K}) \in E(A) \wedge (\texttt{K},\texttt{J}) \in E(B)$' dominate this condition, whereas none of these guards dominates the condition '$(\texttt{I},\texttt{K}) \in E(A) \vee (\texttt{K},\texttt{J}) \in E(B)$'. Dominating guards give rise to a convenient overhead reduction method, because sparse data structures usually provide efficient generation of all entries, i.e. elements for which the guard holds, along an access pattern. Therefore, information about the way enveloping data structures are accessed is collected and propagated to the second phase.

Preparatory Program Transformations

During the first phase, some preparatory program transformations are applied to simplify subsequent data structure transformations and to improve the conditions that become associated with the statements in the program. For example, because in the following fragment another data structure will be selected for array A, the occurrences of the formal argument M in subroutine SUB must be converted accordingly to account for the fact that this subroutine may be called with A as actual argument:

```
      REAL A(10,10), X(100), D(10,10)        SUBROUTINE SUB(M)
C_SPARSE(A)                                  REAL M(10,10)
      ...                                    ...
      CALL SUB(A)                            RETURN
      CALL SUB(X)                            END
      CALL SUB(D)
      END
```

However, since this subroutine is also called with X and D as actual arguments, we cannot bluntly apply program and data structure transformations to SUB. Instead, we first apply procedure cloning [54, 55] to construct a clone 'SUB_A' of the subroutine, in which there is a unique association between the enveloping data structure A and the formal argument M. Thereafter, the first CALL-statements in the main program is converted accordingly and program and data structure transformations can be applied to the clone without interfering with calls to SUB having dense actual arguments.

Some transformations that improve the conditions associated with statements may also be applied. For example, using the attributed grammar that will be presented in chapter 5, condition '**true**' becomes associated with the following two assignment statements because it seems that none of the instances of these statements can exploit sparsity. However, after scalar forward substitution [234, p178-179], the condition of the remaining statement changes into '$(\texttt{I}, \texttt{J}) \in E(A)$':

```
T   = A(I,J)
ACC = ACC + 3.0 * T      →    ACC = ACC + 3.0 * A(I,J)
```

4.3.3 Phase 2: Data Structure Selection

Each occurrence of an enveloping data structure induces access patterns that form paths through the index set of the corresponding implicitly sparse matrix. The elements along every path can be viewed as a vector that, depending on the regions accessed, is either dense, sparse or zero. In the second phase, the sparse compiler selects a storage scheme for each implicitly sparse matrix. Static dense storage is used for the vectors through dense regions, whereas a pool of sparse vectors is selected as dynamic storage for the remaining regions. The layout of the vectors over these regions is selected to enable overhead reducing techniques as much a possible. Loop transformations are used to obtain a number of non-overlapping regions that are accessed in a consistent manner.

Overhead Reducing Techniques

Although the multi-way IF-statements used during the presentation of the first phase are not explicitly generated, the presence of guards, σ_A-lookups and *new*$_A$-functions reflect the overhead inherent to sparse storage schemes for scanning a compact data structure to determine *if* and *where* an entry is stored or for inserting an entry. Because skipping operations by means of conditionals does not reduce the execution time on most machines [78, 169], overhead reducing techniques are required. The prototype sparse compiler has the ability to apply one of the following overhead reducing techniques: (i) replacing accesses to zero regions by zero, (ii) avoiding overhead by using static dense storage for particular regions, (iii) applying so-called guard encapsulation, where a construct that iterates over the entries along an access pattern is generated, or (iv) applying so-called access pattern expansion, where a sparse vector is scattered into a full-sized array before operations are applied to this vector, and gathered back into sparse storage thereafter.

Zero Replacement

If we know that a particular region in an implicitly sparse matrix A is completely zero and will remain so at run-time, then all occurrences of the corresponding enveloping data structure that can only induce accesses to this region can be replaced by the constant zero at compile-time.

If, for example, we know that the whole main diagonal of an implicitly sparse matrix A having A as enveloping data structure is completely zero and will remain so at run-time, then the following replacement becomes possible:

```
ACC = ACC + A(5,5)     →     ACC = ACC + 0.0
```

Because the condition '**false**' becomes associated with the resulting statement, the whole assignment statement can be eliminated thereafter.

Using Static Dense Storage

If a particular region in an implicitly sparse matrix is rather dense, then we can use static dense storage for that region to avoid the overhead that is inherent to sparse storage schemes. For instance, using a one-dimensional array DIAG_A to store all elements along the main diagonal of an implicitly sparse matrix A having the two-dimensional array A as enveloping data structure avoids all lookup overhead in the following fragment (another storage organization may be used for the remaining regions):

```
      REAL A(N,N)
C_SPARSE(A)                              REAL DIAG_A(N), ...
      ...                                ...
      DO I = 1, N          →             DO I = 1, N
        X(I) = A(I,I)                      X(I) = DIAG_A(I)
      ENDDO                              ENDDO
```

If some zero elements along the main diagonal become nonzero, the use of static dense storage even avoids any overhead arising from the run-time insertion of entries. On the other hand, using static dense storage also disables any exploitation of sparsity to reduce either storage requirements or computational time. For example, although condition '$(\mathtt{I},\mathtt{I}) \in E(A)$' is associated with the following statement, sparsity cannot be exploited (viz. selecting static dense storage of the main diagonal implies that the inclusion $\{(1,1),(2,2),\ldots\} \subseteq E(A)$ is already known at compile-time):

```
ACC = ACC + A(I,I)    →    ACC = ACC + DIAG_A(I)
```

Only if the main diagonal is reasonably dense, the number of unnecessarily performed operations and stored zero elements is small. Therefore, static dense storage should only be selected for regions that are or become rather dense, whereas sparse storage should be used for the remaining regions, using the overhead reducing techniques of the following sections where possible.

Guard Encapsulation

For an occurrence of an enveloping data structure that has admissible subscripts represented by the affine transformation $F(\vec{\mathtt{I}}) = \vec{v} + W\vec{\mathtt{I}}$ appearing in a loop with index vector $\vec{\mathtt{I}} = (\mathtt{I}_1,\ldots,\mathtt{I}_d)^T$, we define the **access pattern** $P(\mathtt{I}_1,\ldots,\mathtt{I}_{d-1}) \subseteq \mathcal{Z}^2$ as the index set of all elements accessed in successive iterations of the innermost stride-1 $\mathtt{I}_d$-loop with bounds $\mathtt{L}_d$ and $\mathtt{U}_d$:

$$P(\mathtt{I}_1,\ldots,\mathtt{I}_{d-1}) = \{F(\vec{\mathtt{I}})^T \mid \mathtt{I}_d \in [\mathtt{L}_d,\mathtt{U}_d]\}$$

If guard '$F(\vec{\mathtt{I}}) \in E(A)$' dominates the condition of all statements in the loop-body, we would like to let index $\mathtt{I}_d$ iterate over all values in the irregular execution set $\{\mathtt{I}_d \in [\mathtt{L}_d,\mathtt{U}_d] \mid F(\vec{\mathtt{I}}) \in E(A)\}$. This is obtained by **guard encapsulation**. In this manner, only iterations in which an entry is operated on, i.e. an element for which the dominating guard holds, are executed and test overhead vanishes. Under certain conditions, guard encapsulation can be implemented by iterating over the *entries* along each access pattern, as is illustrated below:

```
DO I_1 = L_1, U_1                                    DO I_1 = L_1, U_1
  ...                                                  ...
    ...                                                  ...
    DO I_d = L_d, U_d                                    DO ad ∈ AD'_A(I_1, ..., I_{d-1})
      IF F(I⃗) ∈ E(A) THEN                                 ACC = ACC + A'[ad]
        ACC = ACC + A'[σ_A(F(I⃗))]      →                 ENDDO
      ENDIF                                              ...
    ENDDO                                              ...
    ...                                              ENDDO
  ...
ENDDO
```

This conversion is feasible if the last column of matrix W is nonzero *and* the organization of the pool of sparse vectors satisfies the following constraints:

(a) For all possible values of the loop indices $\mathtt{I}_1, \ldots, \mathtt{I}_{d-1}$, the addresses in the following set can be generated efficiently:

$$AD'_A(\mathtt{I}_1, \ldots, \mathtt{I}_{d-1}) \quad = \quad \{ \quad \sigma_A(f) \mid f \in P(\mathtt{I}_1, \ldots, \mathtt{I}_{d-1}) \cap E(A) \quad \}$$

(b) To restore the value of loop index $\mathtt{I}_d$, either the value $\pi_1 \cdot \sigma_A^{-1}(ad)$ if $w_{1d} \neq 0$ holds, or the value $\pi_2 \cdot \sigma_A^{-1}(ad)$ if $w_{2d} \neq 0$ holds, can be supplied efficiently together with each address $ad \in AD'_A(\mathtt{I}_1, \ldots, \mathtt{I}_{d-1})$, where $\pi_i \cdot \vec{x} = x_i$.

Furthermore, to prevent the requirement for *ordered* storage (not supported in the current prototype sparse compiler), this conversion is only done if iterations of this DO-loop may be executed in arbitrary order (although, as discussed in section 4.3.1, data dependences caused by accumulations may be ignored):

(c) No data dependence is carried by the $\mathtt{I}_d$-loop and no exit branch [234, p238-241] or STOP-statement can be executed in the loop-body of this DO-loop.

Because, in general, it is very likely that some access patterns are subsets of other access patterns, we usually relax constraint (a) to the requirement that the addresses of entries along a *longitudinal enveloping* access pattern of each true access pattern can be generated efficiently, where a longitudinal enveloping access pattern of an access pattern simply consists of all discrete points on an arbitrary line segment that is placed over the access pattern. However, in this case, we must test if the restored value of the loop index $\mathtt{I}_d$ is an *integer* value in the range $[\mathtt{L}_d, \mathtt{U}_d]$ to determine whether an entry actually corresponds to the true access pattern. Constraint (b) follows from $ad = \sigma_A(\vec{v} + W\vec{\mathtt{I}})$ and the fact that the storage function σ_A is invertible:

$$\left\{ \begin{array}{lcl} \pi_1 \cdot \sigma_A^{-1}(ad) & = & v_1 + w_{11} \cdot \mathtt{I}_1 + \ldots + w_{1d} \cdot \mathtt{I}_d \\ \pi_2 \cdot \sigma_A^{-1}(ad) & = & v_2 + w_{21} \cdot \mathtt{I}_1 + \ldots + w_{2d} \cdot \mathtt{I}_d \end{array} \right. \tag{4.5}$$

Consequently, for column-wise access pattern (viz. $w_{1d} \neq 0$), the value $\pi_1 \cdot \sigma_A^{-1}(ad)$ is required (in combination with the values of the indices of more outer DO-loops) to restore the value of the loop index $\mathtt{I}_d$. This clearly illustrates that row index information is required in column-wise oriented data structures. Likewise, for row-wise access patterns (viz. $w_{2d} \neq 0$), the value $\pi_2 \cdot \sigma_A^{-1}(\mathtt{AD})$, i.e. the column index, is required, while for diagonal-wise access patterns (viz. $w_{1d} \neq 0$ or $w_{2d} \neq 0$) either row or column index information suffices to solve an equation in (4.5).
Example: Below, array A is used as enveloping data structure of a 100×100 implicitly sparse matrix A, and guard '$(\mathtt{I}, \mathtt{I}+\mathtt{J}) \in E(A)$', dominates the (identical) loop condition. Hence, guard encapsulation in the execution set of the J-loop is feasible if (a) the addresses of entries along the access patterns $P(\mathtt{I}) = \{(\mathtt{I}, \mathtt{I}+\mathtt{J}) \mid 1 \leq \mathtt{J} \leq 75\}$ for $1 \leq \mathtt{I} \leq 25$ can be generated efficiently, (b) $\pi_2 \cdot \sigma_A^{-1}(ad)$-values are available together with each address, and (c) the data dependence caused by the accumulation may be ignored:

```
DO I = 1, 25
  DO J = 1, 75
    X = X + A(I,I+J) * J
  ENDDO
ENDDO
```

→

```
DO I = 1, 25
  DO ad ∈ {σ_A(f) | f ∈ P(I) ∩ E(A)}
    J = π_2 · σ_A^-1(ad) - I
    X = X + A'[ad] * J
  ENDDO
ENDDO
```

Relaxing constraint (a) enables the sparse compiler to select, for instance, general row-wise storage of the entries but also requires the test '$\mathtt{J} \in [1, 75]$' before the value of X is actually updated. Although, in general, test overhead remains in the loop-body, still fewer iterations are executed, since the average number of entries along each longitudinal enveloping access pattern is probably less than the size of the original execution set.

It is important to realize that the requirement for fast generation of entries along an access pattern imposes constraints on the *organization* of the pool of sparse vectors. In general, the actual addresses of the entries and their position in the matrix will only be known at run-time. Moreover, if creation may occur in A, this complicates the actual implementation of guard encapsulation, because this affects the value of guards, whereas addresses may change due to data movement or, for some data structures, due to an occasionally required left compression. For scalar-wise access patterns (viz. both w_{1d} and w_{2d} are zero), it is possible that the guard can be hoisted out of some innermost DO-loop, which enables guard encapsulation in the execution set of a more outer DO-loop. These issues are further elaborated in chapter 8.

Access Pattern Expansion

A related overhead reducing technique imposing similar constraints on the organization of the pool of sparse vectors is given by **access pattern expansion**. The compiler may decide to generate a construct that will expand the entries along an access pattern stored in dynamic sparse storage into a temporary one-dimensional array using a scatter-operation, so that subsequent operations on this vector do not suffer from the inherent sparse lookup overhead [69, 78, 169]. The actual number of operations performed, however, is not reduced by this technique.

For example, if the addresses of entries along $P = \{(\mathtt{10}, \mathtt{J}) \mid \mathtt{1} \leq \mathtt{J} \leq \mathtt{N}\}$ can be generated efficiently for an implicitly sparse matrix A with enveloping data structure A, and the value $\pi_2 \cdot \sigma_A^{-1}(ad)$ is available together with each address, then the following fragment, in which sparsity cannot be exploited to reduce computational time, can be implemented without repeated lookup overhead as follows:

```
DO J = 1, N
  D(10,J) = D(10,J) * A(10,J)
ENDDO
```

→

```
DO ad ∈ {σ_A(f) | f ∈ P ∩ E(A)}      } scatter
  AP(π_2 · σ_A^-1(ad)) = A'[ad]      }
ENDDO
DO J = 1, N
  D(10,J) = D(10,J) * AP(J)
ENDDO
```

If changes to the nonzero structure along the 10th row occur, then this can be easily accounted for using the so-called switch technique [169], where a switch array records which elements are entries. Storage is obtained directly to account for creation if necessary and the actual values are stored back into sparse storage afterwards with a gather-operation. During this operation, the switch array and the full-sized array are reset to support any subsequent access pattern expansion, as is further elaborated in chapter 8.

In any case, access pattern expansion is feasible if the organization of the pool of sparse vectors satisfies constraints (a) and (b) of the previous section.

Supporting Transformations

To enable guard encapsulation and access pattern expansion, the sparse compiler must select for each implicitly sparse matrix an organization of the pool of sparse vectors that supports efficient generation of the addresses of entries along certain access patterns. Ideally, we would like to support fast generation of addresses of entries along all access patterns through the corresponding enveloping data structures that are encountered in the program. However, this would demand for extensive overhead storage.

Consequently, it is desirable to have a number of non-overlapping regions in each implicitly sparse matrix that are accessed in a consistent way. Loop transformations can assist in achieving this goal. Unimodular loop transformations can be used to reshape the access patterns of an occurrence of an enveloping data structure. For example, simple loop interchanging changes the row-wise access patterns of the following occurrences into column-wise access patterns:

```
DO I = 1, M                          DO J = 1, N
  DO J = 1, N                          DO I = 1, M
    A(I,J) = A(I,J) * 3.0      →         A(I,J) = A(I,J) * 3.0
  ENDDO                                ENDDO
ENDDO                                ENDDO
```

Likewise, iteration space partitioning can be used to obtain a number of non-overlapping regions as is illustrated below:

```
DO I = 1, M
  A(I,I) = 10.0
  DO J = 1, I                  I ≤ J ≤ I        DO I = 1, M
    D(I,J) = A(I,J) * X(J)        →               A(I,I) = 10.0
  ENDDO                                          DO J = 1, I - 1
ENDDO                                              D(I,J) =  A(I,J) * X(J)
                                                 ENDDO
                                                 D(I,I) = A(I,I) * X(I)
                                               ENDDO
```

After this loop transformation, a different storage organization can be selected for the main diagonal and the strict lower triangular part of an implicitly sparse matrix A. In chapter 7, we discuss methods to control these loop transformations and to update information that has been obtained in the first phase incrementally to avoid re-analysis of the program.

Data Structure Selection

Eventually, the sparse compiler selects a sparse storage scheme for each implicitly sparse matrix. Static dense storage is used for dense regions, whereas a pool of sparse vectors is selected as dynamic storage for the remaining regions. The layout of the vectors over these regions is selected according to the way in which each region is accessed most frequently, in order to enable guard encapsulation and access pattern expansion as much as possible.

The pool of sparse vectors is stored using a variant of the sparse row- or column-wise schemes presented in section 4.1.3. The numerical values of entries in each sparse vector are stored consecutively in an array VAL_A, while the corresponding column or row indices are stored in a parallel integer array IND_A. The pointers LOW_A(p) and HGH_A(p) are used to locate the pth sparse vector in the pool. In figure 4.14, possible contents of these arrays for a general sparse row-wise organization of the pool are shown. In general, however, the layout of sparse vectors over the sparse regions may be selected arbitrarily by the sparse compiler, so that the number of sparse vectors in the pool may exceed the number of rows or columns of the matrix. Row index information is available for column-wise sparse vectors and column index information for sparse vectors stored along other directions.

Obviously, selecting different storage formats for different regions in an implicitly sparse matrix enables the sparse compiler to fully account for both the characteristics of the nonzero structure of this matrix, as well as for the actual operations applied to the matrix.

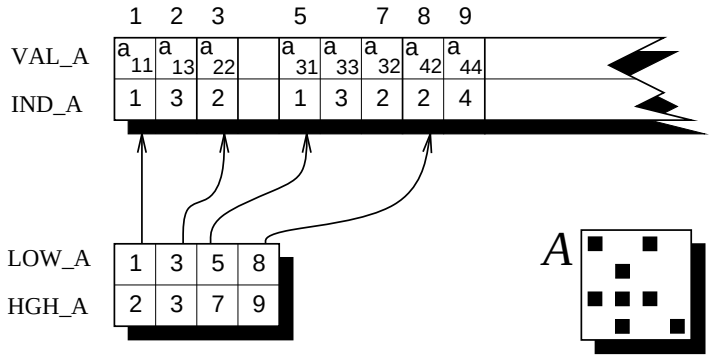


Figure 4.14: Dynamic Storage in a Pool of Sparse Vectors

4.3.4 Phase 3: Sparse Code Generation

The actual data structure transformations are applied by converting the dense code into a form that operates on the selected sparse storage schemes, where overhead reducing techniques are used as much as possible.

For example, the guard encapsulation of section 4.3.3 can be implemented as follows for a general sparse row-wise organization of the pool of sparse vectors:

```
DO I = 1, 25
  DO J = 1, 75
    X = X + A(I,I+J) * J
  ENDDO
ENDDO
```

→

```
DO I = 1, 25
  DO J_ = LOW_A(I), HGH_A(I)
    J = IND_A(J_) - I
    IF ((1.LE.J).AND.(J.LE.75)) THEN
      X = X + VAL_A(J_) * J
    ENDIF
  ENDDO
ENDDO
```

Some frequently occurring constructs are supplied as primitives in a separate library and used in the generated sparse code to reduce the size of this code. Moreover, because the programmer is unaware of the sparse storage scheme that is eventually selected by the sparse compiler, the sparse compiler is responsible for generating appropriate initialization code for each selected data structure. This code is generated at the beginning of the main program, and expects each implicitly sparse matrix in coordinate scheme on file. The file names are supplied to the sparse compiler either interactively or by means of annotations. A detailed presentation of code generation is given in chapter 8.

Chapter 5

Phase 1: Program Analysis

During the first phase, the original dense program is analyzed by the sparse compiler. First, the annotations appearing in the program are analyzed to identify the two-dimensional arrays used as enveloping data structures of implicitly sparse matrices and to obtain information about these matrices. Because, eventually, the data structures of implicitly sparse matrices will change, problems arise if these matrices are passed as parameters to subroutines and functions. These problems are solved by enforcing a unique association between enveloping data structures and formal arguments by means of procedure cloning [54, 55], which is a useful mechanism to differentiate between call-sites with different properties. Thereafter, conditions are associated with the statements in the resulting program and, possibly, improved by some simple transformations. Finally, information about the way in which enveloping data structures are accessed is collected.

5.1 Annotations

Information that cannot be expressed in the original dense program is supplied to the sparse compiler by means of annotations. We distinguish between annotations that may appear in the declarative part, mainly used to identify the enveloping data structures, and annotations that may appear in the executable part.

5.1.1 Annotations in the Declarative Part

In the declarative part, annotations are used to identify the enveloping data structures in a program and to supply information about the corresponding implicitly sparse matrices.

Declaration Annotations

Because the compiler cannot distinguish between ordinary arrays and arrays that are used as enveloping data structures of implicitly sparse matrices, a mechanism to provide the compiler with this kind of information must be available. The identification of enveloping data structures is done by means of annotations. All annotations in the declarative part start at the beginning of a line with '`C_SPARSE`'. In this manner, the annotations are simply handled as comments by other compilers, so that the original dense program can be compiled and tested without any modifications, provided that the implicitly sparse matrices are not too large. In each declaration annotation, a parenthesized list of the identifiers of enveloping data structures is given, separated by semi-colons.

Declaration annotations are generated by the following context free grammar (unless stated otherwise, each token denotes the literal string in either lower or upper case):

```
sp_decl     →   C_SPARSE '(' decl_list ')'
            ;
decl_list   →   decl_list ';' decl
            |   decl
            ;
```

In its most simple form, a declaration annotation consists of single identifier specifying the name of an enveloping data structure. However, information about the corresponding implicitly sparse matrix can also be supplied within the declaration annotation as illustrated below, where the token ID denotes an arbitrary FORTRAN identifier:

```
decl   →   ID
       |   ID ':' info
       ;
```

The following annotation, for instance, informs the compiler about the fact that arrays A and B are used as enveloping data structures of two implicitly sparse matrices A and B of size 100×100 and 20×50 respectively:

```
      REAL A(100,100), B(20,50)
C_SPARSE(A ; B)
```

Each annotation that identifies a particular enveloping data structure must follow the actual declaration of the corresponding array. Such annotations are only allowed *in the main program* and before the first executable statement (although sparsity information can be propagated automatically to subroutines and functions). Moreover, each enveloping data structure must be a two-dimensional array with basis type INTEGER, REAL, DOUBLE PRECISION, or COMPLEX, and the index set $[1..M] \times [1..N]$ for suitable constants M and N. If any of these constraints is violated, an appropriate warning is generated and the incorrect part of the annotation is ignored.
Example: In the following program, the annotations involving arrays A and B in the main program, and the local array G of subroutine PROC violate these constraints and, hence, are ignored:

```
      PROGRAM ANNOT                      SUBROUTINE PROC(F)
      REAL A(-5:5,10), B(10)             REAL F(10,10)
      REAL C(10,10), D(100)              REAL G(10,10)
C_SPARSE(A ; B ; C)              C_SPARSE(G)
      CALL PROC(C)                       ...
      CALL PROC(D)                       RETURN
      END                                END
```

Hence, only the two-dimensional array C is handled as an enveloping data structure. As further explained in section 5.2, procedure cloning is used to construct a clone of PROC, called 'PROC_C', in which there is a unique association between C and F. Hence, the sparsity of C is propagated into this clone. The original subroutine PROC with a dense formal argument F is preserved to handle the call with actual argument D.

File Annotations

As shown below, the file in which the corresponding implicitly sparse matrix resides can be specified within a declaration annotation, where the token STRING denotes any sequence of characters enclosed by single quotes:

```
info   →   _FILE '(' STRING ')'
       ;
```

If several files are specified for the same implicitly sparse matrix, only the first file is recorded for this matrix. If this file is available at compile-time in either the current directory or the directory defined by the environment variable SPARSEDIR, then the file is examined by the automatic nonzero structure analyzer described in chapter 6.

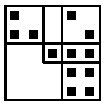
Because this analyzer expects all matrices in coordinate scheme (cf. section 4.1.3), sparse matrices generated in a program or stored in alternative storage schemes must first be converted into coordinate scheme in order to enable this automatic analysis. On file, the coordinate scheme consists of three integers, indicating the number of rows, columns, and entries in the matrix, respectively, followed by the row and column index and numerical value of each nonzero element in arbitrary order. If the size of the stored matrix does not match the declaration of the enveloping data structure, a warning is generated and the results of the analysis are ignored.
Example: The following annotation indicates that the implicitly sparse matrix A having array `A` as enveloping data structure can be found in the file '`mat1.cs`':

```
      REAL A(5,5)
C_SPARSE(A : _FILE('mat1.cs'))
```

Below, we show an example file and the corresponding nonzero structure of the matrix A, annotated with the block form detected by the analyzer:

contents of file '`mat1.cs`'

```
5 5 12
1 1 5.0 2 2 5.0 3 3 5.0 4 4 5.0
5 5 5.0 2 1 1.0 3 4 1.0 1 4 1.0
2 5 1.0 5 4 1.0 4 5 1.0 3 5 1.0
```



Nonzero Structure of A

Because the programmer is unaware of the sparse storage scheme that will be selected by the sparse compiler, the compiler is responsible for the generation of appropriate initialization code for each selected data structure. Independent of whether a file specified in a file annotation is available at compile-time, or not, this file will be used in the automatically generated initialization code of the selected sparse storage scheme. This code is generated before the first executable statement in the main program, and, in contrast with automatic analysis of the nonzero structure, will only read the file at run-time. The automatic generation of initialization code is discussed in detail at the end of chapter 8.

Nonzero Structure Annotations

Because it is unlikely that at compile time, all implicitly sparse matrices are available on file, nonzero structure information can also be supplied directly to the compiler by means of annotations. An approximation of the density of an implicitly sparse matrix can be supplied in a declaration annotation as follows, where symbol `expr` denotes an arbitrary FORTRAN expression:

```
info  →  _DENSITY '(' expr ')'
      ;
```

The approximated density is only recorded if the expression can be evaluated at compile-time and has a real value in the range $(0, 1]$. If several approximations are supplied for an implicitly sparse matrix, only the first density that can be evaluated is recorded.

If a particular region in an implicitly sparse matrix is either sparse, dense (or will become so at run-time), or completely zero (and will remain so at run-time), then this information can be supplied to the compiler by specifying this property followed by a description of the index set of the region. Moreover, a preferred access direction for this region can optionally be supplied. The production for such annotations is shown below:

```
info  →  prop '(' bpair_list ')' opt_dir
      ;

prop  →  _SPARSE
      |  _DENSE
      |  _ZERO
      ;

opt_dir  →  '(' expr ',' expr ')'"
         |  ϵ
         ;
```

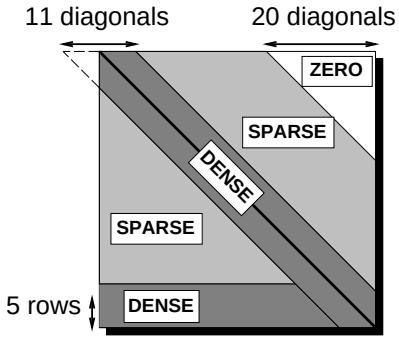


Figure 5.1: Nonzero Structure of B

The index set of a region is described using a, possibly empty, list of boundary pairs, as specified by the following productions:

```
bpair_list  →  bpair_list ',' bpair
            |  bpair
            |  ϵ
            ;
bpair       →  expr '<='     'I'     '<=' expr
            |  expr '<='     'J'     '<=' expr
            |  expr '<=' 'I' '+' 'J' '<=' expr
            |  expr '<=' 'I' '-' 'J' '<=' expr
            ;
```

Symbolic constants may be used in each expression to increase the flexibility of the program, provided that all expressions can be evaluated at compile-time. For an $m \times n$ implicitly sparse matrix A, the region consists of all elements $a_{\mathtt{I,J}}$ with indices $(\mathtt{I}, \mathtt{J}) \in [1..m] \times [1..n]$ that satisfy all the constraints in the list of boundary pairs simultaneously.[1] In this manner, we can describe regions with an index set that can be expressed in terms of a two-dimensional simple section [15, 16]. Empty regions or regions that overlap with earlier supplied regions in a particular implicitly sparse matrix are ignored. All regions that are not specified are assumed to be sparse.

Example: The following annotations indicate that the nonzero structure of an implicitly sparse matrix B with enveloping data structure B has the characteristics shown in figure 5.1, where the preferred access direction of the first dense region is $(1, 1)^T$:

```
      INTEGER   N
      PARAMETER (N=100)
      REAL      B(N,N)
C_SPARSE(B : _DENSE( -5 <= I-J <=    5)(1,1)           )
C_SPARSE(B : _ZERO (1-N <= I-J <= 20-N)                )
C_SPARSE(B : _DENSE(N-4 <= I   <=    N, 6 <= I-J <= N-1))
```

For this matrix B, the sparse compiler will attempt to isolate operations on the different regions, to eliminate redundant operations on the upper right corner, and to reshape the access patterns of occurrences that access the band along the preferred access direction. If these attempts are successful, a storage scheme will be selected in which the band and border are stored statically in two rectangular arrays, and the entries in the sparse regions are stored dynamically in a pool of sparse vectors.

Permutation Annotations

The programmer can specify that an $m \times n$ implicitly sparse matrix A with enveloping data structure A will be permuted into PAQ at run-time using the following permutation annotation:

[1] The indices I and J only serve a notational purpose, and have no relation with FORTRAN variables.

```
      REAL A(M,N)
C_SPARSE(A : _PERM)
```

If the identifier specified in a permutation annotation does not correspond to an enveloping data structure, the annotation is simply ignored because there is no support to reorder dense data structures. Otherwise, the annotation indicates that any programmer-defined a priori reordering method may be applied to A at run-time before this implicitly sparse matrix is initialized. Moreover, the programmer can use interchange annotations (see next section) to indicate positions in the code where certain row and column interchanges may be applied to A, thereby enabling the sparse compiler to select and implement a local strategy.

The actual implementation of permutations is kept transparent to the programmer. The sparse compiler is responsible for generating code in which permutations are possibly selected, applied, and recorded. As far as the programmer is concerned, all programming can be done on the enveloping data structure A as if elements are physically moved in this two-dimensional array, i.e. if at a certain moment A is permuted into PAQ, then the programmer may assume that A contains the elements of PAQ.

5.1.2 Annotations in the Executable Part

In this section, annotations that may appear in the executable part of the original dense program are discussed. Interchange annotations support the incorporation of local strategies. Induction annotations deal with the mathematical consequences of permutations. Because the sparse compiler enforces a unique association between enveloping data structures and formal arguments, in contrast with declaration annotations, the annotations discussed in this section may also appear in subroutines and functions.

Interchange Annotations

The following interchange annotation can be used to specify that at a particular position in the code, at run-time an arbitrary row and column of an implicitly sparse matrix A with enveloping data structure A that are in the range $[\texttt{LR}, \texttt{UR}]$ and $[\texttt{LC}, \texttt{UC}]$ respectively may be interchanged with the Rth row and Cth column:

```
C_INTERCHANGE(A, LR:UR > R, LC:UC > C)
```

Rather than directly specifying the criteria for a local strategy that must be used to determine a row and column, the sparse compiler may select these criteria after analyzing the program. After a particular local strategy has been selected, the sparse compiler is also responsible for generating code that implements the selected local strategy. In this code, desired row and column interchanges are selected at run-time, applied and recorded. The implementation issues are further explored in chapter 10.

Induction Annotations

Permuting an implicitly sparse matrix A into PAQ may have mathematical consequences that have to be dealt with in the original dense program. For example, if A is permuted into PAQ before we compute $\vec{b} \leftarrow A\vec{x}$, then we must permute the original vector $\vec{x}$ into $\vec{x}\,' \leftarrow Q^T\vec{x}$ before the product, and permute the computed vector $\vec{b}\,'$ into the desired result $\vec{b} \leftarrow P^T\vec{b}\,'$ after the product has been computed. This is implied by the following formula, where the part $(*)$ is computed by an implementation that assumes that elements are physically moved in the enveloping data structure:

$$P\vec{b} = \underbrace{\vec{b}\,' = PAQ\vec{x}\,'}_{(*)} = PAQ(Q^T\vec{x}) = PA\vec{x}$$

Likewise, if before factorization or during LU-factorization to solve a system $A\vec{x} = \vec{b}$, the implicitly sparse matrix A becomes permuted into PAQ using an a priori or local strategy, then effectively the factorization $PAQ = LU$ is computed. Hence, before forward substitution is applied we must permute the right-hand side $\vec{x}$ into $\vec{x}\,' \leftarrow P\vec{x}$. After back substitution, the computed $\vec{x}\,'$ is permuted into the desired solution $\vec{x} \leftarrow Q\vec{x}\,'$. This is implied by the following formula, where $(*)$ is solved by an implementation assuming physical data movement:

$$PA\vec{x} = PAQQ^T\vec{x} = LU(Q^T\vec{x}) = \underbrace{LU\vec{x}\,' = \vec{b}\,'}_{(*)} = P\vec{b}$$

Induction annotations are used to deal with such consequences. The implementation of recording and applying permutation matrices are kept transparent to the programmer, i.e. *the compiler is responsible for implementing induction annotations*. However, *it is the responsibility of the programmer to correctly deal with all mathematical consequences of a permutation using induction annotations*, because it seems to be very hard to determine mathematical consequences automatically. Since incorrect use of induction annotations may affect the semantics of the original program, these annotations must be used with care.

In an induction annotation, the identifier of a one-dimensional array that must be permuted,[2] an action, and a row or column permutation matrix currently associated with an implicitly sparse matrix are specified as shown below:

```
sp_exec  →  C_INDUCE ID action matrix '(' ID ')'
         ;
action   →  '>'
         |  '<'
         ;
matrix   →  _ROW
         |  _COLUMN
         ;
```

The sparse compiler replaces each induction annotation with code that has the following impact on a column vector $\vec{x} = (x_1, \ldots, x_m)^T$ and a row vector $\vec{y} = (y_1, \ldots, y_n)$ stored in arrays X and Y respectively, where P and Q denote the row and column permutation matrices that at the time of execution are associated with the implicitly sparse matrix A with enveloping data structure A:

annotation:	result:	alternative result:
`C_INDUCE X < _ROW (A)`	$\vec{x} \leftarrow P\vec{x}$	$\vec{x}^T \leftarrow \vec{x}^T P^T$
`C_INDUCE Y < _COLUMN(A)`	$\vec{y} \leftarrow \vec{y}Q$	$\vec{y}^T \leftarrow Q^T\vec{y}^T$
`C_INDUCE X > _ROW (A)`	$\vec{x} \leftarrow P^T\vec{x}$	$\vec{x}^T \leftarrow \vec{x}^T P$
`C_INDUCE Y > _COLUMN(A)`	$\vec{y} \leftarrow \vec{y}Q^T$	$\vec{y}^T \leftarrow Q\vec{y}^T$

The alternative result arises from the fact that transposition has no impact on FORTRAN array representations (viz. X and Y interpreted as row and column vector). The method of recording permutation matrices and the actual implementation of the computations specified in induction annotations are kept transparent to the programmer. If the number of elements in the one-dimensional array and the order of the permutation matrix differ, a warning is generated and the annotation is ignored. If a dense data structure is specified, then the one-dimensional array remains unaffected (viz. $P = Q = I$ in this case).

Example: Below, we present an implementation of $\vec{b} \leftarrow A\vec{x}$, where the 100×50 implicitly sparse matrix A may be permuted arbitrarily in advance:

[2] This restriction is imposed to simplify the implementation in the prototype sparse compiler.

```
      PROGRAM PERM
      INTEGER   I, J, M, N
      PARAMETER (M=100, N=50)
      REAL      A(M,N), B(M), X(N)
C_SPARSE(A : _PERM)
      ...
C_INDUCE X < _COLUMN(A)
      DO I = 1, M
        B(I) = 0.0
        DO J = 1, N
          B(I) = B(I) + A(I,J) * X(J)
        ENDDO
      ENDDO
C_INDUCE B > _ROW(A)
      ...
      END
```

5.2 Subroutines and Functions

In the original dense program, two-dimensional arrays are used as enveloping data structures of implicitly sparse matrices. Because the sparse compiler eventually selects another data structure for each implicitly sparse matrix, problems arise if enveloping data structures are used as actual arguments in subroutine or function calls. The way in which the array is passed to and handled in the subroutine or function must be changed according to the selected data structure. In the following sections, we discuss which parameter passing mechanisms are allowed for implicitly sparse matrices. Furthermore, we discuss how these mechanisms are dealt with by the sparse compiler.

5.2.1 Parameter Passing Mechanisms

Assume that a subroutine P is called as follows:

```
CALL P(A1,...,An)
```

The expressions $\mathtt{A}_1, \ldots, \mathtt{A}_n$ are referred to as the **actual arguments**. The header of the subroutine definition introduces the **formal arguments** $\mathtt{F}_1, \ldots, \mathtt{F}_n$, which are further declared in the body of the subroutine:

```
SUBROUTINE P(F1,...,Fn)
...
END
```

For each CALL-statement, the number of actual arguments must be equal to the number of formal arguments of the called subroutine. Furthermore, the type of each actual and corresponding formal argument must be the same. During invocation of the subroutine, we say that each actual argument $\mathtt{A}_i$ is **associated with** the formal argument $\mathtt{F}_i$. The same terminology is used for function calls. Moreover, from now on, we use the generic term **procedure** to refer to a subroutine or function.

Single Element Arguments

A *single element* of an implicitly sparse matrix is passed to a procedure, if an arbitrary element of the corresponding enveloping data structure is associated with a *scalar* formal argument. For example, in the following fragment, array A is used as enveloping data structure of a 10×10 implicitly sparse matrix A and element a_{27} is passed to the subroutine USE:

```
      PROGRAM IN                SUBROUTINE USE(X)
      REAL A(10,10)             REAL X
C_SPARSE(A)                     ... = X
      CALL USE( A(2,7) )        RETURN
      END                       END
```

In this fragment, `A(2,7)` is associated with an input-argument, because the formal argument `X` cannot be modified in the subroutine. A similar situation is illustrated below:

```
X = ABS( A(I,J) )
```

In these situations, enveloping data structures used as actual arguments can simply be replaced by a lookup in the selected data structure, i.e. the value of the appropriate element is passed to the called procedure. In this manner, the subroutine or function remains unaware of the fact that elements of an enveloping data structure may be associated with its formal arguments. Although this implies that sparsity cannot be exploited in body of the called procedure (a similar problem was encountered at the end of section 4.3.2 for elements stored in temporary variables), the main advantage of this approach is that no further transformations are required in this body. The following CALL-statement, for instance, would result for the first example shown above if general sparse row-wise storage is selected for A (function '`LKP__`' is presented in chapter 8):

```
CALL USE( VAL_A( LKP__(IND_A, LOW_A(2), HGH_A(2), 7) ) )
```

It is also possible that arbitrary elements of enveloping data structures become associated with output- or input/output-arguments, as is illustrated below:

```
      PROGRAM OUT                SUBROUTINE DEF(X)
      REAL A(10,10)              REAL X
C_SPARSE(A)                      IF (....) X = ...
      CALL DEF( A(3,4) )         RETURN
      END                        END
```

In this case, the subroutine must become aware of the sparsity of the formal argument, and the whole data structure that will be selected for the implicitly sparse matrix must be accessible in the subroutine to account for the possibility of creation. The best way to handle these situations is to rewrite the CALL-statement into a form in which the whole implicitly sparse matrix and the subscripts are passed separately to the subroutine, as is illustrated below:

```
      PROGRAM OUT                SUBROUTINE DEF(M, I, J)
      REAL A(10,10)              REAL    M(10,10)
C_SPARSE(A)                      INTEGER I, J
      CALL DEF(A, 3, 4)          IF (...) M(I,J) = ...
      END                        RETURN
                                 END
```

As further explained in section 5.2.2, a clone '`DEF_A00`' will be generated having a unique association between the enveloping data structure `A` and the formal argument `M`. In this manner, all occurrences of `M` can simply be handled as occurrences of `A`. The data structure that is selected for the corresponding implicitly sparse matrix A will be made available to the clone using COMMON-storage.

Implicitly Sparse Matrix Arguments

A *whole* implicitly sparse matrix is passed as a parameter to a procedure if the first element of an enveloping data structure `A` becomes associated with a two-dimensional formal argument that has the same index set as `A`. In FORTRAN, this first element can be defined in an actual argument as either '`A`' or '`A(1,1)`'. In the following program, for instance, the whole implicitly sparse 10×10 matrix A is passed to the subroutine `GAUSS`:

```
     PROGRAM SOLVE               SUBROUTINE GAUSS(M)
     REAL A(10,10), D(10,10)     REAL M(10,10)
C_SPARSE(A)                      ...
     CALL GAUSS(A)               RETURN
     CALL GAUSS(D)               END
     END
```

Because a new data structure will be selected for the implicitly sparse matrix, the formal argument M must be converted accordingly. However, the subroutine is also called with the actual argument D, the data structure of which will remain unaffected. Consequently, we cannot apply the data structure transformation applied to array A directly to M, because this would make subroutine GAUSS unsuited for calls with other actual arguments. Likewise, if *different* implicitly sparse matrices are passed to the subroutine, application of data structure transformations would make the subroutine GAUSS unsuited for all implicitly sparse matrices for which a different data structure has been selected.

In-line expansion [5, 55][234, p101–102] could be used to resolve this problem, since this would enable the application of arbitrary data structure transformations to the separate call-sites afterwards. However, to prevent the inherent increase in code size caused by in-lining, the sparse compiler uses **procedure cloning** [54, 55] to construct clones (copies) of subroutines or functions in which enveloping data structures are uniquely associated with formal arguments. In the previous fragment, for instance, the subroutine is cloned into copies that will be used for calls having different enveloping data structures as actual argument (such as 'GAUSS_A' and 'GAUSS_B'). In each clone, the unique association between the appropriate enveloping data structure and the formal argument M is recorded. If necessary, the original subroutine GAUSS is preserved and used for all calls with an actual argument of which the data structure is not altered by the sparse compiler.

If a procedure has several formal arguments, then one clone is required for each possible association between enveloping data structures and formal arguments. For example, if a procedure P with two arguments is called as follows, then a clone 'P_AB' in which A and B are uniquely associated with the formal arguments as well as another clone 'P_A0' in which array A and an arbitrary dense data structure are uniquely associated with the formal arguments F and G:

```
      PROGRAM CLONE                    SUBROUTINE P(F, G)
      REAL A(10,10), B(10,10)          REAL F(10,10), G(10,10)
      REAL C(10,10), D(100)            ...
C_SPARSE(A ; B)                        ...
      CALL P(A, B)                     RETURN
      CALL P(A, C)                     END
      CALL P(A, D)
      END
```

If in a clone, an enveloping data structure is uniquely associated with a formal argument, then all occurrences of this formal argument are handled as occurrences of the enveloping data structure. The data structure that will be selected for the corresponding implicitly sparse matrix is made available to the clone using COMMON-storage, which avoids passing an excessive number of parameters in case many variables are required to implement the new data structure. Hence, all arguments that are used to pass a whole implicitly sparse matrix to a clone become redundant and are eliminated. Note that because sparsity information can be propagated into clones, situations may again arise in which implicitly sparse matrices are passed to subroutines or functions that are called within the clones. Hence, propagated cloning may be required, as further discussed in section 5.2.2.

Remaining Arguments

Finally, problems may arise if an enveloping data structure is associated with a formal argument that has a different shape (which is allowed in FORTRAN). The following user-defined function SUM, for instance, linearizes a two-dimensional array A that is used as enveloping data structure into a one-dimensional array:

```
      PROGRAM LINEAR                REAL FUNCTION SUM(M)
      REAL A(10,10), SUM            REAL    M(100)
C_SPARSE(A)                         INTEGER I
      ...
      PRINT *, SUM(A)               SUM = 0.0
      ...                           DO I = 1, 100
      END                             SUM = SUM + M(I)
                                    ENDDO
                                    RETURN
                                    END
```

A similar kind of reshaping occurs if the enveloping data structure becomes associated with a formal argument of higher dimension (e.g. '`REAL M(2,25,2)`'). To simplify subsequent data structure transformations and to prevent the requirement to translate access information according to the ways arrays are reshaped in subroutine or function calls (cf. translations performed during interprocedural analysis [15, 47, 48, 109, 142]), an enveloping data structure may only be associated with a formal argument that has exactly the same index set. All other associations must be resolved explicitly by the programmer before the automatic dense to sparse conversion can be performed.

Valid Associations

Summarizing, the following parameter passing mechanisms are allowed:

- An element of an enveloping data structure is associated with a scalar *input*-argument.
- The first element of an enveloping data structure is associated with a two-dimensional formal argument that has exactly the same index set as the enveloping data structure.

These associations, and any association between variables of which the data structure remains unaffected and arbitrary formal arguments are referred to as **valid associations**. All other associations are invalid. In the prototype sparse compiler, we require that potential invalid associations are resolved by the programmer, as alluded to in the previous sections.
Example: In the following program, associating `A`, `B`, and `C` with the formal argument `M` is valid. However, because a clone will be generated for subroutine `P` in which the enveloping data structure `A` is uniquely associated with `M`, associating `M` as actual argument with the formal argument `L` in the call to subroutine `Q` is invalid:

```
      PROGRAM MAIN                  SUBROUTINE P(M)
      REAL A(10,10)                 REAL M(10,10)
      REAL B(100), C(10,10)         IF (...) CALL Q(M)
C_SPARSE(A)                         RETURN
      CALL P(A)                     END
      CALL P(B)
      CALL P(C)                     SUBROUTINE Q(L)
      END                           REAL L(20,10)
                                    ...
                                    RETURN
                                    END
```

In the next section, we present an algorithm to compute all required procedure clones in a program and to detect potential invalid associations. These problems are handled as flow insensitive problems (or MAY-problems). For the previous example, this implies that even if the IF-statement is used to ensure that the call to `Q` in subroutine `P` will never occur while `A` is associated with `M`, the algorithm records the requirement of a clone `Q_A`, and reports a potential invalid association. Only after the declaration '`REAL L(20,10)`' has been rewritten into '`REAL L(10,10)`', all associations become valid.

5.2.2 Procedure Cloning

In the absence of procedure parameters, construction of a call graph is straightforward [183, 229, 234]: a vertex labeled P is created for each procedure P in a program, and an edge from P to Q is added to the call graph if procedure P may call Q. Dynamic recursion, where at run-time several activations of the same procedure can exist simultaneously [3], is not allowed in FORTRAN. Moreover, we disallow static recursion, where two procedures can (in)directly call each other but two activations of the same procedure will never exist simultaneously. Consequently, the resulting call graph is acyclic. Finally, because the prototype sparse compiler operates on whole programs rather than separate source files, we assume that the complete call graph is available.

Computing the Required Procedure Clones

All clones required to enforce a unique association between enveloping data structures and formal arguments can be determined during a visit of all procedures according to a topological sort of the call graph. Because this implies that for each edge from P to Q in the call graph, procedure P is visited before Q, all possible associations between enveloping data structures and formal arguments are known for all predecessors of a visited vertex. Below, we present a variant of the algorithm given in [55] to compute the required clones, that also strongly resembles the call graph construction method accounting for procedure parameters in [183].

For each procedure P in a program, a table $\mathtt{T_P}$ is created having a column for each formal argument. We can express the algorithm in a more regular manner, if the main program is also handled as a procedure. All local variables of the main program are viewed as formal arguments, denoted by $\mathtt{L}_1, \ldots, \mathtt{L}_n$. We also construct a table for the main program, having a column for each local variable. Because *declaration* annotations are only allowed in the main program, this unit is the only source of sparsity information. The table of the main program is initialized by inserting a single row $[\tau_1, \ldots, \tau_n]$, where $\tau_i = \mathtt{L}_i$, if the 'formal' argument $\mathtt{L}_i$ is used as an enveloping data structure of an implicitly sparse matrix and $\tau_i = \mathtt{0}$, otherwise.

Subsequently, rows are inserted in the other tables by executing the following procedure for all vertices in the call graph according to a topological sort.

```
procedure visit(P)
begin

  for each procedure call 'Q(A1,...,An)' in P do

    for each row r in TP do
      insert(TQ, [τ1(r),...,τn(r)] )
    enddo

  enddo

end
```

Each expression $\tau_i(r)$, where $1 \leq i \leq n$ and n is the number of formal arguments of Q, is evaluated as follows:

$$\tau_i(r) = \begin{cases} \mathtt{T_P}[r][j] & \textbf{if } \mathtt{A}_i \text{ is } j\text{th formal argument of } \mathtt{P} \\ \mathtt{0} & \textbf{otherwise} \end{cases}$$

If the jth formal argument of P is equal to $\mathtt{A}_i$, then $\tau_i(r)$ contains either the identifier of an enveloping data structure that may be associated with this variable, or the value '0' if an unaffected data structure may be associated with this argument. We also have $\tau_i(r) = \mathtt{0}$ in all other cases, i.e. if $\mathtt{A}_i$ is a local variable or an arbitrary expression.

If $\tau_i(r) \neq$ `0`, then the enveloping data structure defined by $\tau_i(r)$ may be associated with the ith formal argument of procedure `Q`, which is denoted by $\texttt{F}_i$.

A potential invalid association is reported in one of the following cases:

1. Formal $\texttt{F}_i$ is a scalar and occurs in the $\text{MOD}_{\texttt{Q}}$-set.
2. Formal $\texttt{F}_i$ is a two-dimensional array and:[3]
   (a) Actual $\texttt{A}_i$ does not define the first element, or
   (b) The index sets of formal argument $\texttt{F}_i$ and enveloping data structure $\tau_i(r)$ differ.
3. Formal $\texttt{F}_i$ is neither a scalar nor a two-dimensional array.

Thereafter, the procedure `insert` is called. In this procedure, we test whether the value of the second argument is present in the table defined by the first argument. If this test fails, a row with this value is added to the table.

After all vertices in the call graph have been visited, each row in a table $\texttt{T}_\texttt{P}$ indicates a potential association between the enveloping data structures of implicitly sparse matrices and the formal arguments of procedure `P`. The value '`0`' indicates that an arbitrary actual argument remaining unaltered by the sparse compiler is associated with the corresponding formal argument. Consequently, if '$[\texttt{0},\ldots,\texttt{0}]$' appears in the table, the original procedure must be preserved. For all other rows, one clone must be generated in which a particular association between enveloping data structures and formal arguments holds. A name is constructed for each clone by concatenating the original name with an underscore and the strings defined by each row ('`P_0A`', '`P_BA`' etc.). The use of an underscore prevents name conflicts with identifiers in the original program.

Enforcing a unique association between enveloping data structures and formal arguments enables the application of different program and data structure transformations to each generated clone without interfering with the other clones that support different kinds of associations for the original procedure. The presented algorithm is more accurate than a naive algorithm that would simply determine possible associations between enveloping data structures and formal arguments, after which clones are generated for *all* possible combinations of these associations.

Adapting CALL-Statements

After all clones have been generated, each procedure call in the program in which enveloping data structures appear as actual arguments are replaced by a call to the appropriate clone. Each actual and corresponding formal argument used to pass a whole implicitly sparse matrix are eliminated from the program after the unique association has been recorded in the clone. Arguments used to pass a single element of an implicitly sparse matrix, on the other hand, are preserved.

If the program is examined, the unique association of an enveloping data structure `A` with a formal argument `F` that has been eliminated from the header of the procedure is indicated by prompting '`<- Sparse(A)`' after the declaration of this formal argument. Moreover, the same construct is used in the main program to indicate which arrays are used as enveloping data structures.
Example: For the following program, the clone '`P_A0`' is generated in which the enveloping data structure `A` is uniquely associated with the formal argument `F`. The CALL-statement in the main program is replaced by a call to this clone. Moreover, because the first argument is used to pass the whole implicitly sparse matrix, the first actual and formal argument are eliminated:

[3] If these tests cannot be performed because formal arguments occur in either $\texttt{A}_i$ or the declaration of $\texttt{F}_i$, we postpone this test until cloning and interprocedural constant propagation have been applied.

```
      PROGRAM CLONE                    PROGRAM CLONE
      REAL    A(10,10)                 REAL A(10,10) <- Sparse(A)
C_SPARSE(A)                            CALL P_A0( A(1,2) )
      CALL P(A, A(1,2))                END
      END                              SUBROUTINE P_A0(X)
      SUBROUTINE P(F, X)     →         REAL F(10,10) <- Sparse(A)
      REAL F(10,10)                    REAL X
      REAL X                           ...
      ...                              RETURN
      RETURN                           END
      END
```

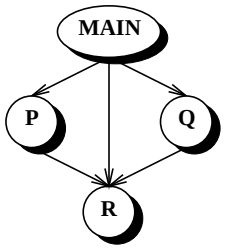
If, after application of cloning, an enveloping data structure A is uniquely associated with a formal argument F, then all occurrences of F are handled as occurrences of the enveloping data structure A and the selected data structure is made available to the clone using a named COMMON-block, as is illustrated below (details are given in chapter 8):

```
PROGRAM CLONE                          SUBROUTINE P_A0(X)
... declarations ...                   ... same declarations ...
COMMON /A/ ... identifiers ...         COMMON /A/ ... identifiers ...
                                       REAL X
CALL P_A0( ... )                       ...
                                       RETURN
END                                    END
```

Examples of Procedure Cloning

Example: In the following program, the arrays A and B are used as the enveloping data structures of implicitly sparse matrices A and B:

```
      PROGRAM MAIN
      REAL X(25),  Y(40)
      REAL A(5,5), B(5,8)
C_SPARSE(A ; B)
      CALL P(A, B)
      CALL Q(A, B, X)
      CALL Q(X, Y, X)
      CALL P(X, B)
      CALL R(Y, A)
      END
```



The following subroutines P, Q, and R are used:

```
SUBROUTINE P(F, G)      SUBROUTINE Q(F, G, H)      SUBROUTINE R(F, G)
REAL F(5,5)             REAL F(5,5)                REAL F(5,8)
REAL G(5,8)             REAL G(5,8)                REAL G(5,5)
REAL H(5,5)             REAL H(5,5)                ...
CALL R(G, F)            CALL R(G, F)               RETURN
CALL R(H, F)            RETURN                     END
RETURN                  END
END
```

Because the main program calls all subroutines, and the subroutines P and Q call subroutine R, the call graph has the form shown above. During initialization, the tables T_{MAIN}, T_P, T_Q, and T_R are constructed. Because arrays A and B are used as enveloping data structure of two implicitly sparse matrices, the table for the main program is initialized as shown below:

MAIN	X	Y	A	B
1	0	0	A	B

All procedures are visited according to a topological sort of the call graph, e.g. in the order MAIN, P, Q, and R. During the visit to the main program, all calls to P, Q and R are considered.

This results in adding the appropriate rows to the tables of the subroutines. For example, if the first call '`CALL P(A,B)`' is considered, we see that associating `A` and `B` with respectively `F` and `G` is valid. Therefore, we insert '`[A,B]`' in T_P. After all calls in the main program are considered, the tables have the following contents:

P	F	G
1	A	B
2	0	B

Q	F	G	H
1	A	B	0
2	0	0	0

R	F	G
1	0	A

If, subsequently, the call '`CALL R(G,F)`' in subroutine `P` is considered, then rows '`[B,A]`' and '`[B,0]`' are added to table T_R. Consideration of the second call to `R` yields the value '`[0,A]`', already present in the table, and the new row '`[0,0]`'.

Visiting subroutine `Q` does not result in the insertion of more rows (viz. '`[B,A]`' and '`[0,0]`' are already present in the table T_R):

R	F	G
1	0	A
2	B	A
3	B	0
4	0	0

The last row in this table indicates that the original subroutine `R` must be preserved. The other rows give rise to the following clones:

```
SUBROUTINE R_0A(F)          SUBROUTINE R_BA()           SUBROUTINE R_B0(G)
REAL F(5,8)                 REAL F(5,8) <- Sparse(B)    REAL F(5,8) <- Sparse(B)
REAL G(5,5) <- Sparse(A)    REAL G(5,5) <- Sparse(A)    REAL G(5,5)
...                         ...                         ...
RETURN                      RETURN                      RETURN
END                         END                         END
```

The tables T_P and T_Q give rise to the following clones, in which calls to the appropriate clones of `R` are used:

```
SUBROUTINE P_AB(F, G)       SUBROUTINE P_0B(F, G)       SUBROUTINE Q_AB0(H)
REAL F(5,5) <- Sparse(A)    REAL F(5,5)                 REAL F(5,5) <- Sparse(A)
REAL G(5,8) <- Sparse(B)    REAL G(5,8) <- Sparse(B)    REAL G(5,8) <- Sparse(B)
REAL H(5,5)                 REAL H(5,5)                 REAL H(5,5)
CALL R_BA()                 CALL R_B0(F)                CALL R_BA()
CALL R_0A(H)                CALL R   (H, F)             RETURN
RETURN                      RETURN                      END
END                         END
```

In addition, the original subroutine `Q` is also preserved. Finally, all calls in the main program are replaced by the following CALL-statements:

```
PROGRAM MAIN
...
CALL P_AB ()
CALL Q_AB0(X)
CALL Q    (X, Y, X)
CALL P_0B (X)
CALL R_0A (Y)
END
```

Note that in a naive approach in which all possibly associations are simply combined, the clones '`Q_A00`' and '`Q_0B0`' would also be generated.

Example: Subroutines and functions can be made more general if scalar formal arguments are used in the declarations of non-scalar formal arguments:

```
      PROGRAM MAIN
      INTEGER K
      PARAMETER (K = 10)
      REAL A(2*K,2*K)
      REAL B(3*K,3*K)
      REAL C(4*K,4*K), S
C_SPARSE(A)

      CALL SCAL(A, S, 2 * K)
      CALL SCAL(B, S, 3 * K)
      CALL SCAL(C, S, 4 * K)
      ...
      END
```

```
SUBROUTINE SCAL(F, S, N)
INTEGER N, I, J
REAL    F(N,N), S
DO I = 1, N
  DO J = 1, N
    F(I,J) = F(I,J) * S
  ENDDO
ENDDO
RETURN
END
```

In the original program, we cannot determine whether all associations are valid, since the value of N cannot be computed by interprocedural constant propagation (the formal argument N can have the values 20, 30 and 40). Therefore, test 2(b) is postponed until after procedure cloning. The following tables result:

MAIN	K	A	B	C	S
1	0	A	0	0	0

SCAL	F	S	N
1	A	0	0
2	0	0	0

After the clone 'SCAL_A00' has been generated, interprocedural constant propagation indicates that the equation N=20 holds in this clone (note that the value of N in the *original* subroutine remains unknown). Hence, the postponed test 2(b) succeeds and all associations are valid. If the following CALL-statement would also appear in the previous program, no additional clone would be generated because the second actual argument is associated with a scalar input-argument:

```
CALL SCAL(A, A(1,1), 20)
```

If the value of the formal argument S could be modified within the subroutine SCAL, however, the previous association would be invalid.

Example: For the following program, test 2(a) must be postponed in the subroutine P, because the value of I cannot be determined using interprocedural constant propagation:

```
      PROGRAM MAIN
      REAL A(10,10), B(10,10)
C_SPARSE(A ; B)
      ...
      READ *, K
      CALL P(A, 1)
      CALL P(B, 1)
      CALL P(B, K)
      ...
      END
```

```
SUBROUTINE P(F, I)
INTEGER I
REAL    F(10,10)
CALL Q(F(I,I))
RETURN
END

SUBROUTINE Q(G)
REAL G(10,10)
...
RETURN
END
```

After cloning and interprocedural constant propagation, the equation I=1 holds within the clone 'P_A0'. Hence, the postponed test 2(a) succeeds and the association with formal argument G is valid. However, this test still cannot be performed within the clone 'P_B0', and a potential invalid association is reported.

In general, if a potential invalid association is detected before procedure cloning, then the original identifier is used in the error message. If a postponed test fails or still cannot be performed after procedure cloning, then the identifier of the clone is used in this message.

5.3 Conditions

In chapter 4, the observation was made that not all instances of a particular statement have to be executed. For example, all instances of the following assignment statement in which the scalar ACC is updated with a zero can be skipped without affecting the semantics of the program:

```
ACC = ACC + A(I,J) * X(J)
```

Hence, condition '$\texttt{A}(\texttt{I},\texttt{J}) \neq 0 \wedge \texttt{X}(\texttt{J}) \neq 0$' could be associated with this statement to indicate the instances that have to be executed. Obviously, a reasonable goal in computing such conditions would be to associate the *strongest* condition with each statement, since this would offer the most potential for execution time reduction. However, not all tests on the value of expressions can be effectively exploited to reduce execution time. Therefore, in this section we focus on conditions arising from accesses to non-entries or zero constants. For example, if in the previous statement array A is used as the enveloping data structure of an implicitly sparse $m \times n$ matrix A, then the condition '$(\texttt{I},\texttt{J}) \in E(A)$' is associated with this statement, where $E(A) \subseteq [1,m] \times [1,n]$ denotes the index set of the entries of the matrix. Hereby, we conservatively ignore the possibility that some entries in A or components of the vector $\vec{x}$ can also have the value zero. In general, such conditions enable the effective exploitation of sparsity (e.g. by means of guard encapsulation), whereas arbitrary tests do not.

5.3.1 Associating Conditions with Statements

Conditions are associated with statements by evaluating the semantic rules in an attributed grammar based on a simple context free grammar for FORTRAN statements.[4] The ambiguity of the grammar used in the following sections is resolved by assigning the usual precedence and associativity to all operators. Furthermore, we assume that the compiler can distinguish between expressions derived from the following two productions for variables and functions calls respectively:

Production:	Examples:
E → var	`X, Y(I), A(I,J)`
E → **id**(arg_list)	`ABS(X), MAX(10,Y), SQRT(2.0)`

Each production in the grammar has a set of semantic rules associated with it that define the value of attributes belonging to the grammar symbols. Some of these values are formed by conditions that eventually will be associated with statements. Conditions may consist of the value '**true**' or '**false**', guards, and conjunctions or disjunctions of conditions. Because such conditions can only be evaluated at run-time, an internal representation to store the value of attributes is needed. Although this induces a subtle difference between conditions occurring as values of attributes, and proper boolean expressions, for convenience sake we use semantics rules like the one shown below to indicate that the internal representation of the value '**true**' must be assigned to the attribute E.p associated with the grammar symbol E if the test succeeds, and the representation of the value '**false**' otherwise:

E.p = (const.val $>$ 0);

Attributes for Numerical Expressions

To exploit sparsity, it is essential to know whether a numerical expression E is zero (due to accessing a non-entry or a zero constant) or not. Moreover, later on we will see that it is also useful to know the sign of numerical expressions.

[4] The author would like to acknowledge very helpful discussions with Arnold Niessen which have contributed substantially to improving the attributed grammar.

For this purpose, three synthesized attributes E.p, E.z, and E.n are associated with each numerical expression E, recording the condition under which the expression *may* be strictly positive, zero, and strictly negative, respectively.

Semantic rules for all numerical expressions formed using the arithmetic operators '$+$' and '$-$' are derived by examination of the following tables in which, given particular assertions on the value of the operands, similar assertions on the value of the resulting expression are shown:

$E_1 + E_2$	$E_2 > 0$	$E_2 = 0$	$E_2 < 0$
$E_1 > 0$	> 0	> 0	?
$E_1 = 0$	> 0	$= 0$	< 0
$E_1 < 0$	?	< 0	< 0

$E_1 - E_2$	$E_2 > 0$	$E_2 = 0$	$E_2 < 0$
$E_1 > 0$	?	> 0	> 0
$E_1 = 0$	< 0	$= 0$	> 0
$E_1 < 0$	< 0	< 0	?

For example, because expression E='$E_1 + E_2$' may be strictly positive if one of the operands may be strictly positive, we define E.p as 'E_1.p $\vee$ E_2.p'. Likewise, expression E='$E_1 - E_2$' may be zero if both operands can be zero or if the operands have identical signs. Hence, in this case E.z is defined as '(E_1.p $\wedge$ E_2.p) $\vee$ (E_1.z $\wedge$ E_2.z) $\vee$ (E_1.n $\wedge$ E_2.n)'. Continuing in this fashion yields the following semantic rules:

Production	Semantic Rule
$E \rightarrow E_1 + E_2$	E.p = E_1.p $\vee$ E_2.p; E.z = (E_1.p $\wedge$ E_2.n) $\vee$ (E_1.z $\wedge$ E_2.z) $\vee$ (E_1.n $\wedge$ E_2.p); E.n = E_1.n $\vee$ E_2.n;
$E \rightarrow E_1 - E_2$	E.p = E_1.p $\vee$ E_2.n; E.z = (E_1.p $\wedge$ E_2.p) $\vee$ (E_1.z $\wedge$ E_2.z) $\vee$ (E_1.n $\wedge$ E_2.n); E.n = E_1.n $\vee$ E_2.p;

A similar table can be given for the unary minus, whereas the tables for the arithmetic operators '$*$', '$/$', and '$**$' are shown below:

$E_1 * E_2$	$E_2 > 0$	$E_2 = 0$	$E_2 < 0$
$E_1 > 0$	> 0	$= 0$	< 0
$E_1 = 0$	$= 0$	$= 0$	$= 0$
$E_1 < 0$	< 0	$= 0$	> 0

E_1 / E_2	$E_2 > 0$	$E_2 = 0$	$E_2 < 0$
$E_1 > 0$	> 0	$\perp$	< 0
$E_1 = 0$	$= 0$	$\perp$	$= 0$
$E_1 < 0$	< 0	$\perp$	> 0

$E_1 ** E_2$	$E_2 > 0$	$E_2 = 0$	$E_2 < 0$
$E_1 > 0$	> 0	> 0	> 0
$E_1 = 0$	$= 0$	$\perp$	$= 0$
$E_1 < 0$	$\neq 0$	> 0	$\neq 0$

A subtlety arises for the operators '$/$' and '$**$', because the values of 0^0 and an expression in which a divisor occurs that is zero are undefined. However, under the assumption that such situations (and underflow) do not occur at run-time, 'convenient' assertions may be placed at the positions of the '$\perp$', which gives rise to the following semantic rules:

Production	Semantic Rule
$E \rightarrow E_1 * E_2$	E.p = (E_1.p $\wedge$ E_2.p) $\vee$ (E_1.n $\wedge$ E_2.n); E.z = E_1.z $\vee$ E_2.z; E.n = (E_1.p $\wedge$ E_2.n) $\vee$ (E_1.n $\wedge$ E_2.p);
$E \rightarrow E_1 / E_2$	E.p = (E_1.p $\wedge$ E_2.p) $\vee$ (E_1.n $\wedge$ E_2.n); E.z = E_1.z; E.n = (E_1.p $\wedge$ E_2.n) $\vee$ (E_1.n $\wedge$ E_2.p);

Production	Semantic Rule
$E \rightarrow E_1 ** E_2$	E.p = E_1.p $\vee$ E_1.n; E.z = E_1.z; E.n = (E_1.n $\wedge$ E_2.p) $\vee$ (E_1.n $\wedge$ E_2.n);
$E \rightarrow - E_1$	E.p = E_1.n; E.z = E_1.z; E.n = E_1.p;

For variables, a guard is supplied in a synthesized attribute var.grd. For an occurrence of an enveloping data structure of an implicitly sparse matrix A with admissible subscripts $F(\vec{\mathtt{I}})$, this guard is '$F(\vec{\mathtt{I}}) \in E(A)$' (where the unique association between enveloping data structures and formal arguments enforced by cloning is accounted for). For all other variables, the value '**true**' is supplied. Likewise, the value of each constant is supplied in an attribute E.val. This enables us to define the following semantic rules:

Production	Semantic Rule
$E \rightarrow$ var	E.p = var.grd; E.z = **true**; E.n = var.grd;

Production	Semantic Rule
$E \rightarrow$ const	E.p = (const.val > 0); E.z = (const.val $= 0$); E.n = (const.val < 0);

Production	Semantic Rule
$E \rightarrow (E_1)$	E.p = E_1.p; E.z = E_1.z; E.n = E_1.n;

Semantic rules for all remaining numerical expressions are given in the following table:

Production	Semantic Rule
E $\rightarrow$ **id** (E_1, . . ., E_n)	E.p = **id**.nm $\in F$? (E_1.p $\vee$ E_1.n) : **true**; E.z = **id**.nm $\in G$? E_1.z : **true**; E.n = **id**.nm $\in H$? **false** : (**id**.nm $\in F$? (E_1.p $\vee$ E_1.n) : **true**);
/* others */	E.p = **true**; E.z = **true**; E.n = **true**;

If no further information is available, we conservatively assume that all values are possible (i.e. we set E.p=E.z=E.n=**true**). For function calls, however, some tests on the synthesized attribute **id**.nm containing the lexeme belonging to **id** are performed first. The set F consists of one-argument zero preserving functions (i.e. $f(0) = 0$):

$$F = \{\texttt{INT}, \texttt{REAL}, \texttt{ABS}, \texttt{SQRT}, \texttt{SIN}, \ldots\}$$

Obviously, for such functions, the resulting value may be strictly positive if the single argument can be nonzero. Likewise, the result of a one-argument nonzero preserving function (i.e. $f(x) \neq 0$ if $x \neq 0$) may be zero if the argument can be zero. The set G consists of such functions (ignoring inexact arithmetic):

$$G = \{\texttt{REAL}, \texttt{ABS}, \texttt{SQRT}, \ldots\}$$

Finally, for functions in F, the resulting value may be strictly negative if the argument can be nonzero, provided that we first account for the fact that some functions always have positive results. Therefore, we first test inclusion in the set H, consisting of all functions having positive results:

$$H = \{\texttt{ABS}, \texttt{DIM}, \texttt{SQRT}, \ldots\}$$

Advanced compile-time analysis techniques could be used to add user-defined functions with such properties to these sets.

Example: Some expressions and the conditions stored in the associated attributes are given below, where we assume that the arrays A and B are used as the enveloping data structures of implicitly sparse matrices A and B respectively, and that I, J, and K are loop-indices:

E	E.p	E.z	E.n	
`- 6.0 - SQRT(X)`	**false**	**false**	**true**	< 0
`ABS(0.0) * (X+1)`	**false**	**true**	**false**	$= 0$
`- SQRT(X)`	**false**	**true**	**true**	≤ 0
`ABS(50) + 10 * ABS(X)`	**true**	**false**	**false**	> 0
`-3.0 ** I`	**true**	**false**	**true**	≥ 0
`- SQRT(X) / (-5.0)`	**true**	**true**	**false**	$\neq 0$
`X + 50.0`	**true**	**true**	**true**	?
`10 - SQRT( A(I,J) )`	**true**	$(\texttt{I},\texttt{J}) \in E(A)$	$(\texttt{I},\texttt{J}) \in E(A)$	
`-1.0 * ABS( A(I,J) * B(K,K) )`	**false**	**true**	$(\texttt{I},\texttt{J}) \in E(A) \wedge (\texttt{K},\texttt{K}) \in E(B)$	
`A(I,J) * 2.0 + A(I,J) * B(I,J)`	$(\texttt{I},\texttt{J}) \in E(A)$	**true**	$(\texttt{I},\texttt{J}) \in E(A)$	

During evaluation of the semantic rules, some simplifications of conditions may be performed, such as $\psi \wedge \mathbf{true} \equiv \psi$, $\psi \wedge \mathbf{false} \equiv \mathbf{false}$, $\psi \wedge \psi \equiv \psi \vee \psi \equiv \psi$, $\psi \vee (\psi \wedge \chi) \equiv \psi$, and $\psi \wedge (\psi \vee \chi) \equiv \psi$. For instance, the value of E.p for the last expression shown above is in fact a simplification of the following condition:

$$\begin{array}{lll} (& ((\texttt{I},\texttt{J}) \in E(A) \wedge \mathbf{true}) \vee ((\texttt{I},\texttt{J}) \in E(A) \wedge \mathbf{false}) &) \quad \vee \\ (& ((\texttt{I},\texttt{J}) \in E(A) \wedge (\texttt{I},\texttt{J}) \in E(B)) \vee ((\texttt{I},\texttt{J}) \in E(A) \wedge (\texttt{I},\texttt{J}) \in E(B)) &) \end{array}$$

Although such simplifications are useful to reduce the amount of information prompted to the programmer, no attempts are made to fully simplify each condition because conditions are only used to determine dominating guards.

Conditions associated with Assignment Statements

If an instance of an assignment statement updates an arbitrary variable with an expression that is certainly zero, or assigns a certain zero to a non-entry, then we may skip execution of this instance.

To support the identification of update statements that can exploit sparsity, a pointer to the left-hand side variable, supplied in a synthesized attribute var.nm, is copied into an inherited attribute E.lhs of the right-hand side expression:

Production	Semantic Rule
stmt → var = E	E.lhs = var.nm;

Subsequently, this pointer is passed down the parse tree by simple copy rules:

Production	Semantic Rule
$E \rightarrow E_1 + E_2$	E_1.lhs = E_2.lhs = E.lhs;
$E \rightarrow E_1 - E_2$	E_1.lhs = E_2.lhs = E.lhs;
$E \rightarrow E_1 * E_2$	E_1.lhs = E_2.lhs = E.lhs;
$E \rightarrow E_1 / E_2$	E_1.lhs = E_2.lhs = E.lhs;
$E \rightarrow E_1 ** E_2$	E_1.lhs = E_2.lhs = E.lhs;
$E \rightarrow - E_1$	E_1.lhs = E.lhs;
$E \rightarrow (E_1)$	E_1.lhs = E.lhs;

The attributes E.p, E.n, and E.lhs are used to associate another synthesized attribute E.ne with each numerical expression. This attribute indicates the condition under which the right-hand side expression in an assignment statement *may* be different from the left-hand side variable. This implies that the condition E.ne is only false if the expression is certainly equal to the left-hand side.

For a single variable, the value of this attribute is simply determined by whether this variable is equal to the left-hand side variable or not. Similarly, an expression with an addition or subtraction is equal to the left-hand side variable if it is certain that one of the positive operands is identical to this variable and the other operand is zero. For all other expressions, we assume that this situation cannot occur:[5]

Production	Semantic Rule
$E \rightarrow$ var	E.ne = (var.nm $\neq$ E.lhs);
$E \rightarrow E_1 + E_2$	E.ne = (E_1.ne $\vee$ E_2.p $\vee$ E_2.n) $\wedge$ (E_1.p $\vee$ E_1.n $\vee$ E_2.ne);
$E \rightarrow E_1 - E_2$	E.ne = E_1.ne $\vee$ E_2.p $\vee$ E_2.n;
$E \rightarrow (E_1)$	E.ne = E_1.ne;
/* others */	E.ne = **true**;

Finally, conditions can be associated with assignment statements. An instance of an assignment statement must be executed, if the left- and right-hand side expression may differ and at least one of these expressions may be nonzero, provided that the statement does not evaluate functions with possible side-effects (see section 5.3.3):

Production	Semantic Rule
stmt → var = E	stmt.cnd = E.ne $\wedge$ (var.grd $\vee$ E.p $\vee$ E.n);

Example: Below, some assignment statements and associated conditions are shown. By construction, only *positive* guards appear within conditions:

stmt	stmt.cnd
`A(I,J) = X * 5.0`	**true**
`A(I,J) = A(I,J)`	**false**
`ACC = ACC + A(I,J) * X`	$(\mathtt{I},\mathtt{J}) \in E(A)$
`A(J,I) = ABS(A(J,I)) / A(I,I)`	$(\mathtt{J},\mathtt{I}) \in E(A)$
`A(I,J) = B(I,K) * B(K,J) + A(I,J)`	$(\mathtt{I},\mathtt{K}) \in E(B) \wedge (\mathtt{K},\mathtt{J}) \in E(B)$
`A(I,J) = A(I,J) * B(1,1) + B(I,J)`	$(\mathtt{I},\mathtt{J}) \in E(A) \vee (\mathtt{I},\mathtt{J}) \in E(B)$

[5] A shortcoming of these rules is that E.ne becomes '**true**' for the right-hand side in '`ACC=A(I,J)-(-ACC)`', although the stronger condition '$(\mathtt{I},\mathtt{J}) \in E(A)$' would be possible.

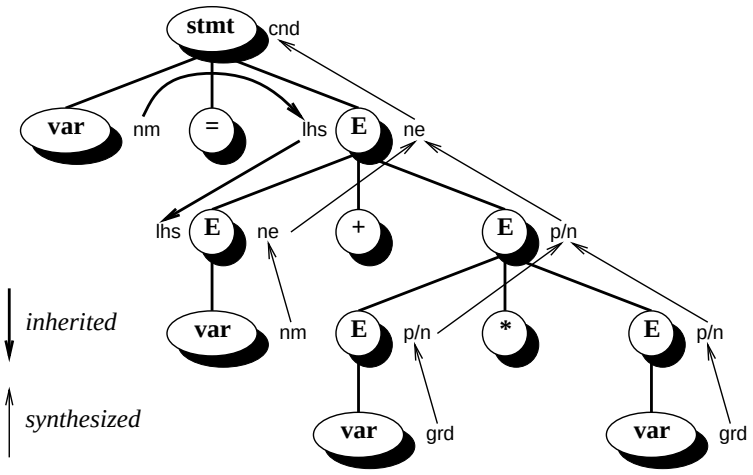


Figure 5.2: Parse Tree of Statement 'ACC=ACC+A(I,J)*X'

The evaluation of the condition associated with the third statement is depicted in figure 5.2. Condition '**false**' is associated with the second statement, because the right-hand side is always equal to the left-hand side. Such statements may be eliminated completely from the program without affecting the semantics. Associating the condition '$(\mathtt{J},\mathtt{I}) \in E(A)$' with the fourth statement is only valid under the assumption that a division by zero does not occur. If in the original code a non-entry would become undefined because it is divided by zero, this non-entry would erroneously remain zero by skipping instances for which the associated condition does not hold.

Attributes for Boolean Expressions

Because conditional statements are under control of boolean expressions, information about the possible values of a boolean expression can be used to determine which instances of conditional statement must be executed. One way to encode this kind of information is to associate two synthesized attributes E.t and E.f with each boolean expression E. These attributes record the condition under which E *may* evaluate to true or false respectively.

Because expression E='E_1 **and** E_2' may hold if it is possible that both operands hold, we define the value of attribute E.t as $E_1.t \wedge E_2.t$. Other semantic rules are obtained in a similar manner:

Production	Semantic Rule	
$E \rightarrow E_1$ **and** E_2	$E.t = E_1.t \wedge E_2.t$;	$E.f = E_1.f \vee E_2.f$;
$E \rightarrow$ **not** E_1	$E.t = E_1.f$;	$E.f = E_1.t$;

The following logical equivalences provide a convenient method to derive the semantic rules for all remaining logical operators:

$$\begin{aligned} \alpha \vee \beta &\equiv \neg(\neg\alpha \wedge \neg\beta) \\ \alpha \ \mathbf{eqv}\ \beta &\equiv (\alpha \wedge \beta) \vee (\neg\alpha \wedge \neg\beta) \\ \alpha \ \mathbf{neqv}\ \beta &\equiv \neg(\alpha \ \mathbf{eqv}\ \beta) \end{aligned}$$

The resulting semantic rules are shown below:

Production	Semantic Rule
$E \rightarrow E_1$ **or** E_2	$E.t = E_1.t \vee E_2.t$; $E.f = E_1.f \wedge E_2.f$;
$E \rightarrow E_1$ **eqv** E_2	$E.t = (E_1.t \wedge E_2.t) \vee (E_1.f \wedge E_2.f)$; $E.f = (E_1.f \vee E_2.f) \wedge (E_1.t \vee E_2.t)$;
$E \rightarrow E_1$ **neqv** E_2	$E.t = (E_1.f \vee E_2.f) \wedge (E_1.t \vee E_2.t)$; $E.f = (E_1.t \wedge E_2.t) \vee (E_1.f \wedge E_2.f)$;

For boolean expression obtained by applying relational operators to numerical operands, it is important to have more knowledge about the possible values of the expression. The tables shown below are used to derive the semantic rules for the operators '$=$', '$<$', and '$>$':

$E_1 = E_2$	$E_2 > 0$	$E_2 = 0$	$E_2 < 0$
$E_1 > 0$	?	**false**	**false**
$E_1 = 0$	**false**	**true**	**false**
$E_1 < 0$	**false**	**false**	?

$E_1 > E_2$	$E_2 > 0$	$E_2 = 0$	$E_2 < 0$
$E_1 > 0$	?	**true**	**true**
$E_1 = 0$	**false**	**false**	**true**
$E_1 < 0$	**false**	**false**	?

$E_1 < E_2$	$E_2 > 0$	$E_2 = 0$	$E_2 < 0$
$E_1 > 0$	?	**false**	**false**
$E_1 = 0$	**true**	**false**	**false**
$E_1 < 0$	**true**	**true**	?

Careful examination of these tables gives rise to the following semantic rules:

Production	Semantic Rule
$E \rightarrow E_1 = E_2$	E.t = (E_1.p $\wedge$ E_2.p) $\vee$ (E_1.z $\wedge$ E_2.z) $\vee$ (E_1.n $\wedge$ E_2.n); E.f = E_1.p $\vee$ E_1.n $\vee$ E_2.p $\vee$ E_2.n;
$E \rightarrow E_1 > E_2$	E.t = E_1.p $\vee$ E_2.n; E.f = E_1.n $\vee$ E_2.p $\vee$ (E_1.z $\wedge$ E_2.z);
$E \rightarrow E_1 < E_2$	E.t = E_1.n $\vee$ E_2.p; E.f = E_1.p $\vee$ E_2.n $\vee$ (E_1.z $\wedge$ E_2.z);

The semantic rules for the productions of the relational operators '$\leq$' and '$\geq$', and '$\neq$' are obtained from the previous rules by interchanging the value for E.t and E.f, as indicated by the logical equivalences $(i \leq j) \equiv \neg(i > j)$, $(i \geq j) \equiv \neg(i < j)$, and $(i \neq j) \equiv \neg(i = j)$.

The semantic rules for all remaining productions are shown below:

Production	Semantic Rule	
$E \rightarrow$ (E_1)	E.t = E_1.t;	E.f = E_1.f;
$E \rightarrow$ const	E.t = (const.val);	E.f = $\neg$(const.val);
/* others */	E.t = **true**;	E.f = **true**;

Example: Some boolean expression and associated conditions E.t and E.f are shown in the following table, where we have used the FORTRAN notation for logical and relational operators:

E	E.t	E.f
`( A(I.J).EQ.0.0 )`	**true**	$(\texttt{I},\texttt{J}) \in E(A)$
`( A(I,J).EQ.9.0).AND.(X.EQ.Y) )`	$(\texttt{I},\texttt{J}) \in E(A)$	**true**
`( A(I,J).GE.(1.0 + ABS(X)) )`	$(\texttt{I},\texttt{J}) \in E(A)$	**true**
`( A(J,K).LT.(- SQRT(X) / 2.0) )`	$(\texttt{J},\texttt{K}) \in E(A)$	**true**
`( A(I,I).GT.A(2,2)).AND.(A(I,I).GT.4.0) )`	$(\texttt{I},\texttt{I}) \in E(A)$	**true**

Conditions associated with IF-statements

Obviously, we can safely skip instances of statements that either can be skipped according to previous made observations or that are under control of a boolean expression that cannot hold. Because the condition under which this boolean expression may hold is recorded in the synthesized attribute E.t, whereas the condition under which this expression may fail is recorded in E.f, the following semantic rule can be used to associate a condition with a general IF-statement, provided that evaluating any of the boolean expressions is free of side-effects (see section 5.3.3):

Production		Semantic Rule
stmt $\rightarrow$	**if** (E_1) **then**	stmt.cnd =
	stmt_list$_1$	(E_1.t $\wedge$ stmt_list$_1$.cnd)
	elseif (E_2) **then**	$\vee$
	stmt_list$_2$	(E_1.f $\wedge$ E_2.t $\wedge$ stmt_list$_2$.cnd)
	:	:
	elseif (E_{n-1}) **then**	$\vee$
	stmt_list$_{n-1}$	(E_1.f $\wedge \ldots \wedge$ E_{n-2}.f $\wedge$ E_{n-1}.t $\wedge$ stmt_list$_{n-1}$.cnd)
	else	$\vee$
	stmt_list$_n$	(E_1.f $\wedge \ldots \wedge$ E_{n-2}.f $\wedge$ E_{n-1}.f $\wedge$ stmt_list$_n$.cnd) ;
	endif	

Semantic rules for IF-statements without **elseif** or **else** branches are obtained by omitting the appropriate parts of the disjunction. The condition associated with an arbitrary statement list is obtained by taking the disjunction of all conditions associated with the individual statements, as defined by the following semantic rule:

Production	Semantic Rule
stmt_list $\rightarrow$ stmt stmt_list$_1$	stmt_list.cnd = stmt.cnd $\vee$ stmt_list$_1$.cnd;

Example: Condition '$(\texttt{I},\texttt{J}) \in E(A)$' is associated with the following one-way IF-statement, because the boolean expression cannot hold for non-entries:

```
IF ( ABS(A(I,J)) > ABS(PIV) ) THEN     ← (I,J) ∈ E(A)
  ...                                      (E.t = (I,J) ∈ E(A), E.f = true)
ENDIF
```

On the other hand, although E.t=**true** and E.f=**true** for the boolean expression used in the following one-way IF-statement, still the condition '$(\texttt{I},\texttt{J}) \in E(A)$' is associated with this statement, because the body can be skipped for non-entries:

```
IF (I.NE.1) THEN      ← (I,J) ∈ E(A)
  X = X + A(I,J)      ← (I,J) ∈ E(A)
ENDIF
```

Example: Nested IF-statements are correctly accounted for:

```
IF (A(I,J).NE.0) THEN     ← (E.t = (I,J) ∈ E(A), E.f=true)
  IF (B(K,J).NE.0) THEN   ← (E.t = (K,J) ∈ E(B), E.f=true)
    X = X + C(K,K)        ← (K,K) ∈ E(C)
  ENDIF
  Y = Y + D(I,K)          ← (I,K) ∈ E(D)
ENDIF
```

If we assume that all arrays are enveloping data structures, the conditions shown to the right are associated with the boolean expressions and assignment statements. Hence, condition $(\texttt{K},\texttt{J}) \in E(B) \wedge (\texttt{K},\texttt{K}) \in E(C)$ is associated with the innermost IF-statement and the condition shown below is associated with the outermost IF-statement:

$$(\texttt{I},\texttt{J}) \in E(A) \quad \wedge \quad (\quad ((\texttt{K},\texttt{J}) \in E(B) \wedge (\texttt{K},\texttt{K}) \in E(C)) \quad \vee \quad (\texttt{I},\texttt{K}) \in E(D) \quad)$$

Example: A somewhat contrived example to illustrate the potential of associating conditions with general IF-statements is shown below:

```
IF (A(I,J) .GT. 0.0) THEN        ← (E.t = (I,J) ∈ E(A), E.f = true)
  POS = POS + 1                  ← true
ELSEIF (A(I,J) .LT. 0.0) THEN    ← (E.t = (I,J) ∈ E(A), E.f = true)
  NEG = NEG + 1                  ← true
ELSE
  X = X + B(I,I)                 ← (I,I) ∈ E(B)
ENDIF
```

Because the condition of the first and second branch can only hold for entries of A (although the loop-body itself cannot exploit sparsity), whereas the last branch (although executed unconditionally) can safely be skipped for non-entries of B, the whole general IF-statement can be placed under control of the condition '$(\texttt{I},\texttt{J}) \in E(A) \vee (\texttt{I},\texttt{I}) \in E(B)$', the evaluation of which is illustrated below:

$$\begin{array}{lll} & ((\texttt{I},\texttt{J}) \in E(A) \wedge \mathbf{true}) & \\ \vee & (\mathbf{true} \wedge (\texttt{I},\texttt{J}) \in E(A) \wedge \mathbf{true}) & \\ \vee & (\mathbf{true} \wedge \mathbf{true} \wedge (\texttt{I},\texttt{I}) \in E(B)) & \equiv \quad (\texttt{I},\texttt{J}) \in E(A) \vee (\texttt{I},\texttt{I}) \in E(B) \end{array}$$

Example: The first branch of the following IF-statement may be skipped for non-entries of B and C, whereas the second branch (which cannot exploit sparsity) is only executed if the first boolean expression *does not hold*:

```
IF (A(I,J) .EQ. 0.0) THEN        ← (E.t = true, E.f = (I,J) ∈ E(A))
  X = X + B(I,I)                 ← (I,I) ∈ E(B)
  X = X + C(I,I)                 ← (I,I) ∈ E(C)
ELSE
  X = 0.0                        ← true
ENDIF
```

The following condition is associated with the whole IF-statement:

$$\begin{array}{rl} & (\ \mathbf{true} \wedge ((\mathtt{I},\mathtt{I}) \in E(B) \vee (\mathtt{I},\mathtt{I}) \in E(C))\ ) \vee ((\mathtt{I},\mathtt{J}) \in E(A) \wedge \mathbf{true}) \\ \equiv & (\mathtt{I},\mathtt{I}) \in E(B) \vee (\mathtt{I},\mathtt{I}) \in E(C) \vee (\mathtt{I},\mathtt{J}) \in E(A) \end{array}$$

Conditions associated with DO-loops

The following semantic rule is used to associate a condition with a DO-loop, where the function 'filter' discards all guards in which the loop-index $\exp_1$ is used by replacing these guards by '**true**':

Production		Semantic Rule
stmt →	**do** $\exp_1$ '=' $\exp_2$ ',' $\exp_3$ ',' $\exp_4$ stmt_list **enddo**	stmt.cnd = filter($\exp_1$.id, stmt_list.cnd);

Example: The following fragment has been annotated with the conditions that are associated with all statements:

```
DO I = 1, M                    ← true
  DO J = 1, N                  ← (I,I) ∈ E(A)
    X = X + A(I,I) * B(I,J)    ← (I,I) ∈ E(A) ∧ (I,J) ∈ E(B)
  ENDDO
ENDDO
```

5.3.2 Dominating Guards

Recall that a positive guard ψ **dominates** a condition ϕ, if $\phi \Rightarrow \psi$ holds. Since each condition consists of conjunctions and disjunction of guards and boolean constants, the following procedure in pseudo-code can be used to determine whether a guard `g` dominates the condition `c`, where construct '`match`' uses the structure of condition `c` to determine which of the branches must be taken:

```
boolean function dom(g, c)
begin
  match c on
    c1∧c2      : dom := dom(g, c1) or  dom(g, c2);
    c1∨c2      : dom := dom(g, c1) and dom(g, c2);
    true       : dom := false;
    false      : dom := true;
    otherwise  : dom := (g == c);
  end on
end
```

The test '`g==c`' succeeds if the guards involve the same enveloping data structure and the corresponding subscripts are structurally equivalent, which means that all coefficients of the loop indices in the common nesting depth are identical, whereas all other coefficients are zero.

Furthermore, we say that *a guard dominates the loop-body of the* $\mathtt{I}_c$*-loop, if this guard dominates the condition of each statement in the loop-body of this DO-loop at nesting depth* c.

Example: The subscripts of the occurrences of array `A` in the following example are structurally equivalent:

```
DO I = 1, 100
  X = X + A(I,I)            ← (I,I) ∈ E(A)
  DO J = 1, 100             ← (I,I) ∈ E(A)
    C(J) = C(J) + A(I,I)    ← (I,I) ∈ E(A)
  ENDDO
  IF (A(I,I).NE.0) THEN     ← (I,I) ∈ E(A)
    X = 0.0                 ← true
  ENDIF
ENDDO
```

Since the guard '$(\mathtt{I},\mathtt{I}) \in E(A)$' is associated with all statements at nesting depth 1, this guard dominates the loop-body of the I-loop.

5.3.3 Accounting for Side-Effects

Obviously, we would affect the semantics of a program by skipping instances of a statement in which functions with possible side-effects are called. In the following fragment, for instance, it seems that condition '$(\mathtt{I},\mathtt{J}) \in E(A)$' can be associated with the assignment statement S_1 in the main program. However, since the value of the actual argument K changes with each call to function F, eliminating instances of S_1 would be incorrect:

```
      PROGRAM MAIN                          REAL FUNCTION F(P)
      INTEGER I, J, K                       INTEGER P
      REAL    A(100,100)                    P = P + 1
C_SPARSE(A)                                 F = 40 * P
                                            RETURN
      DO I = 1, 100                         END
        DO J = 1, 100
S1:       ACC = ACC + A(I,J) * F(K)
        ENDDO
      ENDDO
      END
```

Similar arguments hold for the evaluation of boolean expressions, loop bounds, and strides. For example, although the boolean expression used in the following IF-statement cannot hold for non-entries, it must be evaluated to account for the side-effects of function F:

```
IF ( A(I,J) .GT. ABS(F(K)) ) THEN
  ...
ENDIF
```

Therefore, if a function with possible side-effects (where executing a STOP-statement is also a side-effect) may be evaluated in the left- or right-hand side expression of an assignment statement, in any boolean expression of an IF-statement, or in the stride or loop bounds of a DO-loop (but where we ignore the 'side-effect' of assigning the last value to the loop index), then we overrule any previous computed condition by associating the condition '**true**' with this statement. Note that since intrinsic functions have no side-effects, we can safely associate condition '$(\mathtt{I},\mathtt{J}) \in E(A)$' with the following assignment statement:

```
ACC = ACC + ABS(A(I,J))
```

5.3.4 Condition Improvement

Certain conventional program transformations can be used to improve the conditions associated with statements or to enable the generation of more efficient sparse code.

Loop Distribution

Loop distribution can be used to increase the number of loop-bodies that are dominated by a particular guard.
Example: In the following fragment, loop distribution yields two loops , the loop-bodies of which are dominated by the guards '$(\texttt{I},\texttt{J}) \in E(A)$' and '**true**', respectively:

```
DO I = 1, 100                                  DO I = 1, 100
  DO J = 1, 100                                  DO J = 1, 100
    A(I,J) = A(I,J) * 3.0                          A(I,J) = A(I,J) * 3.0   ← (I,J) ∈ E(A)
    D(I,J) = 10.0                    →           ENDDO
  ENDDO                                          DO J = 1, 100
ENDDO                                              D(I,J) = 10.0           ← true
                                                 ENDDO
                                               ENDDO
```

Hence, this transformation may effectively reduce the total number of times the first statement is executed. In general, loop distribution is valid if there are no lexically backward data dependences. Reordering the statements according to the strongly connected components in the data dependence graph can assist in making effective use of loop distribution.

Loop Fusion

Loop fusion can be used to reduce overhead of loops with the potential of guard encapsulation, by fusing adjacent loops, the loop-bodies of which are dominated by the same guard. Loop fusion is allowed if the bounds of the adjacent loops are identical (which can be achieved by adjusting the bounds, partially unrolling the loop or adding some conditionals), and if no loop-carried data dependence between a statement instance of the second loop to a statement instance of the first loop arises after fusion. Statement reordering can assist in making particular loops adjacent, as is demonstrated in the following example.
Example: Below, two loops with a loop-body dominated by the guard '$(\texttt{I},\texttt{J}) \in E(A)$' can be distinguished. Statement reordering and loop fusion can be used to merge the two loop-bodies:

```
DO I = 1, 100
  DO J = 1, 100                                DO I = 1, 100
    ACC = ACC + A(I,J)                           DO J = 1, 100
  ENDDO                                            ACC    = ACC + A(I,J)   ← true
  COPY(I) = ACC                    →               A(I,J) = 3.0 * A(I,J)   ← (I,J) ∈ E(A)
  DO J = 1, 100                                  ENDDO
    A(I,J) = 3.0 * A(I,J)                        COPY(I) = ACC             ← true
  ENDDO                                        ENDDO
ENDDO
```

Update Expression Splitting

If several occurrences of enveloping data structures appear in one assignment statement, then in some cases the condition associated with the statement consists of a disjunction of guards, as illustrated below:

```
DO I = 1, 100
  DO J = 1, 100
    Y = Y + A(I,J) ** 3 + B(I,J) ** 3
  ENDDO
ENDDO
```

Obviously, none of the guards dominates the condition '$(\texttt{I},\texttt{J}) \in E(A) \vee (\texttt{I},\texttt{J}) \in E(B)$' associated with the assignment statement. This implies that the subcomputations in the statement must be performed for any instance in which at least one entry of either A or B is accessed.

However, if roundoff errors may accumulate differently, then we can use the associativity, commutativity, and distributivity laws of arithmetic operators. The loop-body can be rewritten into the following two statements, after which loop distribution becomes possible if the data dependence cycle is recognized as a coupled reduction [228]. Now, each loop-body is dominated by a guard:

```
DO I = 1, 100                          DO I = 1, 100
  DO J = 1, 100                          DO J = 1, 100
    Y = Y + A(I,J) ** 3                    Y = Y + A(I,J) ** 3   ← (I,J) ∈ E(A)
    Y = Y + B(I,J) ** 3          →         ENDDO
  ENDDO                                  DO J = 1, 100
ENDDO                                      Y = Y + B(I,J) ** 3   ← (I,J) ∈ E(B)
                                         ENDDO
                                       ENDDO
```

Scalar Forward Substitution

Programmers frequently use a temporary scalar variable to save loop invariant accesses to an array. Unfortunately, the use of temporary scalars may obscure the fact that sparsity can be exploited to reduce computational time. In the following loop, for instance, the condition '**true**' results for both assignment statements. However, after scalar forward substitution [234, p178-179] and dead-code elimination [3] if the value of T is not required afterwards, the condition of the remaining statement changes into '$(\mathtt{I},\mathtt{I}) \in E(A)$':

```
DO I = 1, 100                          DO I = 1, 100
  T = A(I,I)                             DO J = 1, I - 1
  DO J = 1, I - 1                          D(I,J) = D(I,J) - A(I,I)   ← (I,I) ∈ E(A)
    D(I,J) = D(I,J) - T          →       ENDDO
  ENDDO                                ENDDO
ENDDO
```

5.4 Access Patterns of Two-Dimensional Arrays

Since enveloping data structures are operated upon in the original dense program, analysis of two-dimensional arrays plays an important role in the sparse compiler.

5.4.1 Preliminaries of Access Patterns

In this section, preliminaries related to the access patterns of occurrences of two-dimensional arrays are given.

Definitions

For an occurrence of a two-dimensional array with admissible subscripts $F(\vec{\mathtt{I}}) = \vec{v} + W\vec{\mathtt{I}}$ appearing in a nested loop with index vector $\vec{\mathtt{I}} = (\mathtt{I}_1, \ldots, \mathtt{I}_d)^T$, the index set of all elements accessed in successive iterations of the innermost $\mathtt{I}_d$-loop (with bounds $\mathtt{L}_d$ and $\mathtt{U}_d$) for a fixed iteration of more outer DO-loops is called a **true access pattern** $P(\mathtt{I}_1, \ldots, \mathtt{I}_{d-1}) \subseteq \mathcal{Z}^2$ of this occurrence:[6]

$$P(\mathtt{I}_1, \ldots, \mathtt{I}_{d-1}) = \{F(\vec{\mathtt{I}})^T \mid \mathtt{I}_d \in [\mathtt{L}_d, \mathtt{U}_d]\} \tag{5.1}$$

The **true access direction** $\vec{r} \in \mathcal{Z}^2$ of this occurrence is defined as the last column of W:

$$\vec{r} = (r_1, r_2)^T = W(\underbrace{0, \ldots, 0,}_{d-1} 1)^T$$

[6] The elements are denoted as row vectors to reflect the correspondence with matrix indices (viz. (i, j) vs. a_{ij}).

If $\vec{r} = \vec{0}$, then true access patterns are called **scalar-wise**. If $\vec{r} \neq \vec{0}$, the true access patterns are called **row-wise** if $r_1 = 0$ and $r_2 \neq 0$, **column-wise** if $r_1 \neq 0$ and $r_2 = 0$, or **diagonal-wise** otherwise. Likewise, if $k \geq 1$ denotes the index of the last nonzero column in matrix W, then we define the **effective access direction** $\vec{x} \in \mathcal{Z}^2$ of the occurrence as the kth column of W. The index set of all elements accessed in successive iterations of the $\mathtt{I}_k$-loop for a fixed iteration of more outer DO-loops form is defined as an **effective access pattern** $P(\mathtt{I}_1, \ldots, \mathtt{I}_{k-1}) \subseteq \mathcal{Z}^2$ of the occurrence (classified as row-, column-, or diagonal-wise in a similar manner). If such a column does not exist, the effective access direction is zero and we leave effective access patterns undefined.

Only for scalar-wise true access patterns, the true and effective access direction may differ, and we can speak of effective access patterns if the effective access direction is nonzero. For occurrences in scalar-statements (viz. $d = 0$) or occurrences with inadmissible subscripts, we leave both the true as well as the effective access patterns undefined, and we set $\vec{r} = \vec{0}$ and $\vec{x} = \vec{0}$.

Given the effective access direction $\vec{x} = (x_1, x_2)^T$ of an occurrence, we define the **normalized direction** $\vec{x}^{\,n} \in \mathcal{Z}^2$ of this occurrence as follows:

$$\vec{x}^{\,n} = \begin{cases} (\quad 0 \quad , \quad 0 \quad)^T & \textbf{if } \vec{x} = \vec{0} \\ (\quad s \cdot \frac{|x_1|}{g} \quad , \quad \frac{|x_2|}{g} \quad)^T & \textbf{otherwise} \end{cases} \tag{5.2}$$

In this formula, g denotes $\gcd(x_1, x_2)$ and we define $s = (x_1 \cdot x_2 > 0)? + 1 : -1$. In this manner, the components of normalized directions are relatively prime, whereas directions that are linearly dependent become normalized into a uniform direction.

Example: Consider the following occurrence of a two-dimensional array A:

```
DO I1 = 1, 3
  DO I2 = I1, 4
     ... A(9-2*I2,I1) ...
  ENDDO
ENDDO
```

$$F(\vec{\mathtt{I}}) = \begin{pmatrix} 9 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 & -2 \\ 1 & 0 \end{pmatrix} \vec{\mathtt{I}}$$

Both the true and effective access direction of this occurrence are $(-2, 0)^T$, which gives rise to the normalized access direction $\vec{x}^{\,n} = (-1, 0)^T$. The column-wise true (and effective) access patterns of this occurrence are defined as follows for $\mathtt{1} \leq \mathtt{I}_1 \leq \mathtt{3}$:

$$P(\mathtt{I}_1) = \{(\mathtt{9} - \mathtt{2} * \mathtt{I}_2, \mathtt{I}_1) \mid \mathtt{I}_1 \leq \mathtt{I}_2 \leq \mathtt{4}\}$$

Hence, the following access patterns are associated with this occurrence:

$$\begin{array}{rcl} P(1) & = & \{ \quad (7,1), (5,1), (3,1), (1,1) \quad \} \\ P(2) & = & \{ \qquad (5,2), (3,2), (1,2) \quad \} \\ P(3) & = & \{ \qquad\qquad (3,3), (1,3) \quad \} \end{array}$$

Example: Consider the following occurrence of a two-dimensional array A:

```
DO I1 = 1, 4
  DO I2 = 1, 50
     ... A(10,3 * I1) ...
  ENDDO
ENDDO
```

$$F(\vec{\mathtt{I}}) = \begin{pmatrix} 10 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 \\ 3 & 0 \end{pmatrix} \vec{\mathtt{I}}$$

The true access direction of this occurrence is $\vec{r} = \vec{0}$, indicating that this occurrence has scalar-wise true access patterns. The four true access patterns that are associated with this occurrence are shown below:

$$\begin{array}{rcl} P(1) & = & \{ \quad (10,3) \quad \} \\ P(2) & = & \{ \quad (10,6) \quad \} \end{array} \qquad \begin{array}{rcl} P(3) & = & \{ \quad (10,9) \quad \} \\ P(4) & = & \{ \quad (10,12) \quad \} \end{array}$$

The effective access direction, however, is $\vec{x} = (0,3)^T$, which gives rise to the normalized access direction $\vec{x}^{\,n} = (0,1)^T$ and the following single effective access pattern:

$$P \quad = \quad \{ \quad (10,3),(10,6),(10,9),(10,12) \quad \}$$

Complications for Access Patterns

The compile-time representation and manipulation of access patterns may be difficult. In fact, this representation requires complete information about the subscript functions as well as the iteration space, which may be impossible in the presence of inadmissible loop bounds and subscripts. Additionally, the generality of representation complicates most operations, such as testing whether access patterns associated with two different occurrences of the same array are identical.

To overcome these difficulties, access patterns are represented by a simpler and uniform representation, consisting of the normalized access direction together with a conservative approximation of the index of the part of the array that may be accessed [30, 32]. A convenient representation of such an index set is the simple section [15, 16]. Even in the presence of inadmissible loop bounds and subscripts, using simple sections enables the computation of a conservative approximation of the index set of the accessed part of the array. In addition, this representation can be stored and manipulated efficiently and supports access shapes that are frequently found in numerical algorithms. This uniform representation of access patterns is discussed in the next section.

5.4.2 Two-Dimensional Simple Sections

A convenient representation of the index set of a part in a two-dimensional array (at programming level) or, likewise, the index set of a region in a matrix (at logical level) consist of the two-dimensional simple section [15, 16]. In this section, simple sections are defined and implementations of some operations on simple sections are presented.

Definitions

A **two-dimensional simple section** $S \subseteq \mathcal{Z}^2$ consists of all discrete points $(i,j) \in \mathcal{Z}^2$ in a convex polygon defined by the following system of 8 linear inequalities, where $\sigma_i, \tau_i \in \mathcal{Z}$:

$$\begin{array}{ccccc} \sigma_1 & \leq & i & \leq & \tau_1 \\ \sigma_2 & \leq & j & \leq & \tau_2 \\ \sigma_3 & \leq & i+j & \leq & \tau_3 \\ \sigma_4 & \leq & i-j & \leq & \tau_4 \end{array}$$

Each pair of inequalities (e.g. $\sigma_3 \leq i+j \leq \tau_3$) is referred to as a **boundary pair**. Let the matrix $\mathcal{M}$ be defined as follows:

$$\mathcal{M} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \\ 1 & -1 \end{pmatrix}$$

Then, the simple section can be denoted as shown below:

$$S = \{(i,j) \in \mathcal{Z}^2 \mid (\sigma_1,\sigma_2,\sigma_3,\sigma_4)^T \leq \mathcal{M}(i,j)^T \leq (\tau_1,\tau_2,\tau_3,\tau_4)^T\} \tag{5.3}$$

A two-dimensional simple section can be stored in a compact way as integer vectors $\vec{\sigma} \in \mathcal{Z}^4$ and $\vec{\tau} \in \mathcal{Z}^4$, indicating the constant **boundary values** used in the boundary pairs. Another advantage of this representation is that certain operations on simple sections are easy to perform.

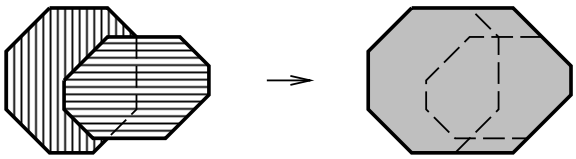


Figure 5.3: Smallest Enveloping Simple Section

Intersection and Union of Simple Sections

Simple sections have the following property:

Proposition 5.1 *Two-dimensional simple sections are closed under intersection*

PROOF Let $\vec{\sigma} \in \mathcal{Z}^4$ and $\vec{\tau} \in \mathcal{Z}^4$ denote the boundary values of a simple section $S \subseteq \mathcal{Z}^2$. Likewise, let $\vec{\sigma}\,' \in \mathcal{Z}^4$ and $\vec{\tau}\,' \in \mathcal{Z}^4$ denote the boundary values of another simple section $S' \subseteq \mathcal{Z}^2$. The intersection of these two sets is defined as follows:

$$S \cap S' = \{(i,j) \in \mathcal{Z}^2 \mid (i,j) \in S \text{ and } (i,j) \in S'\}$$

This implies that each point $(i,j) \in \mathcal{Z}^2$ in this intersection satisfies the inequalities imposed by *both* simple sections. Hence, we can express the intersection as (5.3), where $\sigma_i'' = \max(\sigma_i, \sigma_i')$ and $\tau_i'' = \min(\tau_i, \tau_i')$:

$$S \cap S' = \{(i,j) \in \mathcal{Z}^2 \mid (\sigma_1'', \sigma_2'', \sigma_3'', \sigma_4'')^T \leq \mathcal{M}(i,j)^T \leq (\tau_1'', \tau_2'', \tau_3'', \tau_4'')^T\}$$

□

Consequently, we can construct the intersection $S \cap S'$ of two simple sections $S \subseteq \mathcal{Z}^2$ and $S' \subseteq \mathcal{Z}^2$ by taking the innermost boundary values for all boundary pairs. This observation gives rise to the following procedure `intersect` in pseudo-code to construct the intersection of two simple sections stored in `s1` and `s2`. The constructs '`s.l[i]`' and '`s.u[i]`' are used to access the `i`th component of $\vec{\sigma} \in \mathcal{Z}^4$ and $\vec{\tau} \in \mathcal{Z}^4$ respectively, forming the boundary values of the simple section stored in variable `s`:

```
procedure intersect(s1, s2, var s)
begin
  for i := 1, 4 do
    s.l[i] := max(s1.l[i],s2.l[i]);
    s.u[i] := min(s1.u[i],s2.u[i]);
  enddo
end
```

Unfortunately, simple sections are not closed under union. The *smallest* simple section that contains the union of two simple sections $S \subseteq \mathcal{Z}^2$ and $S' \subseteq \mathcal{Z}^2$ is obtained by taking the outermost boundary values for the boundary pairs [15], as illustrated in figure 5.3. The notation $S \uplus S'$ is used to denote this kind of union.

Procedure `combine` constructs the smallest enveloping simple section of two non-empty simple sections stored in `s1` and `s2`:

```
procedure combine(s1, s2, var s)
begin
  for i := 1, 4 do
    s.l[i] := min(s1.l[i],s2.l[i]);
    s.u[i] := max(s1.u[i],s2.u[i]);
  enddo
end
```

Consider, for instance, the following two simple sections that are shown in figure 5.4:

$$\begin{cases} S_1 &= \{(i,j) \in \mathcal{Z}^2 \mid (1,1,2,0)^T \leq \mathcal{M}(i,j)^T \leq (6,3,9,5)^T\} \\ S_2 &= \{(i,j) \in \mathcal{Z}^2 \mid (1,2,3,-5)^T \leq \mathcal{M}(i,j)^T \leq (4,7,10,-1)^T\} \end{cases}$$

Using procedure `combine` to compute $S_1 \uplus S_2$ yields the following simple section, illustrated with a dashed line in figure 5.4:

$$S_1 \uplus S_2 = \{(i,j) \in \mathcal{Z}^2 \mid (1,1,2,-5)^T \leq \mathcal{M}(i,j)^T \leq (6,7,10,5)^T\}$$

Boundary Refinement

Each boundary pair in a two-dimensional simple section is defined by two **boundaries**, which are straight lines of the form $i = c$, $j = c$, $i + j = c$ or $i - j = c$. A boundary is called **tight** if it contains at least one discrete point of the simple section, or **non-tight** otherwise.

A disadvantage of the given implementation of procedure `intersect` is that, even if all boundaries defining the original simple sections are tight, some *non-tight* boundaries may arise in the resulting intersection. Consider, for instance, the following two simple sections:

$$\begin{cases} S_3 &= \{(i,j) \in \mathcal{Z}^2 \mid (1,1,2,-4)^T \leq \mathcal{M}(i,j)^T \leq (5,5,6,4)^T\} \\ S_4 &= \{(i,j) \in \mathcal{Z}^2 \mid (1,1,2,0)^T \leq \mathcal{M}(i,j)^T \leq (5,5,10,4)^T\} \end{cases}$$

Using procedure `intersect` to compute $S_3 \cap S_4$, results in the following simple section:

$$S_3 \cap S_4 = \{(i,j) \in \mathcal{Z}^2 \mid (1,1,2,0) \leq \mathcal{M}(i,j)^T \leq (5,5,6,4)\}$$

However, as can be seen in figure 5.5, the boundary $j = 5$ becomes non-tight. Although the intersection $S_3 \cap S_4$ is properly defined, such boundaries are undesirable because they may affect the outcome of the construction of the smallest simple section enveloping a number of simple sections in which this intersection is involved [15]. Therefore, a procedure to refine the non-tight boundaries of simple section is required.

The following implementation of such a procedure is based on the observation that the boundaries of a boundary pair can be refined using pair-wise combinations of all other boundary pairs. For instance, because $j \leq \tau_2$ and $\sigma_3 \leq i + j$, we know that $\sigma_3 - \tau_2 \leq i$. Likewise, because $\sigma_1 \leq i$ and $i - j \leq \tau_4$, we may conclude that $2 \cdot \sigma_1 - \tau_4 \leq i + j$. In the procedure, a Pascal-like '`with(s)`'-construct is used for notational convenience:

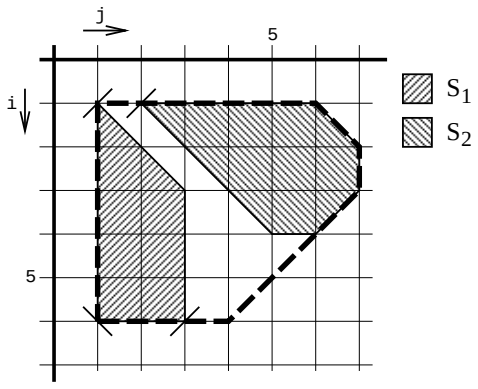


Figure 5.4: Smallest Enveloping Simple Section

```
procedure refine(var s)
begin
  if (not empty(s)) then
    with (s) do
      l[1] := max( l[1], l[3]-u[2], l[2]+l[4], ⌈(l[3]+l[4])/2⌉ );
      u[1] := min( u[1], u[3]-l[2], u[2]+u[4], ⌊(u[3]+u[4])/2⌋ );
      l[2] := max( l[2], l[3]-u[1], l[1]-u[4], ⌈(l[3]-u[4])/2⌉ );
      u[2] := min( u[2], u[3]-l[1], u[1]-l[4], ⌊(u[3]-l[4])/2⌋ );
      l[3] := max( l[3], l[1]+l[2], 2*l[1]-u[4], 2*l[2]+l[4] );
      u[3] := min( u[3], u[1]+u[2], 2*u[1]-l[4], 2*u[2]+u[4] );
      l[4] := max( l[4], l[1]-u[2], 2*l[1]-u[3], l[3]-2*u[2] );
      u[4] := min( u[4], u[1]-l[2], 2*u[1]-l[3], u[3]-2*l[2] );
    enddo
  endif
end
```

Because tight boundaries may cause a refinement of other boundaries, but cannot be refined themselves, a single execution of this procedure suffices.

An improved definition of procedure `intersect` is obtained by adding a call to procedure `refine` after the computation of the most interior values for each boundary pair. In this manner, boundary $j = 5$ is refined into $j = 3$ for the previous example, as implied by $i + j \leq 6$ and $0 \leq i - j$:

$$S_3 \cap S_4 = \{(i,j) \in \mathcal{Z}^2 \mid (1,1,2,0)^T \leq \mathcal{M}(i,j)^T \leq (5,3,6,4)^T\}$$

Other Operations on Simple Sections

If (after boundary refinement) we have $\sigma_i > \tau_i$ for at least one boundary pair of a simple section $S \subseteq \mathcal{Z}^2$, then $S = \emptyset$. For example, computing the empty intersection $S_1 \cap S_2$ of the previous section using `intersect` yields the following simple section with $\sigma_4 > \tau_4$:

$$S_1 \cap S_2 = \{(i,j) \in \mathcal{Z}^2 \mid (1,2,3,0)^T \leq \mathcal{M} \leq (4,3,9,-1)^T\}$$

This observation gives rise to function `empty`, determining whether a simple section stored in `s` is empty, and function `overlap`, which can be used to detect a non-empty intersection of two simple sections stored in `s1` and `s2`:

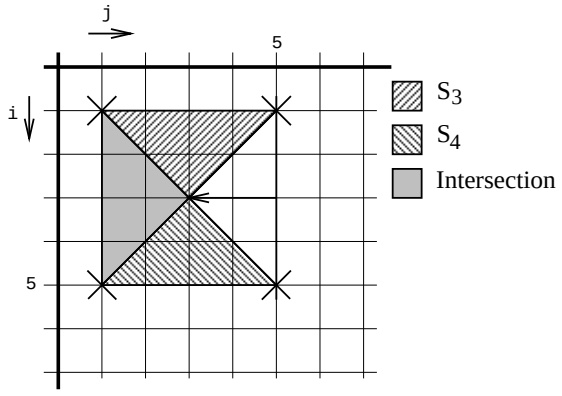


Figure 5.5: Refinement of $j = 5$

```
boolean function empty(s)
begin
  empty :=
   ( (s.l[1] > s.u[1])
  or (s.l[2] > s.u[2])
  or (s.l[3] > s.u[3])
  or (s.l[4] > s.u[4]) );
end
```

```
boolean function overlap(s1, s2)
begin
  intersect(s1, s2, tmp);
  overlap := not empty(tmp);
end
```

If all boundaries are tight, the following function can be used to determine if $S \subseteq S'$ holds for two non-empty simple sections stored in `s1` and `s2`:

```
boolean function subseteq(s1, s2)
begin
  subseteq :=
   (   (s1.l[1] >= s2.l[1]) and (s1.u[1] <= s2.u[1])
   and (s1.l[2] >= s2.l[2]) and (s1.u[2] <= s2.u[2])
   and (s1.l[3] >= s2.l[3]) and (s1.u[3] <= s2.u[3])
   and (s1.l[4] >= s2.l[4]) and (s1.u[4] <= s2.u[4]) );
end
```

Moreover, under the assumption that all boundaries have been refined, we can use the following simple and efficient method to compute the number of discrete points in a simple section $S \subseteq \mathcal{Z}^2$, denoted by $|S|$. This number can be determined by computing the number of discrete points in the rectangle defined by the first two boundary pairs of the simple section, followed by subtraction of the number of points in the four triangles that are cut off by the rectangle that is defined by the other two boundary pairs. This gives rise to the following function `num`, in which an auxiliary function `triangle` is used to compute the number of points in a triangle:

```
integer function num(s)
begin
  num := 0;
  if (not empty(s)) then
    with(s) do
      num =  (u[1]-l[1]+1) * (u[2]-l[2]+1)
      - triangle(l[4]+u[2]-l[1]) - triangle(u[1]+u[2]-u[3])
      - triangle(l[3]-l[1]-l[2]) - triangle(u[1]-l[2]-u[4]);
    enddo
  endif
end
```

```
integer function triangle(n)
begin
  triangle := (n*(n+1))/2;
end
```

Application of function `num` to the following simple section, illustrated in figure 5.6, reveals that $|S| = 20$ holds (i.e. $36 - 15 - 0 - 0 - 1$):

$$S = \{(i,j) \in \mathcal{Z}^2 \mid (-2,-2,-4,0)^T \leq \mathcal{M}(i,j)^T \leq (3,3,6,4)^T\}$$

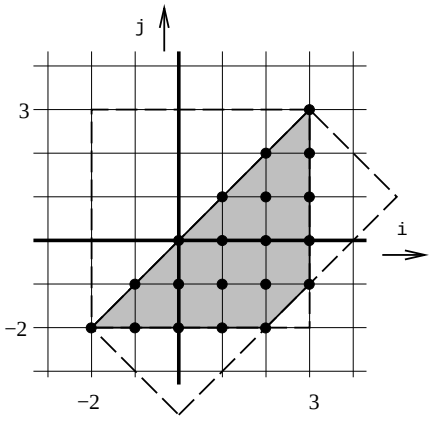


Figure 5.6: Computation of $|S|$

Simple Section Computation

In this section, we present a method to compute the boundary values $\vec{\sigma} \in \mathcal{Z}^4$ and $\vec{\tau} \in \mathcal{Z}^4$ of a simple section approximating the index set of the part of the array that may be accessed by an occurrence.

We assume that this occurrence appears in a loop index vector $\vec{\mathtt{I}} = (\mathtt{I}_1, \ldots, \mathtt{I}_d)^T$ and *simple* bounds, and that the occurrence has admissible subscripts $F(\vec{\mathtt{I}}) = \vec{v} + W\vec{\mathtt{I}}$. The method consists of computing the extremal values of the four integer expressions i, j, $i+j$, and $i-j$, where $(i,j) = F(\vec{\mathtt{I}})^T$:

$$\begin{array}{rcll} v_1 & + & \sum_{j=1}^{d} & w_{1j} \cdot \mathtt{I}_j \\ v_2 & + & \sum_{j=1}^{d} & w_{2j} \cdot \mathtt{I}_j \\ v_1 + v_2 & + & \sum_{j=1}^{d} & (w_{1j} + w_{2j}) \cdot \mathtt{I}_j \\ v_1 - v_2 & + & \sum_{j=1}^{d} & (w_{1j} - w_{2j}) \cdot \mathtt{I}_j \end{array} \tag{5.4}$$

A lower bound of the ith expression in (5.4) defines the value of the ith component of the vector $\vec{\sigma} \in \mathcal{Z}^4$. Likewise, an upper bound of the ith expression defines a value of the ith component of vector $\vec{\tau} \in \mathcal{Z}^4$. Such lower and upper bounds can be obtained by successively replacing the loop indices by extremal values *in decreasing order of nesting depth*.

Starting with $k = d$, each expression in (5.4) can be expressed as follows:

$$a_0 + \sum_{j=1}^{k} a_j \cdot \mathtt{I}_j \tag{5.5}$$

If index $\mathtt{I}_k$ is bounded as $\mathtt{L}_k \le \mathtt{I}_k \le \mathtt{U}_k$, a lower bound of (5.5) is defined by the following inequality, in which $a^+ = \max(a, 0)$ and $a^- = \max(-a, 0)$ [19, p52-54]:

$$a_0 + \sum_{i=1}^{k-1} a_i \cdot \mathtt{I}_i + (a_k^+ \cdot \mathtt{L}_k - a_k^- \cdot \mathtt{U}_k) \;\le\; a_0 + \sum_{i=1}^{k} a_i \cdot \mathtt{I}_i \tag{5.6}$$

Because either $a_k^+ \neq 0$ or $a_k^- \neq 0$ (but not both), only one of the loop bounds of index $\mathtt{I}_k$ is actually required. If we assume that the required loop bound is simple,[7] i.e. it can be expressed as $b_0 + \sum_{j=1}^{k-1} b_j \cdot \mathtt{I}_j$ where all $b_i \in \mathcal{Z}$, then this lower bound can be expressed in the form (5.5) again for a lower value of k. Repetitively using inequality (5.6) to eliminate the loop indices in decreasing order of nesting depth eventually yields a constant, which forms the corresponding component of vector $\vec{\tau} \in \mathcal{Z}^4$.

Likewise, an upper bound of (5.5) is defined by the following inequality:

$$a_0 + \sum_{i=1}^{k} a_i \cdot \mathtt{I}_i \;\le\; a_0 + \sum_{i=1}^{k-1} a_i \cdot \mathtt{I}_i + (a_k^+ \cdot \mathtt{U}_k - a_k^- \cdot \mathtt{L}_k) \tag{5.7}$$

Repetitively eliminating loop indices in decreasing order of nesting depth using this inequality (5.7) eventually yields the corresponding component of vector $\vec{\sigma} \in \mathcal{Z}^4$.

Example: The method is illustrated for the following occurrence of a two-dimensional array A, where $\vec{\mathtt{I}} = (\mathtt{I}, \mathtt{J})^T$:

```
DO I = 1, 10
  DO J = 1, I
    ... A(I,J) ...
  ENDDO
ENDDO
```

$\mathtt{L}_1 = 1, \quad \mathtt{U}_1 = 10$

$\mathtt{L}_2 = 1, \quad \mathtt{U}_2 = \mathtt{I}$

[7] Because we are only interested in the range of values for each index, we allow for arbitrary strides, where negative strides are dealt with by interchanging the role of the loop bounds.

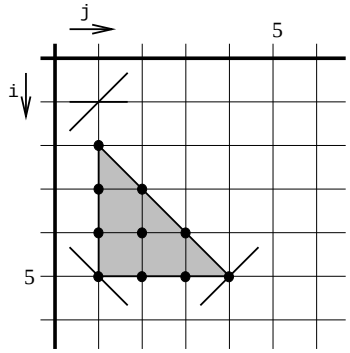


Figure 5.7: Boundaries before Refinement

The eight boundary values are computed by determining the lower and upper bounds for each of the expressions i, j, $i+j$, and $i-j$, where $(i,j) = (\mathtt{I},\mathtt{J})$:

$$\begin{array}{lllllllll}
\mathtt{1} & & & \leq & \mathtt{I} & \leq & & & \mathtt{10} \\
\mathtt{1} & & & \leq & \mathtt{J} & \leq & \mathtt{I} & \leq & \mathtt{10} \\
\mathtt{2} & \leq & \mathtt{I}+\mathtt{1} & \leq & \mathtt{I}+\mathtt{J} & \leq & \mathtt{I}+\mathtt{I} & \leq & \mathtt{20} \\
\mathtt{0} & = & \mathtt{I}-\mathtt{I} & \leq & \mathtt{I}-\mathtt{J} & \leq & \mathtt{I}-\mathtt{1} & \leq & \mathtt{9}
\end{array}$$

This example also illustrates the importance of eliminating loop indices in *decreasing order of nesting depth*. If, for example, at each elimination, *all* indices are replaced at once by extremal values defined by the loop bounds until no indices remain, the lower bound of the last expression would be computed as $\mathtt{I}-\mathtt{J} \geq \mathtt{1}-\mathtt{I} \geq -\mathtt{9}$. However, because indices are replaced one at the time, the following simple section results:

$$S_1 = \{(i,j) \in \mathcal{Z}^2 \mid (1,1,2,0)^T \leq \mathcal{M}(i,j)^T \leq (10,10,20,9)^T\}$$

Example: Assume that the following double loop also appears in the program:

```
DO I = 1, 5
  DO J = 1, I-1
    ... A(I,J) ...
  ENDDO
ENDDO
```

As illustrated in figure 5.7, straightforward application of the method of this section yields a simple section which has a non-tight boundary due to the fact that the execution set of the `J`-loop is empty for `I=1`. Therefore, after boundary values have been computed, procedure `refine` is applied to the resulting simple section. For the example, this yields the following simple section:

$$S_2 = \{(i,j) \in \mathcal{Z}^2 \mid (2,1,3,1)^T \leq \mathcal{M}(i,j)^T \leq (5,4,9,4)^T\}$$

Function `overlap` can be used to determine that $S_1 \cap S_2 = \emptyset$, which implies data independence between the two assignment statements. In fact, simple sections were actually intended to enhance data dependence analysis in a more general context and are also used by the sparse compiler for that purpose.

Dealing with Inadmissible Subscripts and Loop Bounds

Although most subscripts and loop bounds are admissible in numerical codes, occasionally the sparse compiler must deal with inadmissible subscripts or loop bounds.

In this case, we can use the fact that the following simple section defines the index set of the whole array that is used to store an $m \times n$ matrix:

$$S = \{(i,j) \in \mathcal{Z}^2 \mid (1,1,2,1-n)^T \leq \mathcal{M}(i,j)^T \leq (m,n,m+n,m-1)^T\} \qquad (5.8)$$

Hence, if we assume that subscript bounds are not violated,[8] then we can use the boundary value of this simple section in all cases where the method of the previous section fails.

If the first subscript of an occurrence of a two-dimensional array is inadmissible, the extremal values of the expressions i, $i+j$, and $i-j$ cannot be determined using the previous method, and the corresponding boundary values are taken from (5.8) instead. Likewise, if the second subscript is inadmissible, the method is not able to determine the extremal values of the expressions j, $i+j$, and $i-j$, but uses the corresponding boundary values of (5.8). However, the method is initiated for expression i, for expression j, or for all expressions if only the first, only the second, or all subscripts are admissible respectively. If during computation of an extremal value for one of these expressions, a lower or upper loop bound is required (because the corresponding coefficient in either (5.6) or (5.7) is nonzero) that is not simple, then the computation of the extremal value is abandoned, and the corresponding boundary value of (5.8) is used instead. After all boundary values are determined in this manner, boundary refinement is applied.

Example: The following occurrences of an enveloping data structure of a 100×100 implicitly sparse matrix have one inadmissible subscript:

```
DO I = 1, 10                DO I = 1, 50
  Z = ...                     DO J = 1, 90
  ... A1(I,Z) ...               ... A2(PV(I),J) ...
ENDDO                       ENDDO
                            ENDDO
```

For S_1, the inequalities $1 \leq i \leq 10$ can be determined by examination of the admissible subscript $i = \texttt{I}$. The other boundary values arise naturally from (5.8) and boundary refinement. Simple section S_2 is computed similarly:

$$\begin{cases} S_1 = \{(i,j) \in \mathcal{Z}^2 \mid (1,1,2,-99)^T \leq \mathcal{M}(i,j)^T \leq (10,100,110,9)^T\} \\ S_2 = \{(i,j) \in \mathcal{Z}^2 \mid (1,1,2,-89)^T \leq \mathcal{M}(i,j)^T \leq (100,90,190,99)^T\} \end{cases}$$

Below, an example with an inadmissible loop bound is given:

```
Z = ...
DO I = 10, Z
  DO J = 1, I
    .. A3(5+I,J) ..
  ENDDO
ENDDO
```

For instance, determining the lower bound of expression $i-j$ for $i = \texttt{5+I}$ and $j = \texttt{J}$ can be done without any knowledge of the loop bounds because $\texttt{5+I-J} \geq \texttt{5+I-I}$. However, computing an upper bound of this expression fails because the loop bound `Z` is inadmissible. Likewise, the lower bound $\sigma_1 = 15$ of $i = \texttt{5+I}$ arises from the simple lower bound `10` of the `I`-loop, whereas determination of an upper bound of this expression fails.

Eventually, the simple section shown below results in which, rather surprisingly, inequality $\texttt{J} \leq \texttt{I} \leq \texttt{95}$ arises from boundary refinement:

$$S_3 = \{(i,j) \in \mathcal{Z}^2 \mid (15,1,16,5)^T \leq \mathcal{M}(i,j)^T \leq (100,95,195,99)^T\}$$

[8] In fact, if one of the boundary values that can be computed is exterior to the corresponding boundary value of this simple section, then a potential subscript violation is reported.

Although satisfactory results are obtained with this method, more improvements could be incorporated. Symbolic manipulations, for instance could be used to improve computations on symbolic terms. For e.g. $i = \texttt{PV(I)} + \texttt{I}$ and $j = \texttt{PV(I)} + \texttt{J}$, expression $i - j$ would evaluate to $\texttt{I} - \texttt{J}$, which could be further analyzed as an admissible subscript. In the prototype sparse compiler, however, both i and j are marked as inadmissible, and $i - j$ is not analyzed any further.

5.4.3 Access Summary Bag

To get a better grip on the effective access patterns associated with an occurrence of an enveloping data structure, these access patterns are represented by an access summary. In general, access summaries require less storage than the representation of access patterns and can be manipulated more efficiently.

Access Summaries

Two important attributes associated with each occurrence of a two-dimensional array in a program are (i) an effective access direction $\vec{x} \in \mathcal{Z}^2$ and (ii) a simple section $X \subseteq \mathcal{Z}^2$. The **access summary** $\overline{x}$ of such an occurrence is a tuple consisting of this simple section and the normalized access direction $\vec{x}^{\,n} \in \mathcal{Z}^2$:

$$\overline{x} = \langle X, \vec{x}^{\,n} \rangle$$

The access summary representation only requires 10 integers (2 for the normalized access direction and 8 for the boundary values of the simple section). In contrast, complete information about the subscript functions and the iteration space is required to represent true and effective access patterns. Moreover, the uniformity of representation in terms of access summaries simplifies most operations, such as tests for overlap, which can be performed on the corresponding simple sections, or tests for equivalence, which can be performed on the simple sections and normalized access directions.

The collection of access summaries associated with all occurrences of the enveloping data structure A of an implicitly sparse matrix A is called the **access summary bag** $\mathcal{X}_A$.

Approximated Access Patterns

An access summary $\overline{x} = \langle X, \vec{x}^{\,n} \rangle$ with a *nonzero* normalized access direction $\vec{x}^{\,n} = (x_1^n, x_2^n)^T$ gives rise to a number of **approximated access patterns**. As illustrated in figure 5.8, each approximated access pattern $\mathcal{AP}_k$ consists of all points in the simple section that are along a straight line with the direction $\vec{x}^{\,n} \in \mathcal{Z}^2$, where $k \in \mathcal{Z}$:

$$\mathcal{AP}_k = \{(i,j) \in X \mid x_2^n \cdot i - x_1^n \cdot j = k\} \tag{5.9}$$

The **summary constants** of the access summary $\overline{x}$ are defined as the maximum value $\mathcal{L}(\overline{x}) \in \mathcal{Z}$ and the minimum value $\mathcal{U}(\overline{x}) \in \mathcal{Z}$ for which the following constraint is still satisfied:

$$\mathcal{AP}_k \neq \emptyset \quad \Rightarrow \quad \mathcal{L}(\overline{x}) \leq k \leq \mathcal{U}(\overline{x})$$

Hence, $\overline{x}$ gives rise to $\mathcal{U}(\overline{x}) - \mathcal{L}(\overline{x}) + 1$ approximated access patterns that form a partition of the corresponding simple section. In this manner, we obtain a conservative and uniform representation of the *effective* access patterns of an occurrence of a two-dimensional array (viz. $\vec{x}^{\,n} \neq \vec{0}$), in which the distinction between partially overlapping or multiple traversed access patterns vanishes.

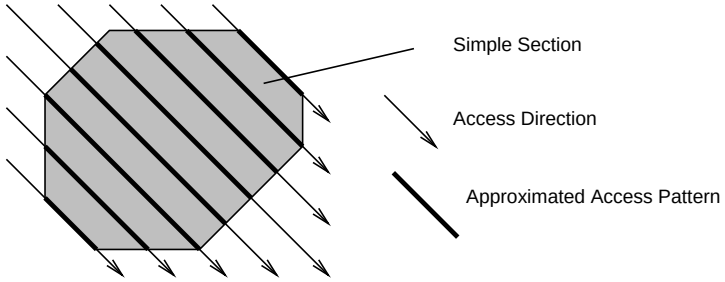


Figure 5.8: Approximated Access Patterns

For each individual effective access pattern $P \subseteq \mathcal{Z}^2$, there is an approximated access pattern that forms a *longitudinal enveloping* access pattern $\mathcal{AP} \supseteq P$ of this access pattern, i.e. $\mathcal{AP}$ consists of all discrete points lying on an arbitrary line segment placed over P.
Example: Consider the following access patterns:

$$P = \{(1,1),(3,5),(5,9)\}$$

As illustrated in figure 5.9, $\mathcal{AP} = \{(1,1),(2,3),(3,5),(4,7),(5,9),(6,11)\}$ forms a longitudinal enveloping access pattern of P.

Computation of the Summary Constants

Given an access summary $\overline{x} = \langle X, \vec{x}^{\,n} \rangle$ with $\vec{x}^{\,n} \neq \vec{0}$, the summary constants $\mathcal{L}(\overline{x}) \in \mathcal{Z}$ and $\mathcal{U}(\overline{x}) \in \mathcal{Z}$ of this access summary are equal to respectively the minimum and maximum value of the following expression for $(i,j) \in S$:

$$x_2^n \cdot i - x_1^n \cdot j \tag{5.10}$$

If $\vec{\sigma} \in \mathcal{Z}^4$ and $\vec{\tau} \in \mathcal{Z}^4$ denote the boundary values of the simple section $X \subseteq \mathcal{Z}^2$, then $(i,j) \in \mathcal{Z}^2$ is subject to the following linear inequalities:

$$\vec{\sigma} \leq \mathcal{M}(i,j)^T \leq \vec{\tau}$$

In general, the extremal values of expression (5.10) can be obtained as follows. First, we use the extended completion method to construct a unimodular 2×2 matrix U with the vector $(x_2^n, -x_1^n)$ as first row and the corresponding inverse U^{-1} (note that the components of $\vec{x}^{\,n} \in \mathcal{Z}^2$ are relatively prime). This matrix is used to establish the following correspondence between i and j and two integer variables k and l:

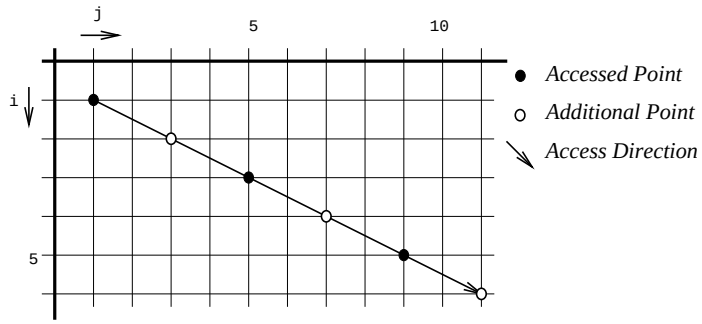


Figure 5.9: Longitudinal Enveloping Access Pattern

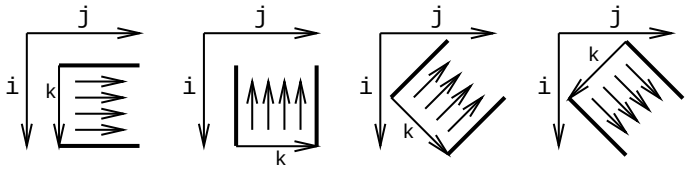


Figure 5.10: Regular Access Patterns

$$\begin{pmatrix} k \\ l \end{pmatrix} = U \begin{pmatrix} i \\ j \end{pmatrix}$$

The extremal integer values of k are obtained by eliminating variable l from the following system of linear inequalities using Fourier-Motzkin elimination:

$$\vec{\sigma} \leq \mathcal{M}U^{-1}(k,l)^T \leq \vec{\tau}$$

If the boundaries of the simple section are tight and the access patterns are regular, i.e. parallel to one of the boundary pairs (viz. figure 5.10), then the extremal values of expression (5.10) are directly defined by the boundary values of that boundary pairs, as shown in the following table for the four possible normalized directions of regular access patterns:

$\vec{x}^{\,n}$	$(0,1)^T$	$(-1,0)^T$	$(-1,1)^T$	$(1,1)^T$
$\mathcal{L}(\overline{x})$	σ_1	σ_2	σ_3	σ_4
$\mathcal{U}(\overline{x})$	τ_1	τ_2	τ_3	τ_4

This table provides an inexpensive method to obtain the summary constants of access summaries with regular access patterns, which are very likely to occur most frequently in numerical programs. For all other access patterns, one step of Fourier-Motzkin elimination is required.

Examples of Access Summaries

In this section, some examples of access summaries are given. Although most access patterns encountered in numerical applications are identical to the corresponding approximated access patterns, the more uniform representation in terms of an access summary may result in some loss of accuracy.

Example: Consider the following occurrence of a two-dimensional array A, where $\vec{\mathtt{I}} = (\mathtt{I}, \mathtt{J}, \mathtt{K})^T$:

```
DO I = 1, 4
  DO J = 1, 2
    DO K = 1, 2
      A(2*I+J-2,I) = ...
    ENDDO
  ENDDO
ENDDO
```

$$F(\vec{\mathtt{I}}) = \begin{pmatrix} -2 \\ 0 \end{pmatrix} + \begin{pmatrix} 2 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \vec{\mathtt{I}}$$

Although the true access direction is zero, the effective access direction of this occurrence is $\vec{x} = (1,0)^T$, giving rise to the following column-wise effective access patterns for $1 \leq \mathtt{I} \leq 4$:

$$P(\mathtt{I}) = \{(2 * \mathtt{I} + \mathtt{J} - 2, \mathtt{I}) \mid 1 \leq \mathtt{J} \leq 2\}$$

The following simple section with 20 discrete points, illustrated in figure 5.11, is associated with this occurrence:

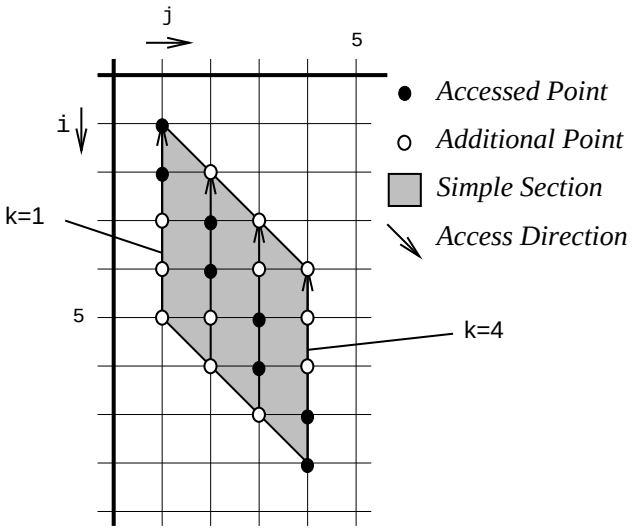


Figure 5.11: Approximated Access Patterns ($\mathcal{L}(\overline{x}) = 1, \mathcal{U}(\overline{x}) = 4$)

$$X = \{(i,j) \in \mathcal{Z}^2 \mid (1,1,2,0)^T \leq \mathcal{M}(i,j)^T \leq (8,4,12,4)^T\}$$

Some additional discrete points have been included because the index set of the accessed part of the array cannot be described exactly in terms of a simple section. Since the normalized access direction is $\vec{x}^{\,n} = (-1,0)^T$, we obtain the access summary $\overline{x} = \langle X, (-1,0)^T \rangle$. Moreover, the second boundary pair defines the summary constants $\mathcal{L}(\overline{x}) = 1$ and $\mathcal{U}(\overline{x}) = 4$. Hence, the access summary gives rise to the following approximated access patterns $\mathcal{AP}_k$, where $1 \leq k \leq 4$:

$$\mathcal{AP}_k = \{(i,j) \in X \mid j = k\}$$

As can be seen in figure 5.11, each approximated access pattern forms a longitudinal enveloping access pattern of one of the effective access patterns. For instance, $P(4)$ is a subset of the approximated access pattern $\mathcal{AP}_4$:

$$\{(7,4),(8,4)\} \subseteq \{(4,4),(5,4),(6,4),(7,4),(8,4)\}$$

Example: In the following triple loop, some of the row-wise true access patterns of the occurrence of array B are traversed multiple times:

```
DO I = 1, 3
  DO J = 1, 2
    DO K = 1, I+J
      B(I+J,K) = ...
    ENDDO
  ENDDO
ENDDO
```

$$F(\vec{\mathtt{I}}) = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \vec{\mathtt{I}}$$

These true access patterns have the following form for $1 \leq \mathtt{I} \leq 3$ and $1 \leq \mathtt{J} \leq 2$:

$$P(\mathtt{I},\mathtt{J}) = \{(\mathtt{I}+\mathtt{J},\mathtt{K}) \mid 1 \leq \mathtt{K} \leq \mathtt{I}+\mathtt{J}\}$$

The simple section associated with the occurrence of array B consists of 14 discrete points:

$$X = \{(i,j) \in \mathcal{Z}^2 \mid (2,1,3,0)^T \leq \mathcal{M}(i,j)^T \leq (5,5,10,4)^T\}$$

The first boundary pair of this simple section defines the summary constants $\mathcal{L}(\overline{x}) = 2$ and $\mathcal{U}(\overline{x}) = 5$ of the access summary $\overline{x} = \langle X, (0,1)^T \rangle$. As illustrated in figure 5.12, this access summary gives rise to the following approximated access patterns $\mathcal{AP}_k$, where $2 \leq k \leq 5$, forming a uniform representation of the multiple traversed true (and, hence, effective) access patterns:

$$\mathcal{AP}_k = \{(i,j) \in X \mid i = k\}$$

Example: Consider the following occurrence of a two-dimensional array C:

```
DO I = 1, 3
  DO J = 1, 3
    DO K = 1, 3
      C(I+K,J+2*K) = ...
    ENDDO
  ENDDO
ENDDO
```

$$F(\vec{\mathtt{I}}) = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 2 \end{pmatrix} \vec{\mathtt{I}}$$

For $1 \leq \mathtt{I} \leq 3$ and $1 \leq \mathtt{J} \leq 3$, the diagonal-wise true access patterns $P(\mathtt{I},\mathtt{J}) \subseteq \mathcal{Z}^2$ of this occurrence have the following form:

$$P(\mathtt{I},\mathtt{J}) = \{(\mathtt{I}+\mathtt{K}, \mathtt{J}+2*\mathtt{K}) \mid 1 \leq \mathtt{K} \leq 3\}$$

The access summary of this occurrence is $\overline{x} = \langle X, (1,2)^T \rangle$, where $X \subseteq \mathcal{Z}^2$, illustrated in figure 5.13, consists of 29 discrete points:

$$X = \{(i,j) \in \mathcal{Z}^2 \mid (2,3,5,-5)^T \leq \mathcal{M}(i,j)^T \leq (6,9,15,1)^T\}$$

The access summary gives rise to a number of approximated access patterns $\mathcal{AP}_k$, consisting of all points in X that are along a straight line with the direction $(1,2)^T$:

$$\mathcal{AP}_k = \{(i,j) \in X \mid 2 \cdot i - j = k\}$$

The summary constants $\mathcal{L}(\overline{x}) \in \mathcal{Z}$ and $\mathcal{U}(\overline{x}) \in \mathcal{Z}$ of this access summary are equal to the extremal values of the expression $2 \cdot i - j$, where $(i,j) \in X$. The extended completion method yields the following matrices:

$$U = \begin{pmatrix} 2 & -1 \\ 1 & 0 \end{pmatrix} \qquad U^{-1} = \begin{pmatrix} 0 & 1 \\ -1 & 2 \end{pmatrix}$$

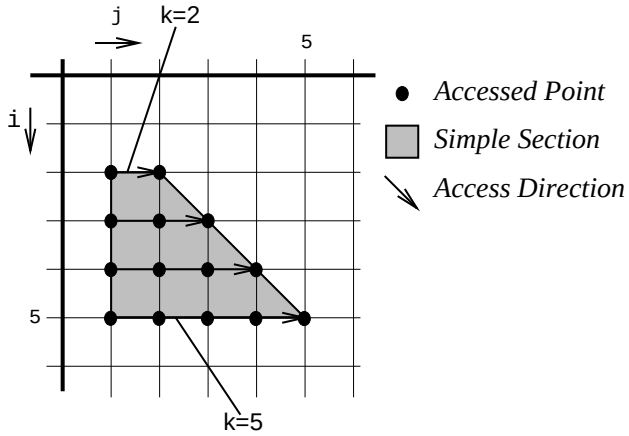


Figure 5.12: Approximated Access Patterns ($\mathcal{L}(\overline{x}) = 2, \mathcal{U}(\overline{x}) = 5$)

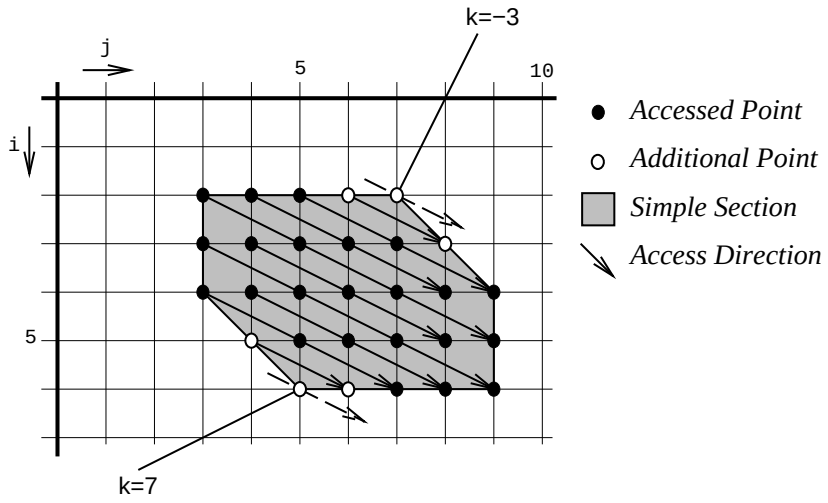


Figure 5.13: Approximated Access Patterns ($\mathcal{L}(\overline{x}) = -3, \mathcal{U}(\overline{x}) = 7$)

Thereafter, we obtain $\mathcal{L}(\overline{x}) = -3$ and $\mathcal{U}(\overline{x}) = 7$ by application of Fourier-Motzkin elimination to the system of inequalities that is obtained by replacing (i, j) with $U^{-1}(k, l)^T$ in the boundaries of the simple section X:

$$\begin{array}{rcrcrcr} 2 & \leq & & & l & \leq & 6 \\ 3 & \leq & -k & + & 2 \cdot l & \leq & 9 \\ 5 & \leq & -k & + & 3 \cdot l & \leq & 15 \\ -5 & \leq & -k & - & l & \leq & 1 \end{array} \quad \begin{array}{c} \text{Elimination} \\ \text{of variable } l \\ \longrightarrow \end{array} \quad -3 \leq k \leq 7$$

Consequently, as illustrated in figure 5.13, the simple section is partitioned into 11 approximated access patterns, where the distinction between partially overlapping access patterns vanishes. For each true (and effective) access pattern $P(\mathtt{I}, \mathtt{J})$, there is a longitudinal enveloping approximated access pattern. For example, the approximated access pattern $\mathcal{AP}_1$ forms a longitudinal enveloping access pattern of both the true access patterns $P(1, 1)$ and $P(2, 3)$:

$$\left.\begin{array}{l} \{(2,3),(3,5),(4,7)\} \\ \{(3,5),(4,7),(5,9)\} \end{array}\right\} \subseteq \{(2,3),(3,5),(4,7),(5,9)\}$$

Example: Consider, as final example, the following double loop, where $\vec{\mathtt{I}} = (\mathtt{I}, \mathtt{J})^T$:

```
DO I = 1, 3
  DO J = 1, 3
    D(4*J-3,2*I-2*J+5) = ...
  ENDDO
ENDDO
```

$$F(\vec{\mathtt{I}}) = \begin{pmatrix} -3 \\ 5 \end{pmatrix} + \begin{pmatrix} 0 & 4 \\ 2 & -2 \end{pmatrix} \vec{\mathtt{I}}$$

The occurrence of the two-dimensional array D has the following diagonal-wise true access patterns for $1 \leq \mathtt{I} \leq 3$:

$$P(\mathtt{I}) = \{(4 * \mathtt{J} - 3, 2 * \mathtt{I} - 2 * \mathtt{J} + 5) \mid 1 \leq \mathtt{J} \leq 3\}$$

Although actually only 9 elements are accessed, the simple section that is associated with this occurrence consists of 61 points:

$$X = \{(i, j) \in \mathcal{Z}^2 \mid (1, 1, 6, -8)^T \leq \mathcal{M}(i, j)^T \leq (9, 9, 14, 8)^T\}$$

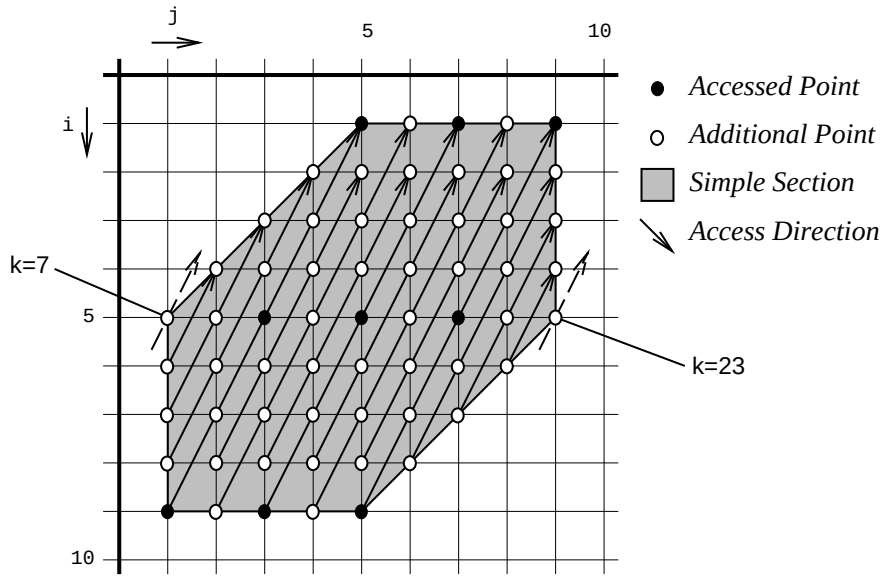


Figure 5.14: Approximated Access Patterns ($\mathcal{L}(\overline{x}) = 7, \mathcal{U}(\overline{x}) = 23$)

The effective access direction of this occurrence is $\vec{x} = (4, -2)^T$. Hence, the resulting access summary $\overline{x} = \langle X, (-2, 1)^T \rangle$ gives rise to the following approximated access patterns $\mathcal{AP}_k$:

$$\mathcal{AP}_k = \{(i, j) \in X \mid i + 2 \cdot j = k\}$$

The summary constants consist of the extremal values of the expression $i + 2 \cdot j$ for $(i, j) \in X$. First, we use the extended completion method to obtain the unimodular matrices U and U^{-1}, where U has $(1, 2)$ as first row:

$$U = \begin{pmatrix} 1 & 2 \\ 0 & 1 \end{pmatrix} \qquad U^{-1} = \begin{pmatrix} 1 & -2 \\ 0 & 1 \end{pmatrix}$$

Thereafter, we obtain $\mathcal{L}(\overline{x}) = 7$ and $\mathcal{U}(\overline{x}) = 23$ by applying one step of Fourier-Motzkin elimination to the system obtained by replacing (i, j) by $U^{-1}(k, l)^T$:

$$\begin{array}{rcrcrcr} 1 & \leq & k & - & 2 \cdot l & \leq & 9 \\ 1 & \leq & & & l & \leq & 9 \\ 6 & \leq & k & - & l & \leq & 14 \\ -8 & \leq & k & - & 3 \cdot l & \leq & 8 \end{array} \qquad \begin{array}{c} \text{Elimination} \\ \text{of variable } l \\ \longrightarrow \end{array} \qquad 7 \leq k \leq 23$$

As illustrated in figure 5.14, quite some additional discrete points are included in X, due to the discrepancy between the index set of the accessed part in the array and a simple section. Consequently, additional approximated access patterns appear in between and next to the true (and effective) access patterns.

In fact, 17 approximated access patterns results as representation of the three true access patterns. However, as stated before, in general an access summary provides a sufficiently accurate representation for most effective access patterns that are encountered in numerical programs.

Chapter 6

Nonzero Structure Analysis

Because the efficiency of sparse codes is very much dependent on the size and structure of the input data, peculiarities of the nonzero structure of each sparse matrix must be accounted for in order to avoid unsatisfactory performance. Therefore, an important part of the sparse compiler consists of an analyzer that obtains some characteristics of the nonzero structure automatically. This information is used to control the data structure selection and sparse code generation. Since analysis time contributes to compile-time, the efficiency of the analyzer is important.

A nonzero structure analyzer is also useful for other purposes. For example, if a representative set of sparse matrices is available beforehand, an analyzer can provide a programmer with useful insights about the characteristics of the matrices for which an application must be developed. Although in this case the efficiency of the analyzer is less important, excessive long running times would disable the analysis of a large set of matrices. To deal with the more realistic situation in which the sparse matrices are not available beforehand, an analyzer can be used at run-time to select between different versions of one algorithm (which are generated either by a sparse compiler or by hand), each of which has been optimized for a particular class of nonzero structures. At run-time, the analyzer is invoked to determine which version is probably the most efficient. This approach has as major advantage that nonzero structures do not have to be known at programming- or compile-time. However, the analyzer must be very efficient to avoid the situation in which the savings in execution time using an optimized version are outweighed by analysis time. In general, it is desirable to keep analysis time proportional to the number of entries in the matrix [75].

The analyzer presented in this chapter examines each matrix as it is, i.e. no attempts are made to permute the matrix into a particular form. If a permutation is applied before the analysis, the analyzer can still be used to determine whether an unforeseen nonzero structure arises (since information about the form for which the permutation is intended is usually obtained as side effect by the method that computes the permutation). First, some methods to automatically analyze the nonzero structure of a sparse matrix are discussed. Thereafter, we discuss how nonzero structure information is propagated to the sparse compiler.

6.1 Automatic Nonzero Structure Analysis

We assume that the nonzero structure of each $m \times n$ sparse matrix A to be analyzed is available on file in coordinate scheme. In this scheme, the file consists of the integers m and n, an integer τ that indicates the number of entries, followed by τ unordered triples (i, j, a_{ij}) to indicate row and column indices and the value of each individual entry. Because there is no advantage in storing zero elements explicitly, we also assume that all entries are nonzero.

6.1.1 Preparatory Analysis

In subsequent sections, we will see that many nonzero structures can be determined efficiently from skyline and used-diagonal information only. This information can be obtained in a *single* pass over the stored nonzero elements in a file by execution of the following fragment presented in pseudo-code, where the lower and upper skyline are computed in the arrays `lsky` and `usky` respectively, and used-diagonal information in array `diagc`:

```
procedure comp_skylines()
begin
  read(n, m, nnz);
  N := max(m, n);

  allocate lsky[1:N] and usky[1:N]
  for i := 1, N do
    lsky[i] := 0;
    usky[i] := 0;
  enddo

  allocate diagc[1-N:N-1]
  for i := 1-N, N-1 do
    diagc[i] := 0;
  enddo

  for k := 1, nnz do
     read(i, j, aij);
     lsky[i]    := max(lsky[i], (i-j));
     usky[j]    := max(usky[j], (j-i));
     diagc[i-j] := diagc[i-j] + 1;
  enddo
end
```

For sake of simplicity, the current implementation of the analyzer handles each arbitrary $m \times n$ matrix A as a square $N \times N$ matrix with $N = \max(m, n)$, as illustrated in figure 6.1. Moreover, skylines are computed under the assumption that this matrix has a full transversal, so that all elements of the arrays `lsky` and `usky` can be initialized to zero. Obviously, all information requires $O(N)$ storage and can be obtained in $O(\tau + N)$ time.

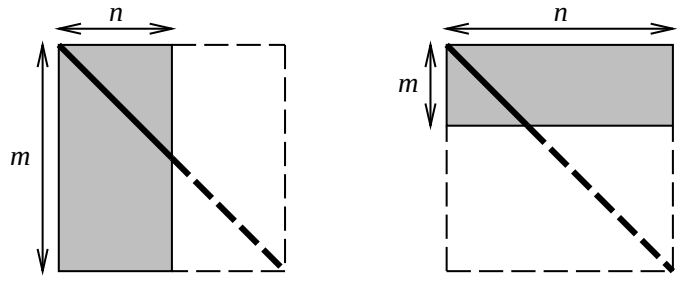


Figure 6.1: Rectangular Matrices

Example: The following lower and upper skyline are obtained for the 15×15 sparse matrix that is depicted in figure 6.2:

	1														15
`lsky`	0	0	1	0	4	0	1	0	1	0	0	0	0	0	3
`usky`	0	0	2	0	1	0	0	0	3	0	0	0	0	3	2

Part of the contents of array `diagc` for this matrix are shown below:

		-5					0					5	
`diagc`	...	0	0	2	2	2	15	3	0	1	1	0	...

6.1.2 Some Nonzero Structures

Skyline information directly defines the variable band form of a matrix. However, this information can also be used to obtain other characteristics of the nonzero structure in an efficient way.

Band Forms

Once the lower and upper skyline of a matrix have been computed, the lower and upper semi-bandwidth of this matrix are determined in $O(N)$ time as follows:

```
procedure semi_bandwidths()
begin
  b_l := 0; b_u := 0;
  for i := 1, N do
    b_l := max(b_l, lksy[i]);
    b_u := max(b_u, uksy[i]);
  enddo
end
```

These semi-bandwidths directly determine the band form of a matrix. Special classes of band matrices are formed by diagonal matrices ($b_l = 0$ and $b_u = 0$), tridiagonal matrices ($b_l = 1$ and $b_u = 1$), and upper and lower triangular matrices (either $b_l = 0$ or $b_u = 0$ but not both). Furthermore, if $1 < b_u \ll N - 1$ or $1 < b_l \ll N - 1$ holds for an upper or lower triangular matrix respectively, we may say that the matrix is band upper or band lower triangular.
Example: For the matrix of figure 6.2, the semi-bandwidths $b_l = 4$ and $b_u = 3$ result, which gives rise to the band form shown in the same figure.

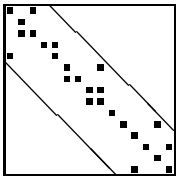


Figure 6.2: Band Form

Block Diagonal and Block Triangular Forms

If during construction of a block partition into block diagonal form, diagonal blocks beyond row and column B already have been identified, the next diagonal block may be of size k if the following constraint is satisfied, where l_i and u_i denote elements of the skylines:

$$\forall B - k < i \leq B : \quad \max(l_i, u_i) + (B - i) < k$$

If this constraint is violated, then an appropriate change to the size of the next diagonal block is required. This observation, illustrated in figure 6.3, gives rise to the following algorithm to construct a block partition into diagonal block form in $O(N)$ time:

```
procedure comp_blockdiag()
begin
  p := 0; k := 1; B := N;
  for i := N, 1, -1 do
    k := max(k, max(lsky[i],usky[i])+B-i+1);          ← (*)
    if (i = B-k+1) then
       p := p + 1;  part[p] := i;  /* Record Block */
       B := i - 1;        k := 1:  /* Next Block   */
    endif
  enddo
end
```

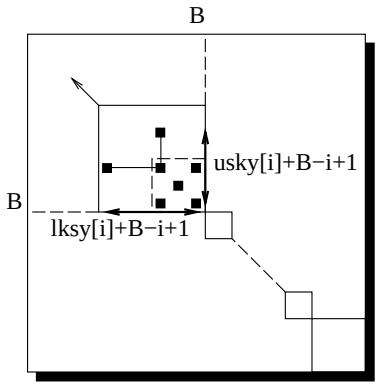


Figure 6.3: Valid Block in Block Diagonal Form

After application of this algorithm, `p` contains the number of diagonal blocks. The row (or column) indices of the upper left corners of all diagonal blocks of the block partition are recorded in reverse order in the first `p` locations of array `part`. The following proposition states that the minimum block partition into block diagonal form is found. Likewise, if only the value `lsky[i]` or `usky[i]` is used in statement $(*)$, then the minimum block partition into respectively block lower, or block upper triangular form is obtained in $O(N)$ time.

Proposition 6.1 *Application of* `comp_blockdiag()` *to the lower and upper skyline of a matrix yields the minimum block partition into block diagonal form.*

PROOF By construction each entry is incorporated in a diagonal block. Now, assume that the resulting block partition is not a minimum block partition into diagonal form. Then, proposition 4.2 implies that there is a certain $k \times k$ diagonal block with the lower right corner at a row index B that has a non-trivial block partition into block diagonal form, i.e. there is $k' < k$ such that the following constraint holds:

$$\forall B - k' < i \leq B : \max(l_i, u_i) + (B - i) < k'$$

Since no diagonal block is recorded at any of the iterations `i` $= B$ through `i` $= B - k' + 1$, during at least one of these iterations, a value is assigned to `k` that is greater than k'. However, this can only occur if $\max(l_i, u_i) + B - i + 1 > k'$ for some $B - k' < i \leq B$, which contradicts the assumption. □

Example: Application of these different versions of the algorithm to the matrix of figure 6.2 yields the block diagonal, block lower and upper triangular form shown in figure 6.4. The contents of array `part` for the first block partition, for instance, is shown below:

part	11	10	6	1

The total number of elements contained in the nonzero blocks that belong to these block forms (67, 140 and 138 respectively) can be used to determine which of the forms describes the nonzero structure most accurately. In this case, the block diagonal form reveals the most information about the nonzero structure of the matrix. This block form also provides a more accurate description of the nonzero structure than the band form shown in figure 6.2, in which 104 elements reside.

Bordered Block Forms

If some nonzero elements appear in the borders of a matrix, very large diagonal blocks may occur in the minimum block partition into a particular block form.

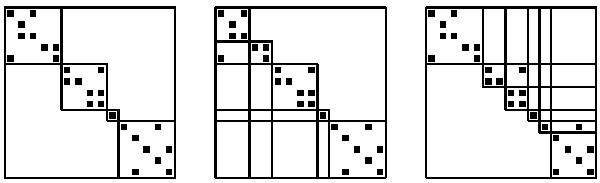


Figure 6.4: Block Forms

For the matrix in figure 6.5, for example, 176 elements appear in the nonzero blocks of the minimum block partition into block upper triangular form. Since only the trivial block partition defines a block diagonal or block lower triangular form, the block partitions into these forms contain even more elements.

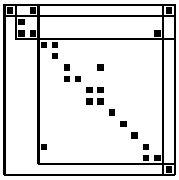


Figure 6.5: Block Upper Triangular Form

Therefore, it may be useful to construct a minimum block partition into *bordered* block diagonal or triangular form. In a naive approach, we could apply the method of the previous section to the remaining sub-matrix for each possible border size, followed by a selection of the block partition with the fewest elements in the nonzero blocks (border blocks included). However, this approach would have an $O(N^2)$ complexity, which is unacceptable for the analysis of a sparse matrix [75]. Fortunately, it is also possible to obtain the best border size in $O(N)$ time, as explained below for doubly bordered block diagonal forms.

Let $E(b)$ denote the number of elements in the nonzero blocks of block partition into *bordered* block diagonal form with border size $b \in [0, N]$ arising from the minimum block partition of the remaining $(N - b) \times (N - b)$ matrix into block diagonal form. We define the improvement of using border size b' instead of b as $I(b', b) = E(b) - E(b')$, satisfying the following property:

Proposition 6.2 *For $b, b', b'' \in [0, N]$, we have $I(b', b'') = I(b', b) + I(b, b'')$*

PROOF $I(b', b'') = E(b'') - E(b') = E(b) - E(b') + E(b'') - E(b) = I(b', b) + I(b, b'')$ []

Now, suppose that for a given border size $b \in [0, N]$, we construct the minimum block partition of the remaining $(N-b) \times (N-b)$ sub-matrix into block diagonal form using the procedure `comp_diagblock()`. At any iteration $\texttt{i} = i$, we may decide to discard the block partition found so far, and to start the algorithm with $B = i - 1$ and $k = 1$ for a new border size $b' = N - i + 1$.

Obviously, selection of this border is only profitable if eventually we are able to determine that $I(b', b) > 0$. However, rather than constructing both block partitions completely, we are already able to compute the improvement during an iteration $\texttt{i} = i'$ in which the last diagonal block of the new block partition that overlaps with the diagonal block that was assumed during iteration $\texttt{i} = i$ has been found. This is *because the block partition of the remaining part of the matrix will be identical for both block partitions.* This new diagonal block may be contained in the old diagonal block (which occurs if the value of `k` would not have been incremented while computing the old block partition), or these blocks may may partially overlap.

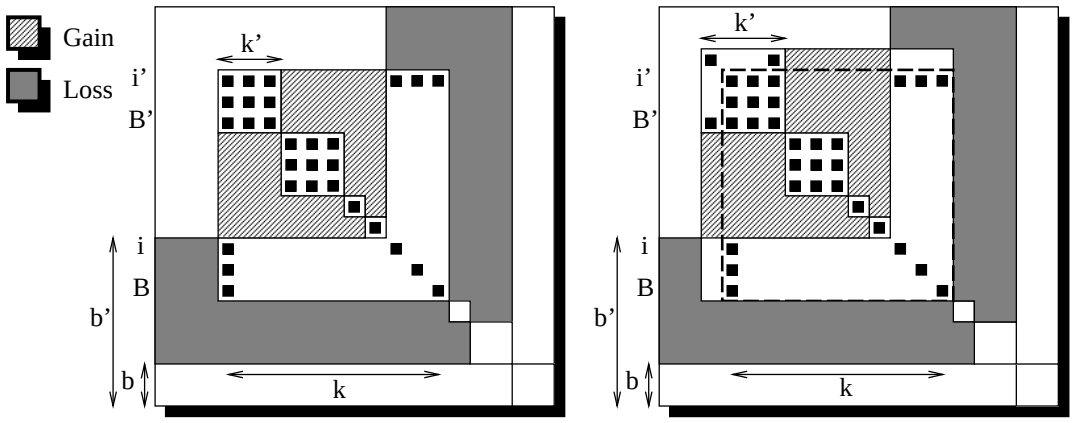


Figure 6.6: Gain and Loss for Border

Both cases are illustrated in figure 6.6. In any case, the improvement is equal to the difference of the number of elements included in the border (*loss*) and the number of elements that do not have to be included in a diagonal block (*gain*). Let B, k and B', k' denote the value of B belonging to the iterations i $= i$ and i $= i'$ respectively. Furthermore, let Z and Z' denote the number of elements in the off-diagonal blocks of the old and new block partition below row B and B', respectively. Then, the improvement of using a new border size b' with respect to the old border size b is equal to the difference between the gain and the loss:

$$I(b', b) \quad = \quad Z' - Z - 2 \cdot (B' - k')(B - B')$$

If the gain exceeds the loss, i.e. $I(b', b) > 0$, then it is profitable to continue with the new block partition and border size b'. Moreover, border size b may be discarded, since proposition 6.2 implies that $I(b', b'') > I(b, b'')$ for all $b'' \in [0, N]$. If no improvement has been obtained, i.e. $I(b', b) \leq 0$, then the block partition corresponding to border size b must be restored and the algorithm can proceed with the search for the next diagonal block (which minimally is of size $\max(k, B - B' + k')$ now). In that case, we may discard border size b', since proposition 6.2 implies that $I(b', b'') \leq I(b, b'')$ for all $b'' \in [0, N]$.

These observations enable us to compute a minimum block partition into block diagonal form in one pass over the skylines. At each step in which no diagonal block is recorded, the current status is saved on a stack, and a new border size is tried. If a diagonal block is recorded, no improvement can be obtained by trying a new border size. Instead, previously constructed block partitions belonging to smaller border sizes that can be verified are restored if an improvement is obtained (which is simply done by restoring the value of p), or discarded otherwise. The following slightly more complex version of procedure comp_blockdiag() results:

```
procedure comp_bord_blockdiag()
begin
  Z := 0; b := 0; s := 0;
  p := 0; k := 1; B := N;
  for i := N, 1,  -1 do
    k := max(k, max(lsky[i],usky[i])+B-i+1);              ← (+)
    if (i = B-k+1) then
       /* Last Overlapping Block? */
       while ( (s > 0) && (i == stackB[s]-new_k()+1) ) do /* Conditional AND */
          /* Improvement? */
          if (I() > 0) then
            s := s - 1;            /* Discard         */
          else
            pop_restore();         /* Restore         */
          endif
       enddo
```

```
      Z := Z + 2 * k * (B-k);     /* #Elts in Border */
      p := p + 1;  part[p] := i;  /* Record Block    */
      B := i - 1;        k := 1:  /* Next Block      */
    else
      stack();                    /* Save State      */
      Z := 0; B := i - 1;         /* New Search      */
      k := 1; b := N - i + 1;
    endif
  enddo
end
```

In this algorithm, the following auxiliary routines are used to implement stack-like operations that save and restore states:

```
procedure push()
begin
  s := s + 1;
  stackk[s] := k;
  stackZ[s] := Z;
  stackB[s] := B;
  stackp[s] := p;
  stackb[s] := b;
end
```

```
procedure pop_restore()
begin
  k := new_k();
  Z := stackZ[s];
  B := stackB[s];
  p := stackp[s];
  b := stackb[s];
  s := s - 1;
end
```

The following auxiliary functions are used to compute the improvement and the new value of `k` for the block partition on top of the stack:

```
integer function I()
begin
 I := Z - stackZ[s] - 2 *
      (B-k) * (stackB[s]-B);
end
```

```
integer function new_k()
begin
  new_k := max(stackk[s],
               stackB[s]-B+k);
end
```

Although a while-loop occurs inside the `i`-loop, this algorithm still runs in $O(N)$ time because each border size can only be pushed and popped from the stack once. Because the algorithm simply applies `comp_blockdiag()` to the sub-matrix that remains for the most profitable border size, it is clear that this adapted algorithm constructs a minimum block partition into bordered block diagonal form.

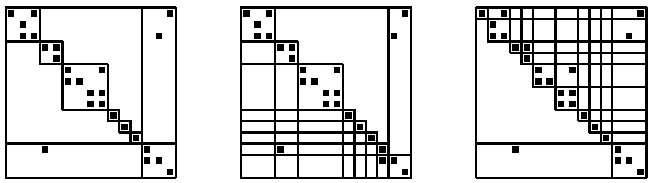
After application of this algorithm, the scalar `b` contains the selected border size (and hence the size of the last diagonal block). The first `p` locations of array `part` represent the block partition into block diagonal form of the remaining sub-matrix. If a zero border size is selected, the last diagonal block is empty and a block partition into block diagonal form results.

If only the value `lsky[i]` or `usky[i]` is used in statement $(+)$, then a minimum block partition into respectively (singly) bordered block lower or upper triangular form is obtained. In these cases, the constant 2 must be removed from the assignment to `Z` and the computation in function `I()` to compute the appropriate improvement.

Example: In figure 6.7, the bordered block forms that result for the matrix of figure 6.5 are shown, containing respectively 113 (viz. $225-112$), 157 (viz. $225-68$), and 162 (viz. $176-14$) elements. The contents of array `part` for the bordered block diagonal form having `b=3` is shown below:

part	12	11	10	6	4	1

Example: Applying the version only operating on `lsky[i]` to the matrix with 22 nonzero elements of figure 6.8 yields a minimum partition into bordered block upper triangular form with border size 1. However, the nonzero blocks of the partitions corresponding to border sizes 2 and 3 contain the same number of elements, namely 166. This example illustrates that a matrix may have different minimum block partitions into a particular bordered block form. Because a border is denied for a zero improvement (viz. $I(3,2) = 0$ and $I(2,1) = 0$ in the example, so that according to proposition 6.2, we have $I(3,0) = I(2,0) = 21$), ties are solved in favor of the smallest border size.



$I(3,0) = 112$ $I(2,0) = 68$ $I(3,0) = 14$

Figure 6.7: Bordered Block Forms

Multi-Diagonal Form

The used-diagonal information computed in array `diagc` enables the analyzer to compute some band related information rather easily [185]. For example, the percentage p of nonzero elements that are confined in a band with lower and upper semi-bandwidth b'_l and b'_u is computed as follows:

$$p = \frac{100\%}{\tau} \cdot \sum_{k=-b'_u}^{b'_l} \texttt{diagc}[k]$$

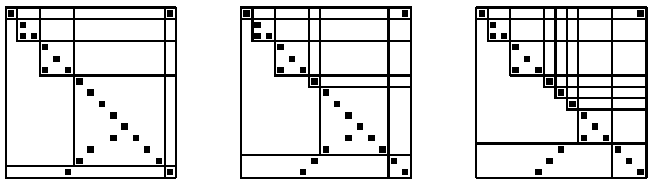
On the other hand, the smallest band with bandwidth $2 \cdot b + 1$ in which, for instance, 90% of the nonzero elements are contained, is given by the smallest $b \geq 0$ for which the following inequality is satisfied:

$$\sum_{k=-b}^{b} \texttt{diagc}[k] \quad \geq \quad 0.9 \cdot \tau$$

Array `diagc` also enables the analyzer to compute the number of used diagonals in a matrix by simply counting the number of nonzero elements in this array, which requires $O(N)$ time. If all entries appear along relatively few diagonals, then the matrix can be classified as a **multi-diagonal matrix**. This form is more flexible that the related band form, because it can account for zero regions in between nonzero diagonals. The number of full diagonals is determined by counting the number of diagonals with density 1, where the density of the kth diagonal is computed as follows:

$$\frac{\texttt{diagc}[k]}{\min(N, N-k) - \max(1, 1-k) + 1}$$

Example: The matrix of figure 6.9 has semi-bandwidths $b_l = 7$ and $b_u = 7$. There are 3 diagonals used (at the locations -7, 0 and 7), of which 2 are actually full. Accumulating for $b'_l = 2$ and $b'_u = 2$ yields $p = 51.72\%$, because 15 of the 29 entries reside in a band with bandwidth 5. Finally, the smallest band in which 90% of the entries is contained has bandwidth 15 (in fact, all entries are contained in this band).



$I(1,0) = 21$ $I(2,0) = 21$ $I(3,0) = 21$

Figure 6.8: Different Minimum Partitions

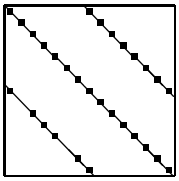


Figure 6.9: Multi-Diagonal Form

Example: Multi-diagonal matrices typically arise in **finite difference methods** (see e.g. [163, 173]). These numerical approximation methods are used to solve partial differential equations arising in the analysis of continuous systems. Many steady-state or equilibrium problems, for example, consist of finding a function $u(x, y)$ over a particular region R, such that the following elliptic equation, referred to as the Poisson equation, or Laplace equation for $\rho(x, y) = 0$, is satisfied for $(x, y) \in R$:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = \rho(x, y) \tag{6.1}$$

Another notation for this equation is $\nabla^2 u = \rho(x, y)$. The behavior of $u(x, y)$ along a boundary B of region R is also specified, which is why the problems are referred to as boundary value problems. For example, so-called Dirichlet conditions define the value of $u(x, y)$ on the boundary explicitly, i.e. $u(x, y) = g(x, y)$ for $(x, y) \in B$. However other kind of conditions are also possible. Since only a few elliptic equations can be solved analytically, approximation methods are used frequently.

In the finite difference method, the value of function $u(x, y)$ is approximated at the discrete points (x_i, y_j) for $x_i = x_0 + h \cdot i$ and $y_j = y_0 + h \cdot j$ forming a grid over region R, where h is referred to as the grid spacing. Based on Taylor expansion, the following finite difference approximations can be used for the derivates in (6.1):

$$\begin{cases} \frac{\partial^2 u}{\partial x^2} & \approx & \frac{u(x+h,y)-2\cdot u(x,y)+u(x-h,y)}{h^2} \\ \frac{\partial^2 u}{\partial y^2} & \approx & \frac{u(x,y+h)-2\cdot u(x,y)+u(x,y-h)}{h^2} \end{cases}$$

If denote the approximation of $u(x_i, y_j)$ in case these two previous two approximations hold exactly by u_{ij}, then we obtain the following five-points finite difference approximation of the Poisson equation at a point (x_i, y_j):

$$u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4 \cdot u_{i,j} = h^2 \cdot \rho(x_i, y_j) \tag{6.2}$$

For the unit square as region, an $M \times N$ interior grid gives rise to a system of $M \cdot N$ linear equations in $M \cdot N$ variables $u_{11}, \ldots, u_{MN}$. On the boundaries, we require the values of u_{0j} and $u_{M+1,j}$ for $j = 1, \ldots, N$, and the values of u_{i0} and $u_{i,N+1}$ for $i = 1, \ldots, M$. These values are defined by the boundary conditions. Representing all interior grid points by an unknown vector $\vec{u}$ according to a page-wise numbering of the variables u_{ij}, gives rise a system of linear equations $A\vec{u} = \vec{b}$, for a sparse matrix A. For a square $N \times N$ grid, this matrix A is an $N^2 \times N^2$ multi-diagonal matrix, as illustrated for $N = 4$ in figure 6.10.

6.1.3 Selection of Best Form

After the different forms have been constructed, the total number of elements contained in the nonzero regions defined by each particular form is used to determine which form provides the most accurate description of the nonzero structure.

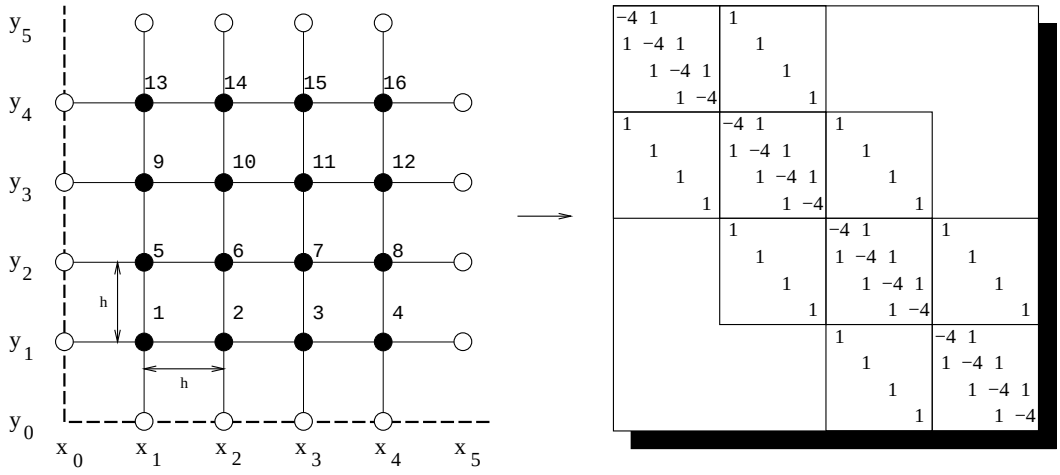


Figure 6.10: Finite Difference Method

Counting the Number of Elements in the Nonzero Regions

For a square $N \times N$ matrix with semi-bandwidths b_l and b_u, the number of elements within the band is given by the following formula:

$$N \cdot (b_l + b_u + 1) - (b_l^2 + b_l)/2 - (b_u^2 + b_u)/2$$

The number of elements in the more general multi-diagonal form are obtained by counting the number of elements in the diagonals that are used in the matrix, which can be done as follows for the set $K = \{k \in [1 - N, N - 1] \mid \texttt{diagc}[k] \neq 0\}$:

$$\sum_{k \in K} \min(N, N - k) - \max(1, 1 - k) + 1$$

If the scalar `b` and the first `p` locations of array `part` describe a bordered block diagonal form or a bordered block triangular form, then the number of elements within the nonzero blocks can be computed using one the following two procedures:

```
integer function cnt_bbd()
begin
  cnt_bbd := 2 * b * N - b * b;
  prv := N - b + 1;
  for i := 1, p do
    cnt_bbd := cnt_bbd
             + (prv-part[i])
             * (prv-part[i]);
    prv := part[i];
  enddo
end
```

```
integer function cnt_bbt()
begin
  cnt_bbt := 2 * b * N - b * b;
  prv := N - b + 1;
  for i := 1, p do
    cnt_bbt := cnt_bbt
             + (prv-part[i])
             * (prv-1);
    prv := part[i];
  enddo
end
```

Classification

First, the analyzer constructs the band form (with (band) lower or upper triangular, diagonal, or tridiagonal form as special classes), minimum block partitions into double bordered block diagonal and singly bordered block lower and triangular of a matrix, and determines how many diagonals appear in the multi-diagonal form. In combination with preparatory analysis, this can be done in $O(N + \tau)$ time. Thereafter, the total number of elements in the nonzero regions of each form is computed in $O(N)$ time as explained in the previous section.

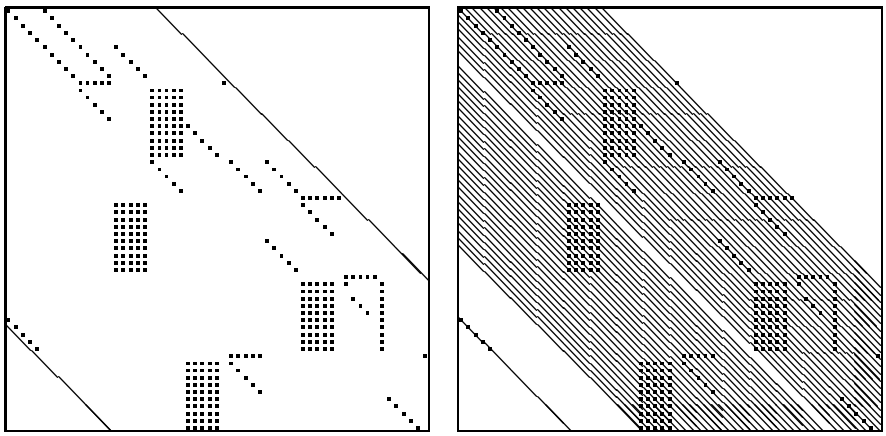


Figure 6.11: Band Form and Multi-Diagonal Form of 'impcol_b'

The form for which this number is minimal, say c, is selected as most representative form, where we prefer a band form over a multi-diagonal form on ties. Only if the actual density within the nonzero regions also satisfies the inequality $\tau/c \geq \rho^t$ where ρ^t is a user-defined threshold, the analyzer uses this representative form in the classification of the matrix. Otherwise, the analyzer classifies the matrix as a general sparse matrix. In this manner, a particular nonzero structure is only used in the classification of a matrix if this structure actually can be exploited to reduce the computational time of an algorithm, rather than on a criterion, for instance, that the matrix is in band form if the semi-bandwidths are relatively small in comparison with the size of the matrix.

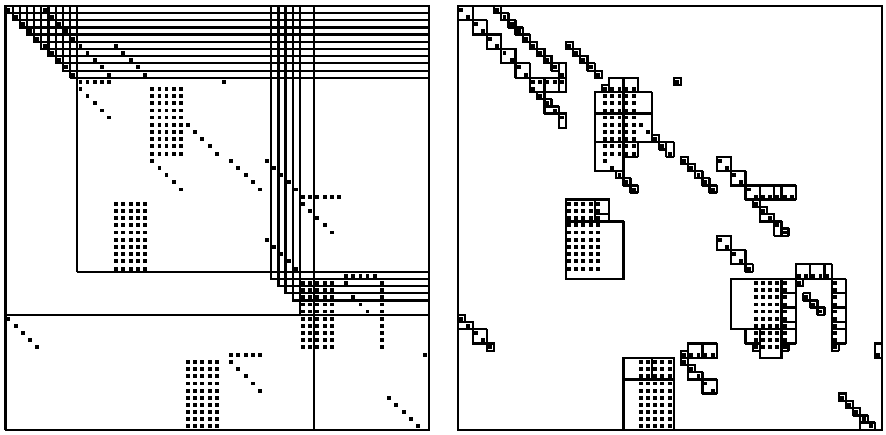


Figure 6.12: Block Form and Dense Sub-Matrices of 'impcol_b' ($\rho^t = 0.5$)

Example: In figure 6.11, the band form and multi-diagonal form of the 59×59 matrix 'impcol_b' of the Harwell-Boeing Sparse Matrix Collection [79] with 312 entries are shown. In the band with semi-bandwidths 43 and 20, 2620 elements reside, whereas only 2379 elements reside in the 54 used diagonals. The nonzero blocks of the bordered block diagonal and block lower and upper triangular form contain respectively 3461, 3206, and 2930 elements. The bordered block upper triangular form is illustrated in the first picture of figure 6.12. Hence, if $312/2379 \geq \rho^t$, the analyzer classifies the matrix as a multi-diagonal matrix, or as a general sparse matrix otherwise.

6.1.4 Dense Sub-Matrices

A more expensive method to obtain some information about the nonzero structure of a matrix is based on the idea behind the quad-tree schemes presented in section 4.11. Dense sub-matrices in an arbitrary sparse $m \times n$ matrix A are detected by recursively partitioning this matrix into sub-matrices. A sub-matrix for which the density exceeds a user-defined threshold ρ^t is treated as a dense matrix, whereas a zero matrix is not considered further. For all other sub-matrices, these criteria are applied recursively to the sub-matrices in four quadrants, as formulated in the following algorithm called as '`partition`$(1, m, 1, n)$':

```
procedure partition(i_low, i_hig, j_low, j_hig)
begin
  X := {(i,j) ∈ Nonz_A|(i_low ≤ i ≤ i_high) ∧ (j_low ≤ j ≤ j_high)};
  if (|X| > 0) then
    frac := |X| /  ( (i_hig-i_low+1) * (j_hig-j_low+1) );
    if (frac < ρ^t) then
      i_mid := ⌊ (i_low + i_hig) / 2 ⌋;
      j_mid := ⌊ (j_low + j_hig) / 2 ⌋;
      partition(i_low,   i_mid, j_low,   j_mid);
      partition(i_low,   i_mid, j_mid+1, j_hig);
      partition(i_mid+1, i_hig, j_low,   j_mid);
      partition(i_mid+1, i_hig, j_mid+1, j_hig);
    else
      record_block(i_low, i_hig, j_low, j_hig);
    endif
  endif
end
```

When carefully coded, the storage requirements can be kept to $O(\tau)$ by performing an in-place sorting of the index set, while pointers into this array are passed as additional parameters to locate the remaining index set for each invocation. The algorithm has an $O(\tau \cdot \log N)$ running time though, where $N = \max(m, n)$.

Example: In the second picture of figure 6.12, the dense sub-matrices that are detected for $\rho^t = 0.5$ in the 59×59 matrix 'impcol_b' of the previous section.

Moreover, in figure 6.13, we present the dense sub-matrices in a 32×32 matrix with 331 entries that result for different values of ρ^t. Although this method captures the dense sub-matrices reasonable well, the placement of sub-matrices is not always optimal because arbitrary divisions of the index set are made. Decreasing the threshold ρ^t partially reduces this problem, but clusters of dense sub-matrices still result in general.

6.2 Nonzero Structure Analyzer

In this section, some issues related to the nonzero structure analyzer are discussed.

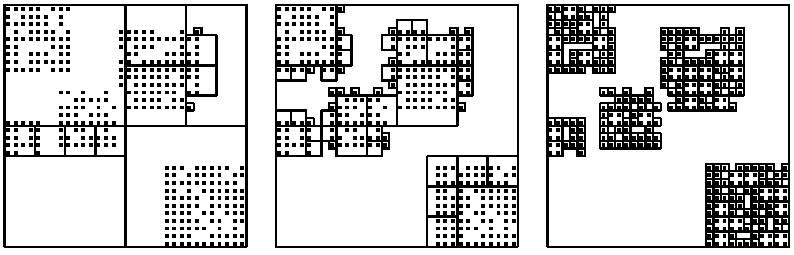


Figure 6.13: Dense Sub-Matrices ($\rho^t = 0.4$, $\rho^t = 0.5$ and $\rho^t = 1.0$)

6.2.1 Feedback to the Programmer

To provide some feedback to the programmer, the results of the analysis can be prompted to the user either as the pictures shown in this chapter, or in a readable format. Below, we present this latter output for the 80×80 matrix 'steam3' of the Harwell-Boeing Sparse Matrix Collection [79] shown in figure 6.14. First, some general information is prompted which is computed under assumption that each nonzero element is stored exactly once in coordinate scheme. Moreover, the number of detected dense blocks, shown in the first picture of figure 6.14, is given.

```
+--- THRESHOLD = 0.50 -------------------------------------+
| Size            :     80 x 80                            |
| #Entries        :        928                             |
| Av #Entries/row :      11.60   Density : 0.1450          |
| Av #Entries/col :      11.60   #Blocks : 86              |
|                                                          |
| Semi-Bandwidths :     43-43    #Elts   : 5068            |
| #Used Diagonals :     29       #Elts   : 1472            |
| #Full Diagonals :      3       2-2 SB  : 30.17  %        |
|                                >= 90% B: 83              |
|                                                          |
| Block-D/L/U     :      4960      5680      5680          |
|                      40/10     40/10     40/10           |
|                                                          |
| Type            : Multi-Diagonal Matrix          (0.6304)|
+----------------------------------------------------------+
```

Subsequently, some band related information is given. In the band with lower and upper semi-bandwidth of 43, 5068 elements appear. There are 29 used diagonals containing 1472 elements in total, whereas only 3 of these diagonals are full. Furthermore, 30.17% of the entries appears in a band with bandwidth 5, and the smallest band containing more than 90% of all entries has bandwidth 83.

Thereafter, the total number of elements in the nonzero blocks of minimum block partitions into (bordered) block diagonal and triangular form are shown in combination with a pair containing the corresponding border size and number of diagonal blocks in the remaining sub-matrix. Finally, the classification of this matrix is shown, together with the density within the corresponding nonzero regions. For the example, the multi-diagonal form, illustrated in figure 6.14, is selected having a density of 0.6304 (viz. $928/1472$).

6.2.2 Performance

In table 6.1, we present the execution time of a straightforward implementation of the analyzer on an HP 9000/720 (compiled with default optimizations enabled) for some matrices of the Harwell-Boeing Sparse Matrix Collection [79] that have been converted into coordinate scheme.

The column denoted with 'R' contains the execution time required to read the matrix from file. The column denoted with 'R-A' contains the execution time required to read and analyze the file if

Matrix	n	τ	R	R-A	R-FA	R (HB)
jpwh_991	991	6027	0.7	0.8	0.9	0.5
gre_1107	1107	5664	0.8	0.9	1.0	0.5
orani678	2529	90158	14.0	14.2	15.3	7.8
lns_3939	3937	25407	3.6	3.8	4.3	2.3
psmigr_1	3140	543162	71.9	72.8	82.4	31.7

Table 6.1: Analysis Time in seconds on an HP 9000/720

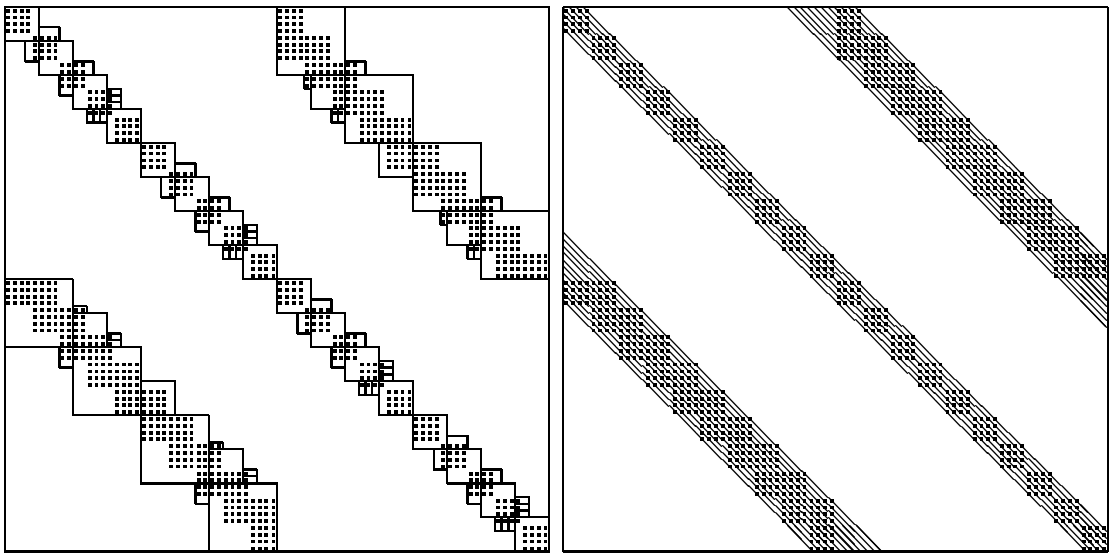


Figure 6.14: Analysis of 'steam3' ($\rho^t = 0.5$)

detecting dense sub-matrices is disabled. The column denoted with 'R-FA' contains the execution time for reading the matrix and performing a full analysis, where the threshold $\rho^t = 0.8$ is used to avoid fast termination of the algorithm that detects dense sub-matrices. Finally, in the last column we show the time required to read the matrix from file using the column-wise Harwell-Boeing standard sparse matrix format.

This table indicates that, although the complete analysis time can be substantial due to the fact that the matrix must be read from file, the execution time required for actually analyzing the matrix is small with respect to the time needed to read the matrix from file. Moreover, we see that reading the coordinate scheme is more expensive than reading the column-wise Harwell-Boeing standard sparse matrix format.

6.3 Propagation of Nonzero Structure Information

In the sparse compiler, nonzero structure information is obtained by means of annotations or automatic analysis of matrices on file. In this section, we discuss how this information is propagated to subsequent phases of the automatic data structure selection and transformation method.

6.3.1 Property Summary Set

Nonzero structure information is propagated to subsequent phases by means of a property summary set. Each individual **property summary** $\overline{p}$ is a triple consisting of a simple section $P \subseteq \mathcal{Z}^2$, a preferred access direction $\vec{p} \in \mathcal{Z}^2$ for the region of which the index set is represented by the simple section, and the property $p \in \{\mathbf{zero}, \mathbf{dense}, \mathbf{sparse}\}$ of this region:

$$\overline{p} = \langle P, \vec{p}, p \rangle$$

Unless a preferred direction is defined explicitly with an annotation, we set the preferred access direction to $\vec{p} = \vec{\delta}(P)$ using the following definition of $\vec{\delta}(P) \in \mathcal{Z}^2$, where k denotes the index that minimizes the expression $\tau_i - \sigma_i$ for the boundary values $\vec{\sigma} \in \mathcal{Z}^4$ and $\vec{\tau} \in \mathcal{Z}^4$ of $P \subseteq \mathcal{Z}^2$:

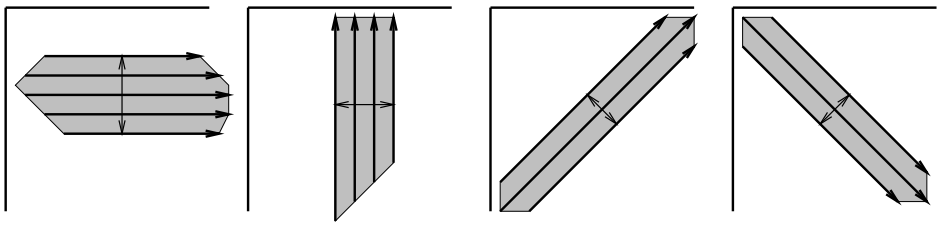


Figure 6.15: Preferred Access Direction

$$\vec{\delta}(P) = \begin{cases} (\ 0, +1)^T & \textbf{if} \quad k = 1 \\ (-1, \ 0)^T & \textbf{if} \quad k = 2 \\ (-1, +1)^T & \textbf{if} \quad k = 3 \\ (+1, +1)^T & \textbf{if} \quad k = 4 \end{cases} \tag{6.3}$$

In this manner, the number of straight lines through the elements of $P \subseteq \mathcal{Z}^2$ that are along the preferred direction is minimized, as is illustrated in figure 6.15.

A set of property summaries of an implicitly sparse matrix A is referred to as the **property summary set** $\mathcal{P}_A$ of this matrix, which must have the property that the simple sections associated with the property summaries are non-empty and mutually disjoint.

6.3.2 Nonzero Structure Annotations

For each nonzero structure annotation describing a property of a region in an $m \times n$ implicitly sparse matrix A, the following steps are taken. First, the property p is set to the property defined in the annotation. Subsequently, we initialize the boundary values $\vec{\sigma} \in \mathcal{Z}^4$ and $\vec{\tau} \in \mathcal{Z}^4$ of a simple section P as follows:

$$P = \{(i,j) \in \mathcal{Z}^2 \mid (1,1,2,1-n)^T \leq \mathcal{M}(i,j)^T \leq (m,n,m+n,m-1)^T\} \tag{6.4}$$

Thereafter, the boundary pairs that occur in the annotation are scanned. For each construct defining boundary values $\sigma_i = l$ and $\tau_i = u$, the following assignment statements are executed:

$$\begin{cases} \sigma_i & := & \max(\sigma_i, l); \\ \tau_i & := & \max(\tau_i, u); \end{cases}$$

Thereafter, the boundaries are refined with the procedure `refine` presented in chapter 5. If a preferred access direction is defined in the annotation, then this direction is normalized according to (5.2) and assigned to $\vec{p} \in \mathcal{Z}^2$, or we set $\vec{p} = \delta(P)$ otherwise. Finally, if the simple section $P \subseteq \mathcal{Z}^2$ is non-empty and does not overlap with a simple section corresponding to a property summary already present in $\mathcal{P}_A$, then we insert the resulting triple $\langle P, \vec{p}, p \rangle$ into the property summary set $\mathcal{P}_A$.

Example: Consider the following nonzero structure annotations:

```
      REAL A(50,50)
C_SPARSE(A : _DENSE ( 1 <= I <=  5, 1 <=     J <= 5)(2,4))
C_SPARSE(A : _SPARSE(40 <= I <= 50, 0 <= I - J <= 0)     )
```

These annotations gives rise to the property summary set:

$$\mathcal{P}_A = \{\langle P_1, (1,2)^T, \textbf{dense} \rangle, \langle P_2, (1,1)^T, \textbf{sparse} \rangle\}$$

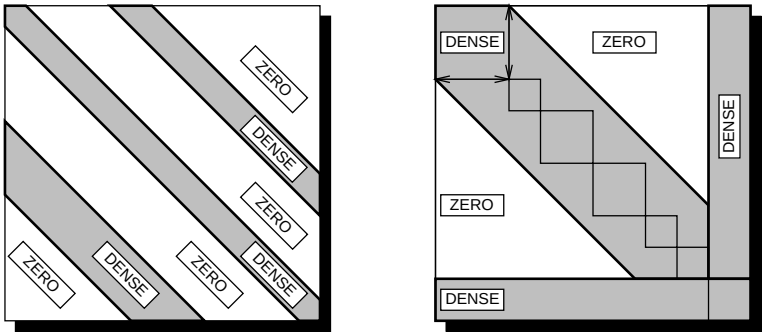


Figure 6.16: Regions in Multi-Diagonal and Bordered Block Diagonal Form

In this set, the simple sections $P_1 \subseteq \mathcal{Z}^2$ and $P_2 \subseteq \mathcal{Z}^2$ have the following form:

$$\left\{ \begin{array}{lcll} P_1 & = & \{(i,j) \in \mathcal{Z}^2 \mid (1,1,2,-4)^T & \leq \mathcal{M}(i,j)^T \leq \quad (5,5,10,4)^T\} \\ P_2 & = & \{(i,j) \in \mathcal{Z}^2 \mid (40,40,80,0)^T & \leq \mathcal{M}(i,j)^T \leq (50,50,100,0)^T\} \end{array} \right.$$

A nonzero structure annotation with an empty list of boundary pairs (viz. '_ZERO()') defines a property for the whole matrix.

6.3.3 Automatic Nonzero Structure Analysis

Because block data structures are not (yet) supported by the prototype sparse compiler and nonzero structure information is done by means of property summaries, not all the information determined by automatic nonzero structure analysis is actually supplied to the sparse compiler.

If a matrix is classified either as a band matrix (with diagonal, tridiagonal and (band) triangular form as special classes) or as a multi-diagonal matrix, then property summaries with simple sections that describe the index sets of the zero and dense regions of this form are constructed. As illustrated in the first picture of figure 6.16, consecutive empty or used diagonals are collapsed into one region. Likewise, if a matrix is classified as a bordered block matrix, access summaries of which the simple sections represent the index sets of the dense borders, the zero regions and possibly the dense band of which the bandwidth is defined by the largest diagonal block in the remaining sub-matrix are constructed, as illustrated for a bordered block diagonal form in the second picture of figure 6.16.

Subsequently, the compiler inquires the programmer whether the zero regions will remain zero at run-time. If not, the properties of these regions are converted into sparse. In addition, for an $m \times n$ implicitly sparse matrix, all simple sections are intersected with the simple section (6.4) to account for the fact that the current analyzer handles the matrix as an $N \times N$ matrix, where $N = \max(m,n)$. Finally, each access summary of which the simple section $P \subseteq \mathcal{Z}^2$ remains non-empty is inserted into the property summary set $\mathcal{P}_A$, where we define $\vec{p} = \vec{\delta}(P)$.

Example: The analyzer classifies matrix 'steam3' as a multi-diagonal matrix, as is illustrated in the second picture of figure 6.14. If the user indicates that all zero regions are preserved at run-time, this classification gives rise to the following property summary set:

$$\mathcal{P}_A = \{\langle P_1, (1,1)^T, \mathbf{zero}\rangle, \langle P_2, (1,1)^T, \mathbf{dense}\rangle, \ldots, \langle P_7, (1,1)^T, \mathbf{zero}\rangle\}$$

The simple sections in this set have the following form:

$$\left\{ \begin{array}{lcll} P_1 & = & \{(i,j) \in \mathcal{Z}^2 \mid (1,45,46,-79)^T & \leq \mathcal{M}(i,j)^T \leq (36,80,116,-44)^T\} \\ P_2 & = & \{(i,j) \in \mathcal{Z}^2 \mid (1,34,35,-43)^T & \leq \mathcal{M}(i,j)^T \leq (47,80,127,-33)^T\} \\ \quad \vdots & & & \quad \vdots \\ P_7 & = & \{(i,j) \in \mathcal{Z}^2 \mid (45,1,46,44)^T & \leq \mathcal{M}(i,j)^T \leq \quad (80,36,116,79)^T\} \end{array} \right.$$

Chapter 7

Phase 2: Data Structure Selection

After all enveloping data structures of implicitly sparse matrices have been identified, and the access summaries of all occurrences of these arrays have been computed, a sparse storage scheme must be selected for each implicitly sparse matrix. Although there are dynamic sparse storage schemes that support the fast generation of entries along different directions, for instance, along rows and columns (cf. the linked list schemes presented in section 4.1.3), the overhead storage and run-time maintenance overhead of such storage schemes is usually substantial. Therefore, we have limited the sparse storage scheme that may be selected to store the entries in the sparse regions of an implicitly sparse matrix to a dynamic data structure that consists of a pool of sparse vectors supporting only one direction for each individual sparse region. However, the layout of sparse vectors may be different for different sparse regions.

This implies that it becomes very important that each sparse region is accessed in a consistent manner, namely along the direction supported for that region. Only in this manner, the the sparse overhead reducing techniques guard encapsulation and access pattern expansion may become enabled. Even dense regions should be accessed in a consistent manner, because this enables the selection of a full-sized array over this region in which elements along the most frequently occurring access patterns are stored along the columns, which enhances spatial locality in FORTRAN. Because it is likely that the enveloping data structures are accessed along arbitrary directions in the original dense program, a method to reshape the access patterns of two-dimensional arrays, based on the unimodular framework presented in chapter 3, has been incorporated in the sparse compiler.

Another important step for the sparse compiler is the construction of a number of mutually disjoint regions in each implicitly sparse matrix A such that each part of the corresponding enveloping data structure A that may be accessed by an arbitrary occurrence of A corresponds to a region in A confined to one of these regions. Given these regions, the sparse compiler can select a different storage organization for each region and convert the code accordingly. Since each occurrence can only access a part of the enveloping data structure corresponding to a region in A that is confined to *only one* of these regions, the need for run-time tests to determine which of the selected storage organizations must be accessed is avoided. Because usually only a limited number of such regions can be distinguished, we also discuss how iteration space partitioning, presented in detail in chapter 3, can be used to increase the resulting amount of fragmentation.

In this chapter, we present a reshaping method as well as a method to construct a number of non-overlapping regions in an implicitly sparse matrix of which the index sets are described in terms of simple sections. Where possible, unnecessary re-computation of information obtained in the first phase is avoided. The chapter is concluded with a discussion of the actual data structure selection and generation of the corresponding declarations. Note that, in general, finding the best data structure is computationally infeasible [148].

7.1 Reshaping Access Patterns

In this section, we first motivate the importance of reshaping access patterns. Thereafter, the actual reshaping method [25] is discussed in detail.

7.1.1 Motivation

Consider, for example, the following fragment, where the operation $\vec{c} \leftarrow A\vec{b}$ is followed by the accumulation of some elements in an implicitly sparse matrix A for which a two-dimensional array A is used as enveloping data structure:

```
      INTEGER    I, J, M, N
      PARAMETER (M = ..., N = ...)
      REAL    A(M,N), C(M), B(N), X
C_SPARSE (A)
      ...
      DO I = 1, M
S1:     C(I) = 0.0
        DO J = 1, N
S2:       C(I) = C(I) + A(I,J) * B(J)
        ENDDO
      ENDDO
      DO J = 1, N / 2
        DO I = 1, M / 2
S3:       X = X + A(2*I,2*J)
        ENDDO
      ENDDO
```

Since the occurrences of A in statements S_2 and S_3 have respectively row- and column-wise true access patterns, only one of the following sparse versions can result after selecting either general sparse row-wise or general sparse column-wise storage, where function LKP__ performs a lookup for a particular entry in one of the sparse vectors of pool and returns '$\perp$' for non-entries (cf. function σ_A of section 4.3.2):

row-wise storage:

```
DO I = 1, M
  C(I) = 0.0
  DO J_ = LOW_A(I), HGH_A(I)
    J    = IND_A(J_)
    C(I) = C(I) + VAL_A(J_) * B(J)
  ENDDO
ENDDO


DO J = 1, N / 2
  DO I = 1, M / 2
    L = LKP__(IND_A, LOW_A(2*I),
+             HGH_A(2*I), 2*J)
    IF (L ≠ ⊥) THEN
      ACC = ACC + VAL_A(L)
    ENDIF
  ENDDO
ENDDO
```

column-wise storage:

```
DO I = 1, M
  C(I) = 0.0
  DO J = 1, N
    L = LKP__(IND_A, LOW_A(J),
+             HGH_A(J), I)
    IF (L ≠ ⊥) THEN
      C(I) = C(I) + VAL_A(L) * B(J)
    ENDIF
  ENDDO
ENDDO
DO J = 1, N / 2
  DO I_ = LOW_A(2*J), HGH_A(2*J)
    I = IND_A(I_)
    IF (MOD(I,2) = 0) THEN
      X = X + VAL_A(I_)
    ENDIF
  ENDDO
ENDDO
```

In the first version, guard encapsulation is feasible for S_2, which implies that a construct that, at run-time, iterates over the entries in each Ith row can be used. A similar construct that iterates over entries in each 2*Jth column can be used for statement S_3 in the second version. Some test overhead remains, to account for the fact that for each column only the entries with even row index are actually operated on. Unfortunately, lookups are required for all occurrences of which the true access patterns conflict with the selected storage scheme.

Each lookup induces substantial overhead because in the worst case all entries in a whole row or column must be scanned to obtain the address of an entry or to conclude that the element is zero. Moreover, no reduction in the number of iterations is obtained. For example, statement S_2 is still executed $\mathtt{M} \cdot \mathtt{N}$ times in the second version, but only τ times in the first version, where τ indicates the total number of entries in matrix A.

Obviously, these problems are caused by the fact that the program induces inconsistent accesses to the implicitly sparse matrix A. Since the sparse storage scheme that can be selected by the compiler only supports storage of entries along one particular access direction to prevent prohibitive storage and maintenance overhead, it is desirable to resolve such conflicts by reshaping access patterns. In the example, interchanging the DO-loops surrounding S_3 enables the generation of the following version if sparse row-wise storage is selected:

reshaped row-wise storage:

```
DO I = 1, M
  C(I) = 0.0
  DO J_ = LOW_A(I), HGH_A(I)
    J = IND_A(J_)
    C(I) = C(I) + VAL_A(J_) * B(J)
  ENDDO
ENDDO
DO I = 1, M / 2
  DO J_ = LOW_A(2*I), HGH_A(2*I)
    J = IND_A(J_)
    IF (MOD(J,2) = 0) THEN
      ACC = ACC + VAL_A(J_)
    ENDIF
  ENDDO
ENDDO
```

In table 7.1, we present the execution time of the sparse versions obtained without reshaping and the reshaped version on one CPU of a Cray C98/4256 for some matrices of the Harwell-Boeing Sparse Matrix Collection [79] (converted into the appropriate storage format), compiled with default optimizations and vectorization enabled. The row-wise version is preferable over the column-wise version because lookups are executed less frequently in the former (viz. $\frac{1}{4} \cdot \mathtt{M} \cdot \mathtt{N}$ vs. $\mathtt{M} \cdot \mathtt{N}$ times). However, the reshaped version is clearly superior, due to the elimination of all lookups. Running the first two versions without the test overhead of '$\mathtt{L} \neq \perp$' by using the property that $\mathtt{VAL\_A}(\perp) = 0.0$ results in almost identical timings.

This experiment illustrates the most important objective for sparse codes, namely that the number of operations performed must be kept proportional to the number of entries in the sparse matrix [68, 78, 169]. Skipping operations on zeros by means of conditionals is useless since conditions have to be evaluated anyway. In addition, scanning a sparse data structure to obtain an entry must be avoided as much as possible. For the automatic data structure selection and sparse code generation method this implies that it is very important that eventually each region in an implicitly sparse matrix is accessed in a consistent manner. Access pattern reshaping can be used to achieve this goal.

Matrix	N	NNZ	Row	Column	Reshaped Row
steam2	600	13760	0.1	0.5	$1.4 \cdot 10^{-3}$
jagmesh1	936	3600	0.3	1.1	$1.7 \cdot 10^{-3}$
gre_1107	1107	5664	0.4	1.5	$2.1 \cdot 10^{-3}$
orani678	2529	90158	2.2	8.8	$6.4 \cdot 10^{-3}$

Table 7.1: Execution Time in seconds on a Cray C98/4256

7.1.2 Objective of Reshaping

Suppose that the admissible subscripts of an occurrence of a two-dimensional array appearing in a perfectly nested loop with index vector $\vec{\mathtt{I}} = (\mathtt{I}_1, \ldots, \mathtt{I}_d)^T$ are represented by the affine transformation $F(\vec{\mathtt{I}}) = \vec{v} + W\vec{\mathtt{I}}$. Recall that the last column of the integer matrix W is called the true access direction $\vec{r}$ of this occurrence.

A loop transformation defined by a $d \times d$ unimodular matrix U, transforming the original loop with index vector $\vec{\mathtt{I}}$ into a target loop with index vector $\vec{\mathtt{I}}' = U\vec{\mathtt{I}}$, changes the true access direction by replacing the original subscripts $F(\vec{\mathtt{I}}) = \vec{v} + W\vec{\mathtt{I}}$ with $F'(\vec{\mathtt{I}}') = \vec{v} + WU^{-1}\vec{\mathtt{I}}'$. We say that *a loop transformation reshapes the access patterns of this occurrence along a preferred access direction $\vec{s} \in \mathcal{Z}^2$, if this latter vector and the resulting true access direction $\vec{r}\,' \in \mathcal{Z}^2$ are linearly dependent,* which implies that the following equation holds for some $\lambda \in \mathcal{Z}$.

$$\vec{r}\,' = WU^{-1}(\underbrace{0, \ldots, 0}_{d-1}, 1)^T = \lambda \cdot \vec{s}$$

If $\gcd(s_1, s_2) = 1$, this objective precisely gives the solutions of the following linear diophantine equation [17, 19]:

$$(+s_2, -s_1) \cdot \vec{r}\,' = (+s_2, -s_1) \cdot WU^{-1}(\underbrace{0, \ldots, 0}_{d-1}, 1)^T = 0 \tag{7.1}$$

Note that we allow reshaping that results in scalar-wise true access patterns, because both equations are trivially satisfied for the new true access direction $\vec{r}\,' = \vec{0}$. In this case, the resulting effective access direction may not be equal to the preferred access direction.

Example: Consider the problem of reshaping the access patterns of the following occurrence of a two-dimensional array A along the preferred access direction $\vec{s} = (0, 1)^T$:

```
DO I1 = 1, 10
  DO I2 = 1, 10
    DO I3 = 1, 10
      ... =  A(I1+2*I3,I2)
    ENDDO
  ENDDO
ENDDO
```

One way to obtain row-wise true access patterns is to keep $\mathtt{I}_1 + 2 \cdot \mathtt{I}_3$ constant in one iteration of the outermost DO-loop. As discussed in section 3.3.4, this can be achieved by applying a transformation that is defined by a unimodular matrix with $(1, 0, 2)$ as first row:

```
DO I1' = 3, 30
  DO I2' = 1, 10
    DO I3' = MAX(1,⌈(I1'-10/2)⌉), MIN(10,⌊(I1'-1)/2⌋)
       ... = A(I1',I2')
    ENDDO
  ENDDO
ENDDO
```

$$U = \left(\begin{array}{ccc} 1 & 0 & +2 \\ \hline 0 & 1 & 0 \\ 0 & 0 & 1 \end{array}\right) \qquad U^{-1} = \left(\begin{array}{ccc} 1 & 0 & -2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array}\right)$$

Although scalar-wise true access patterns result in the target loop, the resulting effective access patterns are row-wise. However, the method provides little flexibility. Especially if we want to reshape the access pattern of *several* occurrences in a loop, it is unlikely that the same pre-described first row of U results. More flexibility is obtained by observing that traversing the original iteration space along any straight line coinciding with a plane defined by $\mathtt{I}_1 + 2 \cdot \mathtt{I}_3 = i_1'$ also induces accesses to (possibly identical) elements in one row, as illustrated in figure 7.1.

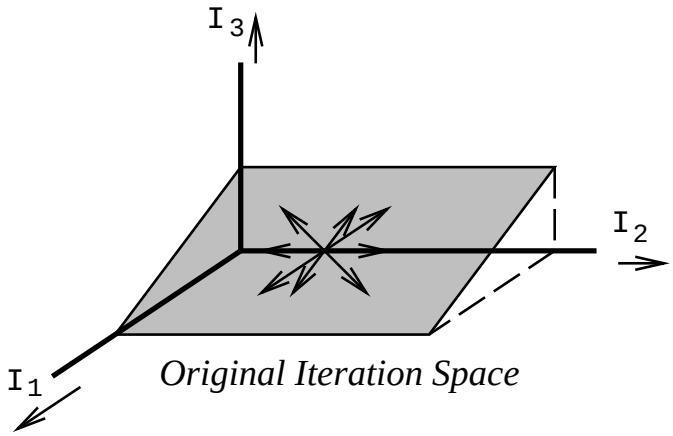


Figure 7.1: Plane Defined by $\mathtt{I}_1 + 2 \cdot \mathtt{I}_3 = i'_1$

The general form of the direction of such lines through the original iteration space is given by the vector $(-2 \cdot \lambda_3, \lambda_2, \lambda_3)^T$, for arbitrary $\lambda_2, \lambda_3 \in \mathcal{Z}$. As was also discussed in section 3.3.4, this goal can be achieved by applying a transformation defined by any unimodular matrix U for which the last column of U^{-1} is equal to an instance of this direction. Not surprisingly, these are exactly the matrices that satisfy objective (7.1). In the transformation above, the direction $(-2, 0, 1)^T$ is chosen (viz. $\lambda_3 = 1$ and $\lambda_2 = 0$). Alternatively, the direction $(0, 1, 0)^T$ gives rise to a transformation defined by the following matrices:

$$U = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \qquad U^{-1} = \left(\begin{array}{cc|c} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{array}\right)$$

Indeed, interchanging the $\mathtt{I}_2$- and $\mathtt{I}_3$-loop of the original fragment yields a fragment with row-wise true access patterns. The increased flexibility, however, may come at a small penalty. For example, given the direction $(-2, 0, 1)^T$, we could equally well apply a transformation defined by the following unimodular matrices:

$$U = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 2 \\ 0 & 0 & 1 \end{pmatrix} \qquad U^{-1} = \left(\begin{array}{cc|c} 0 & 1 & -2 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{array}\right)$$

In this case, a fragment with scalar-wise true access patterns would result in which *column*-wise effective access patterns are induced at higher level. However, because this approach is more flexible, and in the sparse code lookup overhead of scalar-wise access patterns could be amortized over several iterations, we accept the fact if scalar-wise true access patterns result, then the effective access direction and the preferred access direction may differ.

7.1.3 Method of Reshaping

In this section, we present a general method to construct a valid unimodular transformation that simultaneously reshapes the access patterns of c different occurrences of two-dimensional arrays along preferred access directions $\vec{s}^{\,1} \in \mathcal{Z}^2, \ldots, \vec{s}^{\,c} \in \mathcal{Z}^2$, where for all $1 \leq i \leq c$ we have $\gcd(s_1^i, s_2^i) = 1$. We assume that all occurrences appear in a single perfectly nested loop with index vector $\vec{\mathtt{I}} = (\mathtt{I}_1, \ldots, \mathtt{I}_d)^T$ and subscripts $F_1, \ldots, F_c$. For all $1 \leq i \leq c$, the subscripts can be expressed as $F_i(\vec{\mathtt{I}}) = \vec{v}_i + W_i \vec{\mathtt{I}}$. The data dependence structure of this loop is represented by a set $\mathcal{D} \subseteq \mathcal{Z}^d$ of dependence distance vectors.

Construction of Preferred Iteration Directions

A loop transformation defined by a $d \times d$ unimodular matrix U reshapes the access patterns of the ith occurrence in the loop along the preferred access direction $\vec{s}^{\,i} = (s_1^i, s_2^i)^T$ if the last column $\vec{\alpha} \in \mathcal{Z}^d$ of the inverse matrix U^{-1} satisfies objective (7.1):

$$(+s_2^i, -s_1^i) \cdot W_i \vec{\alpha} = 0$$

After application of such a transformation, conceptually the original iteration space is traversed along straight lines with the direction $\vec{\alpha} \in \mathcal{Z}^d$. Therefore, this vector is referred to as the **preferred iteration direction**. All access patterns are reshaped simultaneously if $\vec{\alpha} \in Z^d$ satisfies $S\vec{\alpha} = \vec{0}$ for the following $c \times d$ integer matrix S, called the **objective matrix**:

$$S = \begin{pmatrix} +s_2^1 & -s_1^1 & & & & & \\ & & +s_2^2 & -s_1^2 & & & \\ & & & & \ddots & & \\ & & & & & +s_2^c & -s_1^c \end{pmatrix} \begin{pmatrix} W_1 \\ \vdots \\ W_c \end{pmatrix}$$

Since the elements in each row or column of a unimodular matrix must be relatively prime, any integer solution of $S\vec{\alpha} = \vec{0}$ with $\gcd(\alpha_1, \ldots, \alpha_d) = 1$ may be used as preferred iteration direction, which gives rise to the following set $\Delta \subseteq \mathcal{Z}^d$:

$$\Delta = \{\vec{\alpha} \in \mathcal{Z}^d \mid S\vec{\alpha} = 0 \text{ and } \gcd(\alpha_1, \ldots, \alpha_d) = 1\}$$

The real solutions of the homogeneous system $S\vec{\alpha} = \vec{0}$ form a $(d - \text{rank}(S))$-dimensional linear subspace of $\mathcal{R}^d$, i.e. the kernel of the linear transformation defined by S. More specifically, proposition 2.2 implies that if we use an integer echelon reduction algorithm [19, p32-39] to compute the unimodular matrix R such that RS^T is in echelon form (yielding $r = \text{rank}(S)$ as side-effect), then all integer solutions of the homogeneous integer system $S\vec{\alpha} = \vec{0}$ are given by the following formula for arbitrary $\lambda_i \in \mathcal{Z}$:

$$\vec{\alpha} = [\ (\underbrace{0, \ldots, 0}_{r}, \lambda_{r+1}, \ldots, \lambda_d)R\ ]^T \tag{7.2}$$

This observation provides a simple condition for the existence of a (possibly invalid) unimodular transformation that performs the preferred reshaping:

Proposition 7.1 *There exists a $d \times d$ unimodular matrix U for which the last column $\vec{\alpha} \in \mathcal{Z}^d$ of U^{-1} satisfies $S\vec{\alpha} = \vec{0}$ for some $c \times d$ matrix S if and only if rank$(S) < d$.*

If $r = d$, then the reshaping method fails since we cannot construct a unimodular matrix with a zero vector as last column. Otherwise, $r < d$ holds and the last $d - r$ rows of R form a basis of the linear subspace consisting of all solutions of the homogeneous system. Because there may be infinitely many solutions of which the components are relatively prime, we restrict our attention to these basis vectors. We obtain the following set $\Delta_b \subseteq \Delta$, where the components of each vector in this set are relatively prime because R is unimodular:

$$\Delta_b = \{\vec{\alpha} \in \mathcal{Z}^d \mid \vec{\alpha} = [(\underbrace{0, \ldots, 0}_{k-1}, 1, \underbrace{0, \ldots, 0}_{d-k})R\ ]^T, r < k \leq d\}$$

Construction of Valid Transformation

For any $\vec{\alpha} \in \Delta_b$, we can use an extended completion method presented in section 2.2.2 to construct a $d \times d$ unimodular matrix U for which the last column of U^{-1} consists of this preferred iteration direction. Given these unimodular matrices, we can exploit the fact that for any $(d-1) \times (d-1)$ unimodular matrix Y and integer $z \in \{-1, +1\}$, the following V is still a unimodular matrix for which the last column of the inverse is $\pm\vec{\alpha}$:

$$V = \begin{pmatrix} & & & 0 \\ & Y & & \vdots \\ & & & 0 \\ 0 & \dots & 0 & z \end{pmatrix} U \qquad V^{-1} = U^{-1} \begin{pmatrix} & & & 0 \\ & Y^{-1} & & \vdots \\ & & & 0 \\ 0 & \dots & 0 & z \end{pmatrix} \tag{7.3}$$

Consequently, if for a given $\vec{\alpha} \in \Delta_b$ and corresponding U we can construct a unimodular matrix Y and integer $z \in \{-1, +1\}$ that define a matrix V with $V\vec{d} \succ \vec{0}$ for all $\vec{d} \in \mathcal{D}$, then a valid transformation performing the preferred reshaping has been found. Otherwise, another $\vec{\alpha} \in \Delta_b$ is tried until either this construction is successful, or the set Δ_b has been exhausted. In the latter case, the reshaping method fails.

For $\mathcal{D} = \emptyset$, the construction is trivial, since we can use the loop transformation defined by $V = U$ (viz. $Y = I$ and $z = 1$). Otherwise, we define the following set $\tilde{\mathcal{D}} \subseteq \mathcal{D}$:

$$\tilde{\mathcal{D}} = \{\vec{d} \in \mathcal{D} \mid U\vec{d} = (\underbrace{0, \dots, 0}_{d-1}, \lambda)^T, \lambda \in \mathcal{Z}\}$$

The set of dependence distance vectors $\mathcal{D}$ can be partitioned into $\tilde{\mathcal{D}}$ and $\mathcal{D} - \tilde{\mathcal{D}}$. We will see that dependence distance vectors in the former set determine the selection of the integer $z \in \{-1, +1\}$, whereas dependence distance vectors in the latter set must be dealt with during the construction of the matrix Y:

Proposition 7.2 *Given a $d \times d$ unimodular matrix U and a set $\tilde{\mathcal{D}} \subseteq \mathcal{Z}^d$ where each $\vec{d} \in \tilde{\mathcal{D}}$ satisfies $\vec{d} \succeq \vec{0}$ and $U\vec{d} = (0, \dots, 0, \lambda)^T$ for some $\lambda \in \mathcal{Z}$, then either:*

1. *for all $\vec{d} \in \tilde{\mathcal{D}}$ we have $U\vec{d} \succeq \vec{0}$, or*
2. *for all $\vec{d} \in \tilde{\mathcal{D}}$ we have $U\vec{d} \preceq \vec{0}$.*

PROOF Assume that there are $\vec{d_1}, \vec{d_2} \in \tilde{\mathcal{D}}$ of the following form:

- $U\vec{d_1} = (0, \dots, 0, \lambda_1)^T$ for $\lambda_1 > 0$, and
- $U\vec{d_2} = (0, \dots, 0, \lambda_2)^T$ for $\lambda_2 < 0$.

Obviously, $\vec{d} = U^{-1}(0, \dots, 0, \lambda)^T$ implies that both $\vec{d_1} = \lambda_1 \cdot \vec{v}$ and $\vec{d_2} = \lambda_2 \cdot \vec{v}$ hold for a fixed $\vec{v} \neq \vec{0}$, namely the last column of U^{-1}. This is in contradiction with the assumption that both vectors are lexicographically positive. Consequently, either:

1. for all $\vec{d} \in \tilde{\mathcal{D}}$, $U\vec{d} = (0, \dots, 0, \lambda)^T$ for some $\lambda \geq 0$, i.e. $U\vec{d} \succeq \vec{0}$, or
2. for all $\vec{d} \in \tilde{\mathcal{D}}$, $U\vec{d} = (0, \dots, 0, \lambda)^T$ for some $\lambda \leq 0$, i.e. $U\vec{d} \preceq \vec{0}$.

□

This proposition provides a convenient method to select a suitable integer $z \in \{-1, +1\}$:

Corollary 7.1 *Given a $d \times d$ unimodular matrix U and a set $\tilde{\mathcal{D}} \subseteq \mathcal{Z}^d$ where each $\vec{d} \in \tilde{\mathcal{D}}$ satisfies $\vec{d} \succeq \vec{0}$ and $U\vec{d} = (0, \ldots, 0, \lambda)^T$ for some $\lambda \in \mathcal{Z}$, then for any $(d-1) \times (d-1)$ matrix Y there exists an integer $z \in \{-1, +1\}$ such that (7.3) defines a matrix V with $d \succeq \vec{0}$ for all $\vec{d} \in \tilde{\mathcal{D}}$,*

PROOF Since the first $d-1$ components of $V\vec{d}$ remain zero for any $\vec{d} \in \tilde{\mathcal{D}}$ and Y, this corollary follows directly from proposition 7.2. We select the integer $z = -1$ if $U\vec{d} \prec \vec{0}$ for any $\vec{d} \in \tilde{\mathcal{D}}$, or the integer $z = 1$ otherwise. □

In $z = -1$ is selected, the preferred iteration direction $\vec{\alpha}$ is converted into $-\vec{\alpha}$, which is required if $\vec{\alpha}$ is in the opposite direction of some dependence distance vectors. Thereafter, only the remaining dependence distance vectors in $\mathcal{D} - \tilde{\mathcal{D}}$ have to be accounted for during the construction of the unimodular matrix Y. Let $\tilde{U}$ consist of the first $d-1$ rows of the unimodular matrix U:

$$\tilde{U} = \begin{pmatrix} 1 & & 0 \\ & \ddots & \vdots \\ & 1 & 0 \end{pmatrix} U$$

Obviously, this definition enables us to define the partition as $\tilde{\mathcal{D}} = \{\vec{d} \in \mathcal{D} \mid \tilde{U}\vec{d} = \vec{0}\}$ and $\mathcal{D} - \tilde{\mathcal{D}} = \{\vec{d} \in \mathcal{D} \mid \tilde{U}\vec{d} \neq \vec{0}\}$. Because for any Y and $\vec{d} \in \mathcal{D} - \tilde{\mathcal{D}}$ the inequality $Y\tilde{U}\vec{d} \neq \vec{0}$ holds, we have to find a matrix Y that satisfies the following constraint for all $\vec{d} \in \mathcal{D} - \tilde{\mathcal{D}}$:

$$Y\tilde{U}\vec{d} \succ \vec{0} \tag{7.4}$$

Such a matrix Y does not always exist. For instance, for the matrix $\tilde{U} = (1, -1)$ and the set of remaining dependence distance vectors $\{(1,0)^T, (0,1)^T\}$, there is no $Y = (y)$, where $y \in \{-1, +1\}$, such that this objective is satisfied.

We use the following property:

Proposition 7.3 *Given a set $\mathcal{D}_U \subseteq \mathcal{Z}^{d-1}$ with $\vec{d} \succ \vec{0}$ for all $\vec{d} \in \mathcal{D}_U$, then there exists a $(d-1) \times (d-1)$ unimodular matrix F such that $F\vec{d} \succ_1 \vec{0}$ for all $\vec{d} \in \mathcal{D}_U$.*

Consequently, if there exists a unimodular matrix Y that satisfies (7.4) for all $\vec{d} \in \mathcal{D} - \tilde{\mathcal{D}}$, then this proposition implies that another unimodular matrix $\tilde{Y}$ exists such that the constraint $\tilde{Y}\tilde{U}\vec{d} \succ_1 \vec{0}$ holds for all $\vec{d} \in \mathcal{D} - \tilde{\mathcal{D}}$ (viz. let $\mathcal{D}_U$ consist of all vectors $\tilde{U}\vec{d}$ and set $\tilde{Y} = FY$). Hence, we can safely focus on finding this latter matrix directly.

First, we determine whether there is a vector $\vec{y} \in \mathcal{Z}^{d-1}$ with $\gcd(y_1, \ldots, y_{d-1}) = 1$ such that the inequality $\vec{y} \cdot \tilde{U}\vec{d} > 0$ holds for all $\vec{d} \in \mathcal{D} - \tilde{\mathcal{D}}$. This problem is equivalent to finding a suitable solution of the following system of inequalities, where the rows of the integer matrix M are formed of the vectors $\tilde{U}\vec{d}$ for all $\vec{d} \in \mathcal{D} - \tilde{\mathcal{D}}$:

$$-M \begin{pmatrix} y_1 \\ \vdots \\ y_{d-1} \end{pmatrix} \leq \begin{pmatrix} -1 \\ \vdots \\ -1 \end{pmatrix}$$

We use Fourier-Motzkin elimination to test the consistency of this system (see chapter 2). The construction of Y fails if the system is inconsistent (e.g. we have $1 \leq y_1$ and $y_1 \leq -1$ for the example above). If the system is consistent, however, any rational solution $\vec{y}^r$, which can be obtained as side-effect of the elimination, can be scaled to an integer solution of which the components are relatively prime (viz. $\vec{y} = \lambda \cdot \vec{y}^r$ for $\lambda \geq 1$). Thereafter, the extended completion method is used to construct a unimodular matrix Y with $\vec{y} \in \mathcal{Z}^{d-1}$ as first row and the corresponding inverse. Obviously, $Y\tilde{U}\vec{d} \succ_1 \vec{0}$ holds for all $\vec{d} \in \mathcal{D} - \tilde{\mathcal{D}}$.

Summary

Summarizing, the following steps are applied. First, we construct the objective matrix S using the subscripts and preferred access directions. If rank$(S) = d$, then the reshaping method fails (proposition 7.1).

Otherwise, for each preferred iteration direction $\vec{\alpha} \in \Delta_b$, a corresponding $d \times d$ unimodular matrix U of which the last column of U^{-1} is equal to this direction is constructed and the following steps are applied until either the method succeeds or this set has been exhausted, in which case the reshaping method fails:

- Select $z = -1$ if $U\vec{d} \prec \vec{0}$ for any $\vec{d} \in \tilde{\mathcal{D}}$ or select $z = 1$ otherwise (corollary 7.1).
- Find a $\vec{y} \in \mathcal{Z}^{d-1}$ with $\gcd(y_1, \ldots, y_{d-1}) = 1$ such that $\vec{y} \cdot \tilde{U}\vec{d} > 0$ for all $\vec{d} \in \mathcal{D} - \tilde{\mathcal{D}}$.

If this $\vec{y} \in \mathcal{Z}^{d-1}$ exists, then the extended completion is used to construct a unimodular matrix Y with this vector as first row and the corresponding inverse. The resulting matrices Y, Y^{-1} and integer $z \in \{-1, +1\}$ define matrices V and V^{-1} according to (7.3) such that $V\vec{d} \succeq \vec{0}$ holds for all $\vec{d} \in \mathcal{D}$. Hence, application of the loop transformation defined by this matrix is valid and reshapes the access patterns of each ith occurrence along the preferred direction $\vec{s}^{\,i} \in \mathcal{Z}^2$.

Examples of Reshaping in Double Loops

Example: Consider the following loop with index vector $\vec{\mathtt{I}} = (\mathtt{I}_1, \mathtt{I}_2)^T$, where the subscripts of both occurrences of the two-dimensional array A are represented by $F(\vec{\mathtt{I}})$:

```
DO I1 = 1, 100
  DO I2 = 1, I1
    A(I1,I2) = A(I1,I2) * 3.0
  ENDDO
ENDDO
```

$$F(\vec{\mathtt{I}}) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \vec{\mathtt{I}}$$

Below, we present the construction of three loop transformations enforcing row-, column-, or regular true diagonal-wise access patterns respectively for array A. In the first step, one of the following matrices S is constructed:

row-wise$(\vec{s} = (0,1)^T)$: column-wise$(\vec{s} = (1,0)^T)$: diagonal-wise$(\vec{s} = (1,1)^T)$:

$$S = (1,0)\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \qquad S = (0,-1)\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \qquad S = (1,-1)\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Reducing S^T into echelon form according to $E = RS^T$ is done as shown below:

row-wise: column-wise: diagonal-wise:

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} S^T \quad \begin{pmatrix} -1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} S^T \quad \begin{pmatrix} -1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} S^T$$

In each case we have rank$(S) = 1$. Hence, for $\lambda_2 = 1$ in the equation $\vec{\alpha} = [(0, \lambda_2)R]^T$, these three matrices define the sets $\Delta^b = \{(0,1)^T\}$, $\Delta^b = \{(1,0)^T\}$, and $\Delta^b = \{(1,1)^T\}$ respectively. The extended completion method is used to obtain a matrix U for which the last column of U^{-1} consists of $\vec{\alpha} \in \Delta^b$:

row-wise:		column-wise:		diagonal-wise:	
U	$= \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$	U	$= \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$	U	$= \begin{pmatrix} -1 & 1 \\ 0 & 1 \end{pmatrix}$
U^{-1}	$= \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$	U^{-1}	$= \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$	U^{-1}	$= \begin{pmatrix} -1 & 1 \\ 0 & 1 \end{pmatrix}$

Not surprisingly, row-wise access patterns are obtained by application of the identity mapping. New code, however, must be generated to implement the other transformations:

column-wise:

```
DO I'1 = 1, 100
  DO I'2 = I'1, 100
    A(I'2,I'1) = A(I'2,I'1) * 3.0
  ENDDO
ENDDO
```

diagonal-wise:

```
DO I'1 = -99, 0
  DO I'2 = 1, I'1 + 100
    A(I'2-I'1,I'2) = A(I'2-I'1,I'2) * 3.0
  ENDDO
ENDDO
```

In general, data dependences have to be accounted for. For double loops, the reshaping method can be formulated as follows: if for a unimodular matrix U for which the last column of U^{-1} is equal to the preferred iteration direction $\vec{\alpha} \in \mathcal{Z}^2$ there is an integer $y \in \{-1, +1\}$ such that $y \cdot (u_{11}, u_{12}) \cdot \vec{d} \geq 0$ for all $\vec{d} \in \mathcal{D}$, then an integer $z \in \{-1, +1\}$ exists such that following matrix V defines a valid transformation performing the reshaping:

$$V = \begin{pmatrix} y & 0 \\ 0 & z \end{pmatrix} U \qquad V^{-1} = U^{-1} \begin{pmatrix} y & 0 \\ 0 & z \end{pmatrix}$$

Hence, the possibility of reshaping in a double loop solely depends on the existence of this integer $y \in \{-1, +1\}$.

Example: Suppose that in the following fragment for which we assume that the dependence structure is represented by $\mathcal{D} = \{(1,0)^T, (0,1)^T\}$, we want to reshape the access patterns of the occurrence of A along $\vec{s} = (1,1)^T$:

```
DO I1 = 2, 5
  DO I2 = 2, 5
    A(I1,I2) = ...
  ENDDO
ENDDO
```

$$F(\vec{\mathtt{I}}) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \vec{\mathtt{I}}$$

Application of the reshaping method yields $\Delta_b = \{(1,1)^T\}$ which gives rise to the construction of the following unimodular matrices:

$$U = \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix} \qquad U^{-1} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$$

Because $y \cdot (1,-1) \cdot (1,0)^T = y$ and $y \cdot (1,-1) \cdot (0,1)^T = -y$, the reshaping method fails. The reason for failure is illustrated in figure 7.2, where the index set of array A is annotated with data dependences. To enforce diagonal-wise access patterns, all iterations in the target iteration space that satisfy $\mathtt{I}_1 - \mathtt{I}_2 = i'_1$ must be accessed before a next value of i'_1 is considered. This kind of traversal is impossible, however, since data dependences impose a cyclic ordering on the corresponding access patterns.

Example: If there exists an appropriate integer $y \in \{-1, +1\}$, the reshaping method is successful. Assume that in the following double loop we want to reshape the access patterns of the occurrences of the two-dimensional arrays A and B along $\vec{s}^{\,1} = (-3,1)^T$ and $\vec{s}^{\,2} = (1,1)^T$ respectively:

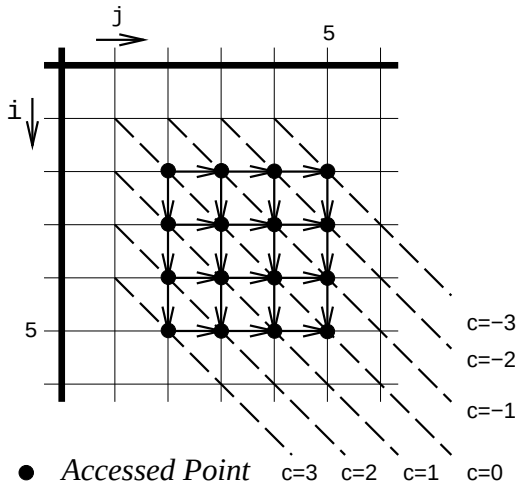


Figure 7.2: Cyclic Ordering on Access Patterns

```
DO I1 = 1, 10
  DO I2 = 1, 10
     A(I1+1,I2+1)  = ...
     B(3*I2,11-I1) = ...
  ENDDO
ENDDO
```

First, we compute the objective matrix S:

$$S = \begin{pmatrix} 1 & 3 \\ 1 & 3 \end{pmatrix} = \begin{pmatrix} 1 & 3 & & \\ & & 1 & -1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ \hline 0 & 3 \\ 1 & 0 \end{pmatrix}$$

Matrix S^T is reduced into echelon form according to $E = RS^T$ as follows:

$$\begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ -3 & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 3 & 3 \end{pmatrix}$$

Hence, since $\Delta_b = \{(-3,1)^T\}$, the following unimodular matrices are computed using the extended completion method:

$$U = \begin{pmatrix} 1 & 3 \\ 0 & 1 \end{pmatrix} \qquad U^{-1} = \begin{pmatrix} 1 & -3 \\ 0 & 1 \end{pmatrix}$$

If $\mathcal{D} = \{(1,-1)^T, (3,-1)^T\}$ represents the data dependence structure of the original loop, then the integers $y = -1$ and $z = -1$ are selected to make the loop transformation defined by $V = -U$ valid (viz. $V(1,-1)^T = (2,1)^T$ and $V(3,-1)^T = (0,1)^T$). In the target loop, the new access directions are $(3,-1)^T$ and $(-3,-3)^T$:

```
DO I1' = -40, 4
  DO I2' = MAX(-10,⌈(I1'+1)/3)⌉, MIN(-1,⌊(I1'+10)/3⌋)
    A(3*I2'-I1'+1,1-I2')   = ...
    B(-3*I2',I1'-3*I2'+11) = ...
  ENDDO
ENDDO
```

Examples of Reshaping in Triple Loops

Example: Consider the following triple loop:

```
DO I1 = 10, 15
  DO I2 = 1, 3
    DO I3 = 10, 15
      A(  I1 + 3*I2 + I3,   I1 + I3) = ...
      B(2*I1 +        I3, 2*I2     ) = ...
      C(  I1 - 3*I2     ,        I3) = ...
    ENDDO
  ENDDO
ENDDO
```

Suppose that all data dependences are represented by the set shown below:

$$\mathcal{D} = \{(1,0,0)^T, (0,1,0)^T, (0,0,1)^T, (3,1,-6)^T\}$$

The subscripts of the occurrences of the two-dimensional arrays A, B, and C are represented by the following three affine transformations:

$$F_1(\vec{\mathtt{I}}) = \begin{pmatrix} 1 & 3 & 1 \\ 1 & 0 & 1 \end{pmatrix} \vec{\mathtt{I}} \quad F_2(\vec{\mathtt{I}}) = \begin{pmatrix} 2 & 0 & 1 \\ 0 & 2 & 0 \end{pmatrix} \vec{\mathtt{I}} \quad F_3(\vec{\mathtt{I}}) = \begin{pmatrix} 1 & -3 & 0 \\ 0 & 0 & 1 \end{pmatrix} \vec{\mathtt{I}}$$

Now, suppose that row-wise true access patterns are preferred for these three occurrences, i.e. $\vec{s}^{\,i} = (0,1)$ for $1 \leq i \leq 3$. Reshaping the access patterns accordingly seems to be a non-trivial task at first sight. However, the reshaping method proceeds as follows.

First, the objective matrix S is constructed:

$$S = \begin{pmatrix} 1 & 0 & & & & \\ & & 1 & 0 & & \\ & & & & 1 & 0 \end{pmatrix} \left(\begin{array}{ccc} 1 & 3 & 1 \\ 1 & 0 & 1 \\ \hline 2 & 0 & 1 \\ 0 & 2 & 0 \\ \hline 1 & -3 & 0 \\ 0 & 0 & 1 \end{array}\right) = \begin{pmatrix} 1 & 3 & 1 \\ 2 & 0 & 1 \\ 1 & -3 & 0 \end{pmatrix}$$

Echelon reduction of S^T yields the following form for $E = RS^T$:

$$\begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & -1 \\ 3 & 1 & -6 \end{pmatrix} \begin{pmatrix} 1 & 2 & 1 \\ 3 & 0 & -3 \\ 1 & 1 & 0 \end{pmatrix}$$

Because $\text{rank}(S) = 2$, all integer solutions of the homogeneous system $S\vec{\alpha} = \vec{0}$ are given by $\vec{\alpha} = [(0,0,\lambda_3)R]^T$ for arbitrary $\lambda_3 \in \mathcal{Z}$. Hence, we have $\Delta_b = \{(3,1,-6)^T\}$. The following matrices are constructed with the extended completion method:

$$U = \begin{pmatrix} -1 & 3 & 0 \\ 0 & 6 & 1 \\ 0 & 1 & 0 \end{pmatrix} \qquad U^{-1} = \begin{pmatrix} -1 & 0 & 3 \\ 0 & 0 & 1 \\ 0 & 1 & -6 \end{pmatrix}$$

Let $\tilde{U}$ denote the matrix consisting of the first 2 rows of U. Then $\tilde{\mathcal{D}} = \{\vec{d} \in \mathcal{D} \mid \tilde{U}\vec{d} = \vec{0}\}$ is equal to $\{(3,1,-6)^T\}$. Since $U(3,1,-6)^T = (0,0,1)^T$, we select $z = 1$.

Finding an integer vector $\vec{y} \in \mathcal{Z}^2$ such that $\vec{y} \cdot \tilde{U}\vec{d} > 0$ for all $\vec{d} \in \mathcal{D} - \tilde{\mathcal{D}}$ is equivalent to solving the following system of linear inequalities:

$$\begin{pmatrix} -1 & 0 \\ 3 & -6 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} \leq \begin{pmatrix} -1 \\ -1 \\ -1 \end{pmatrix}$$

Obviously, this system is consistent, and a solution $(1,1)^T$ can be used directly for $\vec{y}$. The following matrices results:

$$Y = \begin{pmatrix} 1 & 1 \\ -1 & 0 \end{pmatrix} \qquad Y^{-1} = \begin{pmatrix} 0 & -1 \\ 1 & 1 \end{pmatrix}$$

These matrices and $z = 1$ give rise to the following V and V^{-1}:

$$V = \begin{pmatrix} 1 & 3 & 1 \\ -1 & 3 & 0 \\ 0 & 1 & 0 \end{pmatrix} \qquad V^{-1} = \begin{pmatrix} 0 & -1 & 3 \\ 0 & 0 & 1 \\ 1 & 1 & -6 \end{pmatrix}$$

Application of the loop transformation defined by V yields the following target loop in which row-wise access patterns result for all occurrences:

```
DO I'1 = 23, 39
  DO I'2 = MAX(-12,I'1-45,16-I'1), MIN(-1,I'1-30,33-I'1)
    DO I'3 = MAX(⌈(I'1+I'2-15)/6⌉,⌈(I'2+10)/3⌉),
                        MIN(⌊(I'1+I'2-10)/6⌋,⌊(I'2+15)/3⌋)
      A(I'1      , I'1       - 3*I'3) = ...
      B(I'1 - I'2,             2*I'3) = ...
      C(    - I'2, I'1 + I'2 - 6*I'3) = ...
    ENDDO
  ENDDO
ENDDO
```

Example: Below, an example is given in which several possible preferred iteration directions result:

```
DO I1 = 1, 4
  DO I2 = 1, 4
    DO I3 = 1, 4
      A(I3 + 5, I2 -   I1 - 2*I3 + 2) = ...
      B(I3    , I1 + 3*I3 -   I2    ) = ...
    ENDDO
  ENDDO
ENDDO
```

Suppose that we want to reshape the access patterns of both occurrences, of which the subscripts are represented by the following affine transformation, along $\vec{s}^{\,1} = (1,0)$ and $\vec{s}^{\,2} = (1,1)^T$ respectively:

$$F_1(\vec{\mathtt{I}}) = \begin{pmatrix} 5 \\ 2 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 1 \\ -1 & 1 & -2 \end{pmatrix} \vec{\mathtt{I}} \qquad F_2(\vec{\mathtt{I}}) = \begin{pmatrix} 0 & 0 & 1 \\ 1 & -1 & 3 \end{pmatrix} \vec{\mathtt{I}}$$

First, the objective matrix S is constructed:

$$S = \begin{pmatrix} 0 & -1 & & \\ & & 1 & -1 \end{pmatrix} \left(\begin{array}{ccc} 0 & 0 & 1 \\ -1 & 1 & -2 \\ \hline 0 & 0 & 1 \\ 1 & -1 & 3 \end{array} \right) = \begin{pmatrix} 1 & -1 & 2 \\ -1 & 1 & -2 \end{pmatrix}$$

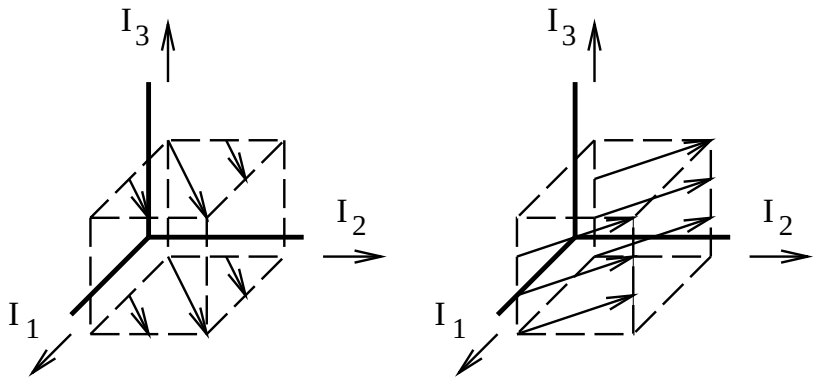


Figure 7.3: Preferred Iteration Directions

Echelon reduction of S^T yields the following form for $E = RS^T$:

$$\begin{pmatrix} -1 & 1 \\ 0 & 0 \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 2 & 1 \end{pmatrix} \begin{pmatrix} 1 & -1 \\ -1 & 1 \\ 2 & -2 \end{pmatrix}$$

Because rank$(S) = 1$, all integer solutions of the homogeneous system $S\vec{\alpha} = \vec{0}$ are given by the formula shown below, for arbitrary $\lambda_2 \in \mathcal{Z}$ and $\lambda_3 \in \mathcal{Z}$:

$$\vec{\alpha} = [(0, \lambda_2, \lambda_3)R]^T$$

We obtain the set $\Delta_b = \{(1,1,0)^T, (0,2,1)^T\}$, which is illustrated in figure 7.3. Now, suppose that the data dependences in this loop are represented by the following set:

$$\mathcal{D} = \{(1,0,0)^T, (0,1,0)^T, (1,1,0)^T, (1,0,1)^T\}$$

First, we try $\vec{\alpha} = (1,1,0)^T$:

$$U = \begin{pmatrix} 1 & -1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \qquad U^{-1} = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

Since $U\vec{d} = (0,0,1)^T$ for $\vec{d} \in \tilde{\mathcal{D}}$ where $\tilde{\mathcal{D}} = \{\vec{d} \in \mathcal{D} \mid \tilde{U}\vec{d} = \vec{0}\} = \{(1,1,0)^T\}$ and $\tilde{U}$ consists of the first 2 rows of U, we select $z = 1$. Thereafter, we determine whether there is a suitable solution of the following system of inequalities:

$$\begin{pmatrix} -1 & 0 \\ 1 & 0 \\ -1 & -1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} \leq \begin{pmatrix} -1 \\ -1 \end{pmatrix}$$

Application of Fourier-Motzkin elimination yields the following sequence:

$$\left(\begin{array}{cc|c} -1 & -1 & -1 \\ 1 & 0 & -1 \\ -1 & 0 & -1 \end{array}\right) \rightarrow \left(\begin{array}{c|c} 1 & -1 \\ -1 & -1 \end{array}\right) \rightarrow (-2)$$

As revealed by the inconsistency of this system, the construction of matrix Y fails because the same problem as illustrated in figure 7.2 occurs.

Thereafter, we try $\vec{\alpha} = (0, 2, 1)^T$:

$$U = \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 2 \\ 0 & 0 & 1 \end{pmatrix} \qquad U^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 2 \\ 0 & 0 & 1 \end{pmatrix}$$

In this case, we have $\tilde{\mathcal{D}} = \emptyset$. Hence, we try to find a vector $\vec{y} \in \mathcal{Z}^2$ that satisfies $\vec{y} \cdot \tilde{U}\vec{d} > 0$ for all $\vec{d} \in \mathcal{D}$, which is equivalent to solving the following system:

$$\begin{pmatrix} -1 & 0 \\ 0 & 1 \\ -1 & 1 \\ -1 & -2 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} \leq \begin{pmatrix} -1 \\ -1 \\ -1 \\ -1 \end{pmatrix}$$

Hence, we may use the solution $\vec{y} = (3, -1)$:

$$\begin{pmatrix} 3 & -1 \\ 1 & 0 \end{pmatrix} \qquad \begin{pmatrix} 3 & -1 \\ -1 & 3 \end{pmatrix}$$

These matrices together with $z = 1$ define the following matrices:

$$V = \begin{pmatrix} 3 & 1 & -2 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad V^{-1} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & -3 & 2 \\ 0 & 0 & 1 \end{pmatrix}$$

Application of the loop transformation defined by V yields the target loop shown below, in which the occurrences of A and B have the appropriate access direction:

```
DO I'1 = -4, 14
  DO I'2 = MAX(1,⌈(I'1-2)/3⌉), MIN(4,⌊(I'1+7)/3⌋)
    DO I'3 = MAX(1,⌈(3*I'2-I'1+1)/2⌉), MIN(4,⌊(3*I'2-I'1+4)/2⌋)
      A(I'3 + 5,   I'1 - 4*I'2 + 2 ) = ...
      B(I'3    , 4*I'2 -   I'1 + I'3) = ...
    ENDDO
  ENDDO
ENDDO
```

7.1.4 Implementation of Reshaping in the Prototype Sparse Compiler

In the prototype sparse compiler, reshaping access patterns is implemented by scanning over all perfected nested *sub*-loops with admissible loop bounds in the dense program. Given such a perfectly nested sub-loop, all occurrences of enveloping data structures with admissible subscripts that only depend on loop indices of the sub-loop are examined. If access summary $\langle X, \vec{x}^{\,n} \rangle \in \mathcal{X}_A$ is associated with such an occurrence, then we choose a preferred access direction $\vec{s} \in \mathcal{Z}^2$ of this occurrence as any $\vec{s} = \vec{p}$ for which there is a property summary $\langle P, \vec{p}, p \rangle \in \mathcal{P}_A$ such that $X \cap P \neq \emptyset$, or we simply discard the occurrence from further consideration otherwise.

Subsequently, the reshaping method presented in previous sections is applied to the perfectly nested sub-loop and the remaining occurrences.[1] On failure, some interaction with the programmer is performed about whether some data dependences or occurrences may be ignored during the reshaping, after which another attempt is possibly taken. On success, the access summary $\langle X, \vec{x}^{\,n} \rangle$ of each occurrence affected by the reshaping is replaced by a new access summary $\langle X, \vec{y}^{\,n} \rangle$ where $\vec{y}^{\,n} \in \mathcal{Z}^2$ denotes the new effective access direction. In this manner, re-computation of simple sections is avoided. Finally, all changes in subscripts of enveloping data structures are correctly accounted for by altering the conditions computed by the method of section 5.3.1 accordingly.

[1] Constructing a valid unimodular transformation is complicated by the fact that only dependence *directions* are computed by the prototype compiler, so that the way of testing validity presented at the end of section 3.3.2 must be used.

7.2 Construction of Representatives

Another important step in the automatic data structure selection and transformation method is the construction of a set $\Sigma_A = \{S_1, S_2, \ldots\}$ of representative simple sections for each implicitly sparse matrix A. These simple sections, called **representatives** for short, should have the following properties:

$$\begin{array}{lll} (1) \quad \forall\, S, S' \in \Sigma_A & : & S \neq S' \Rightarrow S \cap S' = \emptyset \quad \text{(mutually disjoint)} \\ (2) \quad \forall\, \langle X, \vec{x}^{\,n} \rangle \in \mathcal{X}_A & : & \exists\, S \in \Sigma_A : X \subseteq S \quad \text{(representative)} \end{array}$$

Although these constraints are trivially satisfied for a single representative that contains the whole index set of the matrix A, some of the simple sections in $\mathcal{X}_A$ are fragmented using iteration space partitioning to enable the construction of a set Σ_A with a reasonable degree of fragmentation. If possible, simple sections are fragmented according to the simple sections in the property summary set, since it is desirable to have the following property:

$$(3) \quad \forall\, S \in \Sigma_A, \langle P, \vec{p}, p \rangle \in \mathcal{P}_A \quad : \quad S \cap P \neq \emptyset \Rightarrow S \subseteq P \quad \text{(property fragmented)}$$

First, a simple approach is discussed that is based on repetitively combining overlapping simple sections. Since usually only one representative results, we explore how iteration space partitioning, discussed in chapter 3, can be used to increase the resulting amount of fragmentation.

7.2.1 Simple Approach

A simple way to construct the representatives of an implicitly sparse matrix A is to use the union '$\uplus$' to combine overlapping simple sections associated with the occurrences of the corresponding enveloping data structure A, until a set of mutually disjoint representatives remains. Starting with $\Sigma_A = \emptyset$, we scan over all access summaries in the set $\mathcal{X}_A$. For each $\langle X, \vec{x}^{\,n} \rangle \in \mathcal{X}_A$, the following tests and associated actions are performed:

1. If $X \cap S = \emptyset$ for all $S \in \Sigma_A$, then X is added to the set Σ_A.
2. Otherwise, we have $X \cap S \neq \emptyset$ for some $S \in \Sigma_A$.
   (a) If $X \subseteq S$, then S is used directly as representative of X.
   (b) If $X \not\subseteq S$, then $\overline{X} = X \uplus S$ is computed and S is deleted from Σ_A. If $\overline{X} \cap S \neq \emptyset$ still holds for some (other) $S \in \Sigma_A$, then this step is repeated until $\overline{X} \cap S = \emptyset$ for all $S \in \Sigma_A$. The final $\overline{X}$ is inserted into the set Σ_A.

Example: Consider the following occurrences of a two-dimensional array A that is used as the enveloping data structure of an 100×100 implicitly sparse matrix A:

```
DO I = 1, 100
  DO J = 1, I-1
    A1(I,J) = C(I,J)
  ENDDO
  B(I) = A2(I,I)
  DO J = I, 100
    A3(I,J) = D(I,J)
  ENDDO
ENDDO
```

This fragment gives rise to the following access summary bag:

$$\mathcal{X}_A = \{\langle X_1, (0,1)^T \rangle, \langle X_2, (1,1)^T \rangle, \langle X_3, (0,1)^T \rangle\}$$

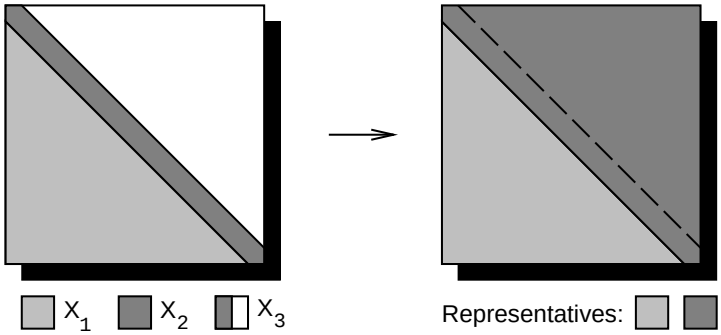


Figure 7.4: Resulting Representatives

The following simple sections, illustrated in figure 7.4, appear in this bag:

$$\left\{ \begin{array}{lcll} X_1 & = & \{(i,j) \in \mathcal{Z}^2 \mid (2,1,3,1)^T & \leq \mathcal{M}(i,j)^T \leq \ (100,99,199,99)^T \} \\ X_2 & = & \{(i,j) \in \mathcal{Z}^2 \mid (1,1,2,0)^T & \leq \mathcal{M}(i,j)^T \leq \ (100,100,200,0)^T \} \\ X_3 & = & \{(i,j) \in \mathcal{Z}^2 \mid (1,1,2,-99)^T & \leq \mathcal{M}(i,j)^T \leq \ (100,100,200,0)^T \} \end{array} \right.$$

Because initially the set Σ_A is empty, no overlap is detected for the first simple section and X_1 is inserted into this set. Moreover, because $X_1 \cap X_2 = \emptyset$, the simple section X_2 can also be used as representative and we obtain $\Sigma_A = \{X_1, X_2\}$. Finally, we have $X_2 \cap X_3 \neq \emptyset$ and $X_3 \not\subseteq X_2$. Therefore, we compute $\overline{X} = X_2 \uplus X_3 = X_3$ and delete X_2 from Σ_A. Since $X_1 \cap \overline{X} = \emptyset$ for the only remaining representative X_1, we add $\overline{X} = X_3$ to Σ_A.

The final set $\Sigma_A = \{X_1, X_3\}$ represents the index set of the strict lower triangular and upper triangular part of the matrix A, as illustrated in the second picture of figure 7.4. Note that if X_3 would be inserted before X_2, then $X_2 \subseteq X_3$ would imply that X_3 could be used directly as representative. In general, because the union computed by `combine` is associative [15], the representatives computed by this simple approach are independent of the order in which the access summaries in $\mathcal{X}_A$ are considered.

7.2.2 Improved Approach

The disadvantage of the simple approach is that it is very likely that only a few representatives result for each implicitly sparse matrix. Typical representatives that arise describe the index set of, for instance, the strict lower or upper triangular parts. It is also likely that only one representative describing the whole index set of the matrix results. Therefore, we explore how iteration space partitioning can be used to increase the amount of resulting fragmentation.

Intuition Behind Iteration Space Partitioning Support

Before we compute the union in step (2)b of our simple approach, we can substitute the subscripts of the occurrences to which $\langle X, \vec{x}^{\,n} \rangle$ with $X \cap S \neq \emptyset$ for some $S \in \Sigma_A$ belongs, for the expression (i,j) in the inequalities that define the intersection $X \cap S$. The resulting inequalities on the loop indices define the part of the iteration space in which elements with indices in this intersection are accessed.

Example: Substituting the admissible subscripts (`I,J`) of the third occurrence in the previous example for (i,j) in the inequalities that define $X_2 \cap X_3$ yields the following system:

$$\begin{cases} 1 \leq \mathtt{I} \leq 100 \\ 1 \leq \mathtt{J} \leq 100 \\ 2 \leq \mathtt{I}+\mathtt{J} \leq 200 \\ 0 \leq \mathtt{I}-\mathtt{J} \leq 0 \end{cases} \quad \text{Simplification} \rightarrow \quad \begin{cases} 1 \leq \mathtt{I} \leq 100 \\ \mathtt{I} \leq \mathtt{J} \leq \mathtt{I} \end{cases}$$

Confining overlap to $X_2 \cap X_3$ can be done by isolating the loop-body in which the third occurrence appears for all iterations satisfying $1 \leq \mathtt{I} \leq 100$ and $\mathtt{I} \leq \mathtt{J} \leq \mathtt{I}$:

```
DO I = 1, 100
  DO J = 1, I-1
    A1(I,J) = C(I,J)
  ENDDO
  B(I) = A2(I,I)
  DO J = I, 100
    A3(I,J) = D(I,J)
  ENDDO
ENDDO
```

$\mathtt{I} = \mathtt{J}$ $\rightarrow$

```
DO I = 1, 100
  DO J = 1, I-1
    A1(I,J) = C(I,J)
  ENDDO
  B(I) = A2(I,I)
  A3a(I,I) = D(I,I)
  DO J = I+1, 100
    A3b(I,J) = D(I,J)
  ENDDO
ENDDO
```

Obviously, after application of a relatively simple transformation, the index sets of the main diagonal and the strict lower and upper triangular part of the matrix can be selected as representatives, as illustrated with a dashed line in the second picture of figure 7.4.

Although iteration space partitioning can be used to increase the number of resulting representatives, the code size may also increase. Another disadvantage arises if iteration space partitioning induces redundant fragmentation of the simple sections that are associated with other occurrences. Although loop distribution can be used to limit this effect, in general, we must find a balance between code duplication and this redundant fragmentation on one side, and the amount of useful fragmentation on the other side.

Outline of the Improved Approach

Rather than starting the construction of representatives with $\Sigma_A = \emptyset$, in the improved approach we first insert the simple sections arising in the property set $\mathcal{P}_A$ into this set, i.e. for each $\langle P, \vec{p}, p \rangle \in \mathcal{P}_A$, the simple section P is added to Σ_A. In this manner, the simple sections associated with the occurrences of the corresponding enveloping data structure A may become fragmented according to the simple sections arising in $\mathcal{P}_A$.

Subsequently, we scan over all the access summaries in the bag $\mathcal{X}_A$ in increasing order of the simple section size, because small simple sections tend to induce fragmentation of larger simple sections (note that once a representative has been added to the set Σ_A, it cannot be further fragmented because this representative may represent the simple sections of *many* occurrences). For each access summary in $\mathcal{X}_A$, the steps of the simple approach of section 7.2.1 are performed. However, before we combine simple sections in step (2)b, we proceed as follows.

Let $IS \subseteq \mathcal{Z}^d$ and $\vec{\mathtt{I}}$ denote respectively the iteration space and the index vector of the loop in which this occurrence appears. Furthermore, suppose that the subscripts of the occurrence to which the access summary $\langle X, \vec{x}^{\,n} \rangle$ with $X \cap S \neq \emptyset$ for some $S \in \Sigma_A$ belongs are admissible, and represented by $F(\vec{\mathtt{I}}) = \vec{v} + W\vec{\mathtt{I}}$. Now, our goal is to isolate the loop-body of the loop for all iterations lying in the following set:

$$\{\vec{\mathtt{I}} \in IS \mid F(\vec{\mathtt{I}}) \in X \cap S\} \tag{7.5}$$

If $\vec{\sigma} \in \mathcal{Z}^4$ and $\vec{\tau} \in \mathcal{Z}^4$ denote the boundary values of the simple section $X \cap S$, substituting $F(\vec{\mathtt{I}})$ for (i, j) in the corresponding set of inequalities yields the following system:[2]

[2] If only *one* subscript is admissible, still one of the first two pairs can be constructed.

$$\begin{array}{rrcrcl} \sigma_1 \leq & v_1 & + & \sum_{j=1}^{d} & w_{1j} \cdot \mathtt{I}_j & \leq \tau_1 \\ \sigma_2 \leq & v_2 & + & \sum_{j=1}^{d} & w_{2j} \cdot \mathtt{I}_j & \leq \tau_2 \\ \sigma_3 \leq & v_1 + v_2 & + & \sum_{j=1}^{d} & (w_{1j} + w_{2j}) \cdot \mathtt{I}_j & \leq \tau_3 \\ \sigma_4 \leq & v_1 - v_2 & + & \sum_{j=1}^{d} & (w_{1j} - w_{2j}) \cdot \mathtt{I}_j & \leq \tau_4 \end{array} \tag{7.6}$$

For appropriate $l^k, a_j^k, u^k \in \mathcal{Z}$, the kth pair of linear inequalities in this system gives rise to the definition of a slice $C^k \subseteq \mathcal{Z}^d$ that can be written in the following form:

$$C^k = \{\vec{\mathtt{I}} \in \mathcal{Z}^d \mid l^k \leq a_1^k \cdot \mathtt{I}_1 + \ldots + a_d^k \cdot \mathtt{I}_d \leq u^k\}$$

Rather than using all linear inequalities simultaneously to isolate the loop-body for all iterations in (7.5), partitioning the iteration space according to only one of these slices enables the incremental construction of new simple sections into which the simple section X becomes fragmented, which prevents re-computation of these simple sections by means of subscripts and loop bounds analysis. Moreover, using only one slice provides more control over the amount of resulting fragmentation, since it enables the compiler to determine whether further iteration space partitioning is useful. If this is not the case or if iteration space partitioning fails, then X is combined with representatives in Σ_A as in the simple approach. Otherwise, X is replaced by the new simple section, each of which is handled separately thereafter.

These issues are further elaborated upon in the following sections.

Incremental Construction of New Simple Sections

Central to incrementally constructing new simple sections is the procedure `alter`, in which a simple section stored in `new` is obtained from a simple section stored in `old` by refining all boundaries after the boundary values of the `k`th boundary pair have been replaced by `l` and `u`:

```
procedure alter(old, k, l, u, var new)
begin
 new      := old;
 new.l[k] := l;
 new.u[k] := u;
 refine(new)
end
```

If we partition the iteration space according to the kth slice $C^k \subseteq \mathcal{Z}^d$ that is defined by system (7.6), then at most three duplicates of the occurrence are generated. Obviously, this iteration space partitioning fragments X into three new simple sections that become associated with the duplicates. These new simple section can be computed in `s1`, `s2`, and `s3` by calling the following procedure `increment` with X, k, and $X \cap S$ as first three arguments:

```
procedure increment(s, t, k, swp, var s1, var s2, var s3)
begin
  alter(s, k, s.l[k],   t.l[k]-1, s1);
  alter(s, k, t.l[k],   t.u[k],   s2);
  alter(s, k, t.u[k]+1, s.u[k],   s3);
  if (swp) then
    swap(s1, s3);
  endif
end
```

If the last nonzero coefficient $a_i^k \in \mathcal{Z}$ defining $C^k \subseteq \mathcal{Z}^d$ is negative, then we must swap the first and last simple section to restore the association between the new simple sections and the three duplicates of the original occurrence in the resulting DO-loops. In this case, the boolean variable `swp` is set.

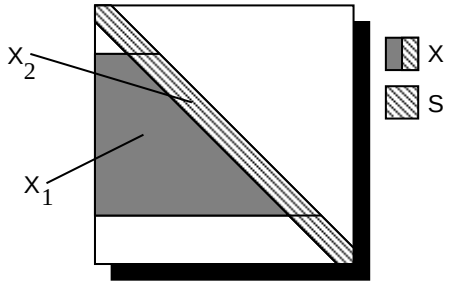


Figure 7.5: Fragmentation

Example: Suppose that we initialize the set of representatives of a 100×100 implicitly sparse matrix A to the index set of the main diagonal, i.e. we have $\Sigma_A = \{S\}$ for the following simple section:

$$S = \{(i,j) \in \mathcal{Z}^2 \mid (1,1,2,0)^T \leq \mathcal{M}(i,j)^T \leq (100,100,200,0)^T\}$$

Now, consider the following occurrence of the enveloping data structure of A:

```
DO I = 0, 80
  DO J = 1, 10+I
    A(10+I,J) = ...
  ENDDO
ENDDO
```

Obviously, this occurrence gives rise to the access summary bag $\mathcal{X}_A = \{\langle X, (0,1)^T\rangle\}$, where the simple section in the only access summary has the following form:

$$X = \{(i,j) \in \mathcal{Z}^2 \mid (10,1,11,0)^T \leq \mathcal{M}(i,j)^T \leq (90,90,180,89)^T\}$$

Because $X \cap S \neq \emptyset$ and $X \not\subseteq S$, as illustrated in figure 7.5, we substitute $(\mathtt{I}+\mathtt{10}, \mathtt{J})$ for (i,j) in the inequalities defining $X \cap S$ (in which some boundaries have been refined):

$$\begin{array}{lrcrcr}
(1) & \mathtt{0} & \leq & \mathtt{I} & \leq & \mathtt{90} \\
(2) & \mathtt{10} & \leq & \mathtt{J} & \leq & \mathtt{90} \\
(3) & \mathtt{10} & \leq & \mathtt{I}+\mathtt{J} & \leq & \mathtt{170} \\
(4) & -\mathtt{10} & \leq & \mathtt{I}-\mathtt{J} & \leq & -\mathtt{10}
\end{array}$$

Subsequently, we partition the iteration space of the double loop according to one of these pairs. For instance, the 4th pair of inequalities induces the following transformation:

```
DO I = 0, 80                          DO I = 0, 80
  DO J = 1, 10+I                        DO J = 1, 9+I
    A(10+I,J) = ...      J = I + 10       A1(10+I,J) = ...
  ENDDO                    →            ENDDO
ENDDO                                   A2(10+I,10+I) = ...
                                      ENDDO
```

The simple sections associated with the three resulting occurrences (of which the third is not generated because execution set $[\mathtt{11}+\mathtt{I}, \mathtt{10}+\mathtt{I}]$ defines a zero trip loop) are computed by calling `increment` with X, $X \cap S$, and 4 as first three arguments. Effectively, the three resulting simple sections are obtained from X by boundary refinement after the 4th pair has been replaced with $0 \leq i-j \leq -1$, $0 \leq i-j \leq 0$, and $1 \leq i-j \leq 89$ respectively. The simple sections X_1 and X_3 have been swapped to account for the negative coefficient of index `J` in the inequalities:

$$\left\{\begin{array}{rcl} X_1 & = & \{(i,j) \in \mathcal{Z}^2 \mid (10,1,11,\mathbf{1})^T \leq \mathcal{M}(i,j)^T \leq (90,89,179,\mathbf{89})^T\} \\ X_2 & = & \{(i,j) \in \mathcal{Z}^2 \mid (10,10,20,\mathbf{0})^T \leq \mathcal{M}(i,j)^T \leq (90,90,180,\mathbf{0})^T\} \\ X_3 & = & \{(i,j) \in \mathcal{Z}^2 \mid (10,1,11,\mathbf{0})^T \leq \mathcal{M}(i,j)^T \leq (90,90,180,-\mathbf{1})^T\} \end{array}\right.$$

As illustrated in figure 7.5, simple section X becomes fragmented into X_1, $X_2 = X \cap S$ and $X_3 = \emptyset$ (associated with the duplicate in the zero-trip loop).

7.2.3 Implementation of Representative Construction

Using only one pair of inequalities of the system (7.6) to partition an iteration space not only enables the incremental construction of the new simple sections, but it also enables the sparse compiler to determine whether further iteration space partitioning is useful. Once the system (7.6) has been determined for an occurrence of which the associated simple section $X \subseteq \mathcal{Z}^2$ satisfies $X \cap S \neq \emptyset$ and $X \not\subseteq S$ for some $S \in \Sigma_A$, we proceed as follows.

Procedure `increment` is used to determine the new simple sections $X_1^k \subseteq \mathcal{Z}^2$, $X_2^k \subseteq \mathcal{Z}^2$, and $X_3^k \subseteq \mathcal{Z}^2$ into which X becomes fragmented if the iteration space is partitioned according to the kth pair of inequalities. We define the **potential gain** of each such iteration space partitioning as follows:

$$\mathcal{G}_k = |X| - |X_2^k|$$

Let k' denote the pair of inequalities that induces an execution set partitioning on the *outermost* DO-loop of all the inequalities k that satisfy $\mathcal{G}_k \geq t$ for some threshold $t \in Z$ (the one with the largest potential gain is used on ties). The threshold is used to determine whether further iteration space partitioning is useful. In order to keep fragmentation proportional to the size of access patterns, a value $t \approx max(m,n)$ seems appropriate for an $m \times n$ implicitly sparse matrix. Furthermore, as for general iteration space partitioning, first applying execution set partitioning to more outer DO-loops eventually induces the least increase in code size.

If such a k' does not exist or if iteration space partitioning fails (because execution set partitioning is not applicable), then another representative $S \in \Sigma_A$ with $X \cap S \neq \emptyset$ and $X \not\subseteq S$ is considered until either the set Σ_A has been exhausted or iteration space partitioning becomes possible. In the former case, we repetitively combine X with representatives in Σ_A according to step 2(b) of the simple approach (see section 7.2.1). In the latter case, we partition the iteration space according to the k'th pair of inequalities and perform the following steps to deal with the corresponding program transformations, after which another access summary is processed using the improved approach:

1. For the occurrence to which X belongs, the original access summary $\langle X, \vec{x}^{\,n} \rangle$ in the bag $\mathcal{X}_A$ is replaced by the new access summaries $\langle X_1^{k'}, \vec{x}^{\,n} \rangle$, $\langle X_2^{k'}, \vec{x}^{\,n} \rangle$, and $\langle X_3^{k'}, \vec{x}^{\,n} \rangle$, which become associated with the duplicates of this occurrence.

2. For all other occurrences of (possibly different) enveloping data structures that are duplicated, identical copies of the original access summary are associated with the duplicates of this occurrence, thereby replacing the original in the corresponding summary bag.

Obviously, if a duplicate occurrence appears in a zero trip loop, the corresponding new access summary is discarded. Moreover, the normalized access direction may change if loop unrolling is applied, which must be accounted for in the corresponding new access summary. These steps avoid re-computing simple sections by means of subscript and loop bounds analysis.

Because, in general, it is not straightforward to determine how other simple sections become fragmented, step 2 simply ignores any fragmentation of other simple sections. Moreover, all redundant fragmentation is ignored, whereas useful fragmentation is eventually accounted for anyway. In the latter case, some subsequent transformations have no impact on the code, but just induces incremental construction of the appropriate simple sections.

Finally, the method of section 5.3.1 is used to associate conditions with all new statements or statements of which the condition may alter due to the previous transformations.

Examples of Construction of Representatives

Example: Consider the following fragment in which there are two occurrences of the enveloping data structure B of an implicitly sparse 10×10 matrix B, where we use the threshold $t = 10$:

```
DO I = 1, 7
  B1(I,10) = 20.0
ENDDO
DO I = 1, 10
  DO J = 1, 10
    C(I,J) = B2(I,J)
  ENDDO
ENDDO
```

The corresponding access summary bag $\mathcal{X}_B = \{\langle X_1, (-1,0)^T\rangle, \langle X_2, (0,1)^T\rangle\}$ contains the following simple sections:

$$\begin{cases} X_1 &= \{(i,j) \in \mathcal{Z}^2 \mid (1,10,11,-9)^T \leq \mathcal{M}(i,j)^T \leq (7,10,17,-3)^T\} \\ X_2 &= \{(i,j) \in \mathcal{Z}^2 \mid (1,1,2,-9)^T \leq \mathcal{M}(i,j)^T \leq (10,10,20,9)^T\} \end{cases}$$

If no properties of B are known, we start with $\Sigma_B = \emptyset$. The smallest simple section X_1 is considered first and directly inserted into this set. Thereafter, $X_1 \cap X_2 \neq \emptyset$ and $X_2 \not\subseteq X_1$ hold. At this stage, we have the choice between combining these simple sections into the whole index set, or partitioning the iteration space according to one of the following pairs of inequalities, obtained by substituting (I, J) for (i, j) in the inequalities that define $X_1 \cap X_2$:

				$\lvert X_1^k \rvert$	$\lvert X_2^k \rvert$	$\lvert X_3^k \rvert$	$\mathcal{G}_k$	
(1)	1 ≤	I	≤ 7	0	70	30	30	←
(2)	10 ≤	J	≤ 10	90	10	0	90	
(3)	11 ≤	I+J	≤ 17	45	49	6	51	
(4)	-9 ≤	I-J	≤ -3	72	28	0	72	

Each potential gain exceeds the threshold $t = 10$. Since the first inequality induces an execution set partitioning of the most outer DO-loop, the following transformation is applied:

```
DO I = 1, 10
  DO J = 1, 10
    C(I,J) = B2(I,J)
  ENDDO
ENDDO
```

$1 \leq I \leq 7$ →

```
DO I = 1, 7
  DO J = 1, 10
    C(I,J) = B2a(I,J)
  ENDDO
ENDDO
DO I = 8, 10
  DO J = 1, 10
    C(I,J) = B2b(I,J)
  ENDDO
ENDDO
```

After this loop transformation, we obtain the following altered access summary bag:

$$\mathcal{X}_B = \{\langle X_1, (-1,0)^T\rangle, \langle X_{2a}, (0,1)^T\rangle, \langle X_{2b}, (0,1)^T\rangle\}$$

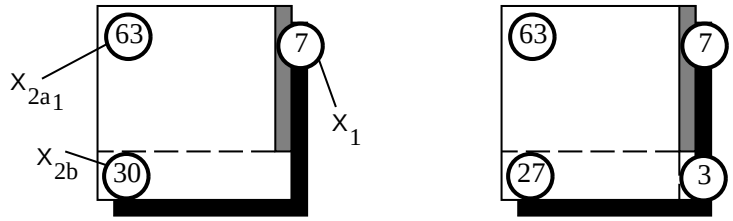


Figure 7.6: Resulting Representatives

In this new access summary bag, $X_{2a} = X_2^1$ and $X_{2b} = X_3^1$ holds for the three incrementally constructed simple sections shown below (with respectively 0, 70 and 30 elements), where X_1^1 is associated with the duplicate of occurrence $\mathtt{A}_3$ in the zero-trip loop having the execution set $[\mathtt{1}, \mathtt{0}]$:

$$\left\{ \begin{array}{lcl} X_1^1 & = & \{(i,j) \in \mathcal{Z}^2 \mid (\mathbf{1},1,2,-9)^T \leq \mathcal{M}(i,j)^T \leq (\mathbf{0},10,20,9)^T\} \\ X_2^1 & = & \{(i,j) \in \mathcal{Z}^2 \mid (\mathbf{1},1,2,-9)^T \leq \mathcal{M}(i,j)^T \leq (\mathbf{7},10,17,6)^T\} \\ X_3^1 & = & \{(i,j) \in \mathcal{Z}^2 \mid (\mathbf{8},1,9,-2)^T \leq \mathcal{M}(i,j)^T \leq (\mathbf{10},10,20,9)^T\} \end{array} \right.$$

Since $X_{2b} \cap X_1 = \emptyset$ holds for X_{2b}, we obtain $\Sigma_B = \{X_1, X_{2b}\}$ thereafter. However, we still have $X_1 \cap X_{2a} \neq \emptyset$ and $X_{2a} \not\subseteq X_1$.

Substituting $(\mathtt{I}, \mathtt{J})$ for (i,j) in the inequalities that define this intersection gives rise to the same set of inequalities, but with different associated potential gains:

				$\lvert X_1^k \rvert$	$\lvert X_2^k \rvert$	$\lvert X_3^k \rvert$	$\mathcal{G}_k$	
(1)	`1` $\leq$	`I`	$\leq$ `7`	0	70	0	0	
(2)	`10` $\leq$	`J`	$\leq$ `10`	63	7	0	63	$\leftarrow$
(3)	`11` $\leq$	`I+J`	$\leq$ `17`	42	28	0	42	
(4)	`-9` $\leq$	`I-J`	$\leq$ `-3`	42	28	0	42	

Although the first pair still induces an execution set partitioning of the most outer DO-loop, the associated potential gain is below the threshold (viz. $\mathcal{G}_1 = 0$). The other pairs all induce an execution set of the J-loop, and the one with the largest potential gain is used:

```
DO J = 1, 10
  C(I,J) = B2a(I,J)
ENDDO
```

$10 \leq \mathtt{J} \leq 10$ $\rightarrow$

```
DO J = 1, 9
  C(I,J) = B2a1(I,J)
ENDDO
C(I,10) = B2a2(I,10)
```

This transformation further alters the access summary bag (note that a normalized direction changes due to loop unrolling):

$$\mathcal{X}_B = \{\langle X_1, (-1,0)^T\rangle, \langle X_{2a_1}, (0,1)^T\rangle, \langle X_{2a_2}, (-1,0)^T\rangle, \langle X_{2b}, (0,1)^T\rangle\}$$

In this new access summary bag, we have $X_{2a_1} = X_1^2$ and $X_{2a_2} = X_2^2$ for the following incrementally constructed simple sections, where X_3^2 is associated with the duplicate of $\mathtt{A}_{2a}$ in the zero trip loop having the execution set $[\mathtt{11}, \mathtt{10}]$:

$$\left\{ \begin{array}{lcl} X_1^2 & = & \{(i,j) \in \mathcal{Z}^2 \mid (1,\mathbf{1},2,-8)^T \leq \mathcal{M}(i,j)^T \leq (7,\mathbf{9},16,6)^T\} \\ X_2^2 & = & \{(i,j) \in \mathcal{Z}^2 \mid (1,\mathbf{10},11,-9)^T \leq \mathcal{M}(i,j)^T \leq (7,\mathbf{10},17,-3)^T\} \\ X_3^2 & = & \{(i,j) \in \mathcal{Z}^2 \mid (1,\mathbf{11},2,-9)^T \leq \mathcal{M}(i,j)^T \leq (7,\mathbf{10},17,6)^T\} \end{array} \right.$$

Because $X_1 \cap X_{2a_1} = \emptyset$, $X_{2a_1} \cap X_{2b} = \emptyset$, and $X_{2a_2} \subseteq X_1$ hold thereafter, eventually we obtain the set $\Sigma_B = \{X_1, X_{2b}, X_{2a_1}\}$ as illustrated in the first picture of figure 7.6.

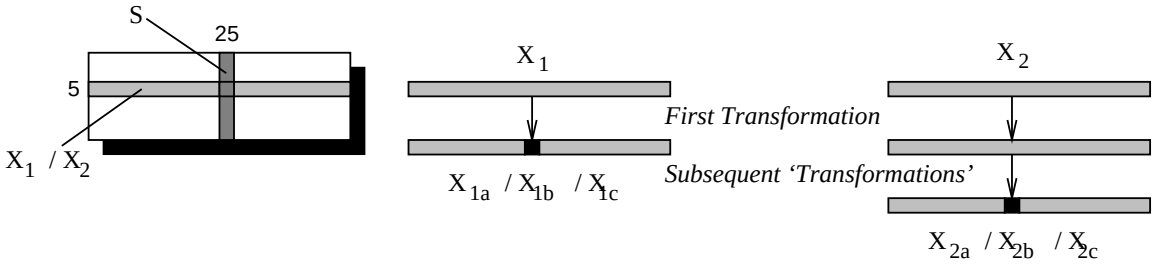


Figure 7.7: Accounting for Useful Fragmentation

If we would eagerly select the pair of inequalities inducing the largest potential gain at each step (rather than one with $\mathcal{G}_k \geq t$ that induces an execution set partitioning of the most *outer* DO-loop), at the first step we would have used the inequalities $\texttt{10} \leq \texttt{J} \leq \texttt{10}$. Because the execution set of the `I`-loop is partitioned thereafter, the four representatives shown in the second picture of figure 7.6 would result due to the unnecessary partitioning of the execution set of the `J`-loop for $\texttt{I} \in [\texttt{8}, \texttt{10}]$. Obviously, if the column would appear somewhere in the 'middle' of the matrix, this effect would become worse (viz. 5 vs. 9), which clearly illustrates the usefulness of partitioning the execution sets of more outer DO-loops first.

Example: Suppose that we start with $\Sigma_A = \{S\}$, where S contains the index set of the 25th column in a 10×50 implicitly sparse matrix A. Now, suppose that the following fragment is considered with the access summary bag $\mathcal{X}_B = \{\langle X_1, (0,1)^T\rangle, \langle X_2, (0,1)^T\rangle\}$:

```
DO J = 1, 50
  A1(5,J) = A2(5,J) * 2.0
ENDDO
```

As illustrated in figure 7.7, intersection $X_1 \cap S \neq \emptyset$ induces the following loop transformation:

```
DO J = 1, 50                          J = 25
  A1(5,J) = A2(5,J) * 2.0               →
ENDDO
```

```
DO J = 1, 24
  A1a(5,J) = A2a(5,J) * 2.0
ENDDO
A1b(5,25) = A2b(5,25) * 2.0
DO J = 26, 50
  A1c(5,J) = A2c(5,J) * 2.0
ENDDO
```

Thereafter, the appropriate simple sections X_{1a}, X_{1b}, and X_{1c} are constructed incrementally, which is illustrated in figure 7.7. Because for simplicity, however, a copy of X_2 is associated with the occurrences $\texttt{A}_{2a}$, $\texttt{A}_{2b}$, and $\texttt{A}_{2c}$, it seems that useful fragmentation is not accounted for.

However, it can be easily verified that $X_{2a} \cap S \neq \emptyset$, $X_{2b} \cap S \neq \emptyset$ and $X_{2c} \cap S \neq \emptyset$, where $X_{2a} = X_{2b} = X_{2c} = X_2$ still holds. This induces an iteration space partitioning according to $\texttt{25} \leq \texttt{J} \leq \texttt{25}$, $\texttt{25} \leq \texttt{25} \leq \texttt{25}$, and $\texttt{25} \leq \texttt{J} \leq \texttt{25}$ for respectively the first loop, the scalar-statement, and the second loop. Obviously, none of the corresponding loop transformations has impact on the code, but just induces the incremental construction of the appropriately fragmented simple sections (see figure 7.7). Eventually, we obtain $\Sigma_A = \{S, X_{1a}, X_{1c}\}$ (viz. $X_{1b} \subseteq S$).

Example: Finally, consider the following fragment with two occurrences of the enveloping data structure of an implicitly sparse 100×100 matrix A:

```
DO I = 1, 50
  B(I) = A1(I+25,I+10)
  DO J = 1, 50
    C(I,J) = A2(I+J,J)
  ENDDO
ENDDO
```

This fragment gives rise to the access summary bag $\mathcal{X}_A = \{\langle X_1, (1,1)^T\rangle, \langle X_2, (1,1)^T\rangle\}$. The two simple sections consist of the index set of 50 elements along the 15th diagonal below the main diagonal and the index set of a trapezoidal part with 2500 elements, as illustrated in figure 7.8:

$$\begin{array}{lll} X_1 = \{(i,j) \in \mathcal{Z}^2 \mid & (26,11,37,15)^T & \leq \mathcal{M}(i,j)^T \leq \quad (75,60,135,15)^T\} \\ X_2 = \{(i,j) \in \mathcal{Z}^2 \mid & (2,1,3,1)^T & \leq \mathcal{M}(i,j)^T \leq \quad (100,50,150,50)^T\} \end{array}$$

Starting with $\Sigma_A = \emptyset$, the smallest simple section X_1 is inserted directly into this set. Thereafter, we have $X_1 \cap X_2 \neq \emptyset$ and $X_2 \not\subseteq X_1$. Substituting (`I+J,J`) for (i,j) in the inequalities defining the intersection $X_1 \cap X_2$ (in which some boundaries have been refined) gives rise to the following table:

				$\lvert X_1^k\rvert$	$\lvert X_2^k\rvert$	$\lvert X_3^k\rvert$	$\mathcal{G}_k$	
(1)	`26` $\leq$	`I+J`	$\leq$ `65`	300	1570	630	930	
(2)	`11` $\leq$	`J`	$\leq$ `50`	500	2000	0	500	
(3)	`37` $\leq$	`I+2J`	$\leq$ `115`	306	1870	324	630	
(4)	`15` $\leq$	`I`	$\leq$ `15`	700	50	1750	2450	$\leftarrow$

For a threshold $t > 2450$, the simple sections are combined into one representative with 2800 elements as illustrated in the first picture of figure 7.9. Otherwise, the execution set of the `I`-loop becomes partitioned into $[1,14]$, $[15,15]$, and $[16,50]$. Fragmentation of X_1 is simply ignored by associating a copy of this simple section with the three duplicates of the first occurrence. The simple sections into which X_2 becomes fragmented are incrementally computed by refining the boundaries after replacement of the last pair with $1 \leq i-j \leq 14$, $15 \leq i-j \leq 15$, and $16 \leq i-j \leq 50$ respectively:

$$\begin{array}{lll} X_{2a} = \{(i,j) \in \mathcal{Z}^2 \mid & (2,1,3,\mathbf{1})^T & \leq \mathcal{M}(i,j)^T \leq \quad (64,50,114,\mathbf{14})^T\} \\ X_{2b} = \{(i,j) \in \mathcal{Z}^2 \mid & (16,1,17,\mathbf{15})^T & \leq \mathcal{M}(i,j)^T \leq \quad (65,50,115,\mathbf{15})^T\} \\ X_{2c} = \{(i,j) \in \mathcal{Z}^2 \mid & (17,1,18,\mathbf{16})^T & \leq \mathcal{M}(i,j)^T \leq \quad (100,50,150,\mathbf{50})^T\} \end{array}$$

Subsequently, because X_{2a} and X_{2c} do not overlap with any representative simple section, these simple sections are inserted into Σ_A. However, X_{2b} and the representative X_1 still overlap. Again, we have the choice between combining these simple sections, yielding a simple section of size 60, or slicing the iteration space. Below, we present the sizes of the simple sections that result if we partition the iteration space according to one of the pairs obtained by substituting (`15+J,J`) for (i,j) in the inequalities defining $X_1 \cap X_{2b}$.

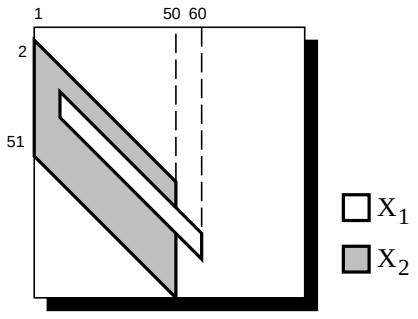


Figure 7.8: Simple Sections

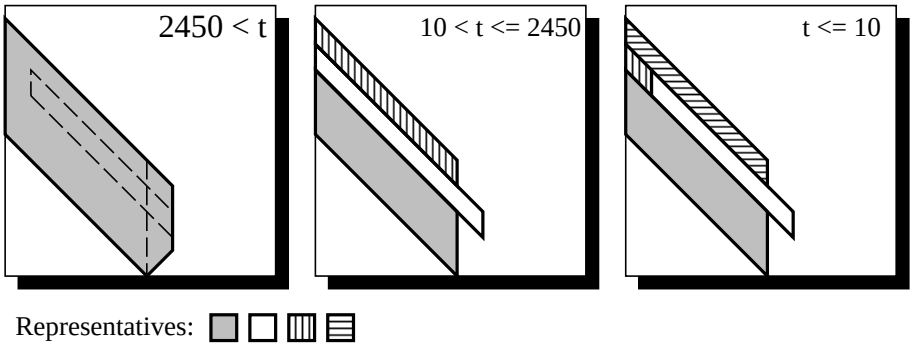


Figure 7.9: Resulting Representatives

				$\lvert X_1^k \rvert$	$\lvert X_2^k \rvert$	$\lvert X_3^k \rvert$	$\mathcal{G}_k$	
1	11 $\leq$	J	$\leq$ 50	10	40	0	10	$\leftarrow$
2	11 $\leq$	J	$\leq$ 50	10	40	0	10	
3	22 $\leq$	2J	$\leq$ 100	10	40	0	10	
4	0 $\leq$	0	$\leq$ 0	0	50	0	0	

For $t > 10$ we combine the simple sections. This gives rise to the representatives shown in the second picture of figure 7.9. For a smaller threshold, the first pair induces the following loop transformation:

```
DO J = 1, 50
 C(15,J) = A2b(J+15,J)
ENDDO
```

$11 \leq J \leq 50$ $\rightarrow$

```
DO J = 1, 10
 C(15,J) = A2b'(J+15,J)
ENDDO
DO J = 11, 50
 C(15,J) = A2b''(J+15,J)
ENDDO
```

Again, the simple sections into which X_{2b} becomes fragmented can be constructed incrementally by refining boundaries after the first pair has been replaced by $16 \leq i \leq 25$, $26 \leq i \leq 65$, and $66 \leq i \leq 65$, which gives rise to two non-empty simple sections. The simple section X_{2b_1} may be directly added to Σ_A, whereas $X_{2b_2} \subseteq X_1$ holds. Hence, the representatives shown in the last picture of figure 7.9 result. The figure clearly illustrates the usefulness of a threshold to control the resulting amount of fragmentation.

7.3 Data Structure Selection

The selection of a sparse storage scheme for each implicitly sparse matrix A is based on the access summary bag $\mathcal{X}_A$ arising after reshaping and iteration space partitioning, the constructed set Σ_A of representatives, and the original property summary set $\mathcal{P}_A$.

7.3.1 Storage Summary Set

For each representative in the set Σ_A, a property and a direction are selected by the sparse compiler, which gives rise to a storage summary set.

Storage Summaries

Given an arbitrary representative $S \in \Sigma_A$, the property $\text{prop}_A(S)$ of this simple section is defined as shown below:

$$\text{prop}_A(S) = \begin{cases} \textbf{zero} & \textbf{if } \exists \langle P, \vec{p}, \textbf{zero} \rangle \in \mathcal{P}_A : S \subseteq P \\ \textbf{dense} & \textbf{if } \exists \langle P, \vec{p}, \textbf{dense} \rangle \in \mathcal{P}_A : S \subseteq P \\ \textbf{sparse} & \textbf{otherwise} \end{cases}$$

The direction $\text{dir}_A(S) \in \mathcal{Z}^2$ of each representative $S \in \Sigma_A$ is defined as follows, where $\delta(S)$ has been defined in section 6.3.1:

$$\text{dir}_A(S) = \begin{cases} \vec{p} & \textbf{if } \exists \langle P, \vec{p}, p \rangle \in \mathcal{P}_A : S \subseteq P \\ \vec{\delta}(S) & \textbf{otherwise} \end{cases}$$

Given these definitions, each representative simple section $S \in \Sigma_A$ defines the following triple $\overline{s}$, called a **storage summary**:

$$\overline{s} = \langle S, \text{dir}_A(S), \text{prop}_A(S) \rangle$$

The set of storage summaries defined by the simple sections in the set Σ_A is referred to as the **storage summary set** $\mathcal{S}_A$.

Storage Patterns

Each storage summary $\langle S, (s_1, s_2)^T, p \rangle \in \mathcal{S}_A$ gives rise to a number of **storage patterns**, where each storage pattern $\mathcal{SP}_k$ is defined for a particular $k \in \mathcal{Z}$ as follows:

$$\mathcal{SP}_k = \{(i,j) \in S \mid s_2 \cdot i - s_1 \cdot j = k\}$$

The **summary constants** of a storage summary $\overline{s} = \langle S, \vec{s}, p \rangle \in \mathcal{S}_A$ with $\vec{s} \neq \vec{0}$ are defined as the maximum value $\mathcal{L}(\overline{s}) \in \mathcal{Z}$ and the minimum value $\mathcal{U}(\overline{s}) \in \mathcal{Z}$ for which the following constraint is still satisfied:

$$\mathcal{SP}_k \neq \emptyset \quad \Rightarrow \quad \mathcal{L}(\overline{s}) \leq k \leq \mathcal{U}(\overline{s})$$

As explained for access summaries, the summary constants of storage summary are either directly defined by one of the boundary pairs for regular storage patterns, or obtained by one step of Fourier-Motzkin elimination to compute the extremal integer values of the expression $s_2 \cdot i - s_1 \cdot j$ for $(i,j) \in S$. Moreover, the same terminology will be used for storage patterns. For instance, if $\vec{s} = (0,1)^T$, then the storage patterns are called row-wise.

7.3.2 Declaration of the Selected Storage Scheme

The storage summary set $\mathcal{S}_A$ represents the storage scheme that has been selected for the implicitly sparse matrix A. Storage will be done according to the storage patterns arising from the storage summaries in this set.

Zero Regions

A storage summary $\langle S, \vec{0}, \textbf{zero} \rangle \in \mathcal{S}_A$ reflects the fact that $a_{ij} = 0$ holds for all $(i,j) \in S$. Hence, no explicit storage of the region represented by $S \subseteq \mathcal{Z}^2$ is required. Moreover, each occurrence of the corresponding enveloping data structure A in the program having an access summary $\langle X, \vec{x}^{\,n} \rangle \in \mathcal{X}_A$ with $X \subseteq S$ is replaced by a zero constant. Thereafter, the condition of the statement in which this replacement occurs is re-computed, which may also affect the conditions of surrounding DO-loops or IF-statements.

Example: In the following fragment, an annotation is used to inform the compiler that the main diagonal of an implicitly sparse matrix A with enveloping data structure A is zero, and will remain so at run-time:

```
      REAL A(N,N), X(N)
C_SPARSE(A : _ZERO(0 <= I-J <= 0))
      ...
S1:   DG = 0.0                          →     ...
      DO I = 1, N                             DG = 0.0
S2:     DG = DG + A(I,I) * X(I)               ...
      ENDDO
      ...
```

Under the assumption that the simple section representing the zero region results as a representative, occurrence `A(I,I)` is replaced by '`0.0`'. Thereafter, the condition '**false**' becomes associated with S_2 and, hence, with the `I`-loop (since we always ignore the 'side-effect' that sets the final value of a loop index). This implies that both statements can be eliminated.

Dense Regions

A storage summary $\overline{s} = \langle S, \vec{s}, \mathbf{dense} \rangle \in \mathcal{S}_A$ indicates that dense storage must be used for the region in A that is represented by the simple section $S \subseteq \mathcal{Z}^2$. Let $\vec{\sigma} \in \mathcal{Z}^4$ and $\vec{\tau} \in \mathcal{Z}^4$ denote the boundary values of this simple section. First, an appropriate indexing method is selected:

$$\theta(\overline{s}) = \begin{cases} \text{row-indexed} & \textbf{if} \quad \vec{s} = (-1,0)^T \\ & \vee \quad \vec{s} \neq (0,+1)^T \ \wedge \ (\tau_1 - \sigma_1) < (\tau_2 - \sigma_2) \\ \text{column-indexed} & \textbf{otherwise} \end{cases}$$

Dense storage of row- and column-wise storage patterns will be column- and row-indexed respectively. For diagonal-wise storage patterns, the index inducing the smallest range will be used. A unique label *lab*$(\overline{s})$, obtained by successively incrementing an integer-valued variable, is associated with each storage summary belonging to a dense region. Thereafter, the following declaration is generated, where k = *lab*$(\overline{s})$ and TYPE denotes the basis type of the enveloping data structure A:

```
TYPE DNk_A(R, L(s̄) : U(s̄))
```

In this declaration, the range R is defined as follows:

$$\mathtt{R} = \begin{cases} \sigma_1 : \tau_1 & \textbf{if } \theta(\overline{s}) = \text{row-indexed} \\ \sigma_2 : \tau_2 & \textbf{otherwise} \end{cases}$$

As is further discussed in chapter 8, appropriate initialization code for this array will be generated at the beginning of the program, where each entry a_{ij} with $(i,j) \in S$ is stored at either the location $(i, s_2 \cdot i - s_1 \cdot j)$ or $(j, s_2 \cdot i - s_1 \cdot j)$ for row- or column-indexed storage respectively, where $\vec{s} = (s_1, s_2)^T$. Moreover, the following replacement is performed for each occurrence of the enveloping data structure in the program with an access summary $\langle X, \vec{x}^{\,n} \rangle \in \mathcal{X}_A$ that satisfies $X \subseteq S$:

```
A(E1,E2)  →  DNk_A(E, s2 * E1 - s1 * E2)
```

In this replacement, k = *lab*$(\overline{s})$ and subscript E is defined as follows:

$$\mathtt{E} = \begin{cases} \mathtt{E1} & \textbf{if } \theta(\overline{s}) = \text{row-indexed} \\ \mathtt{E2} & \textbf{otherwise} \end{cases}$$

After this replacement, the condition of the statement in which this expression occurs is recomputed. Obviously, if $\vec{x}^{\,n} = \vec{s}$, then the second subscript will remain constant in successive iterations of the innermost DO-loop, which tends to enhance data locality for the FORTRAN column-major storage of arrays. In fact, if $\mathcal{L}(\overline{s}) = \mathcal{U}(\overline{s})$ holds, then this second subscript even becomes redundant and a one-dimensional array is used as dense storage.

Example: Below, an annotation is used to indicate that a small band in a 5×5 implicitly sparse matrix B is dense. If we obtain a $S \in \Sigma_B$ representing this band, then the corresponding storage summary $\overline{s} = \langle S, (1,1)^T, \mathbf{dense} \rangle \in \mathcal{S}_B$ gives rise to the following conversion, since $\theta(\overline{s})$ =column-indexed:

```
      REAL B(5,5)
C_SPARSE(B:_DENSE(-1<=I-J<=1))
      DO I = -1, 1
        DO J = MAX(1,1-I), MIN(5,5-I)
          B(I+J,J) = ...
        ENDDO
      ENDDO
```

→

```
REAL DN1_B(1:5,-1:1)
...
DO I = -1, 1
  DO J = MAX(1,1-I), MIN(5,5-I)
    DN1_B(J,I) = ...
  ENDDO
ENDDO
```

The band scheme illustrated below has been selected, and the code has been altered accordingly:

$$B = \begin{pmatrix} b_{11} & b_{12} & 0 & 0 & 0 \\ b_{21} & b_{22} & b_{23} & 0 & 0 \\ 0 & b_{32} & b_{33} & b_{34} & 0 \\ 0 & 0 & b_{43} & b_{44} & b_{45} \\ 0 & 0 & 0 & b_{54} & b_{55} \end{pmatrix}$$

DN1_B	-1	0	$+1$
1	$-$	b_{11}	b_{21}
2	b_{12}	b_{22}	b_{32}
3	b_{23}	b_{33}	b_{42}
4	b_{34}	b_{44}	b_{54}
5	b_{45}	b_{55}	$-$

Sparse Regions

All entries of an implicitly sparse matrix A residing in one of the sparse regions of this matrix are stored in dynamic storage as a pool of sparse vectors. The number of sparse vectors in this pool and the total number of *elements* in the sparse regions are computed as follows, where we define $\mathcal{S}_A^s = \{\langle S, \vec{s}, p \rangle \in \mathcal{S}_A \mid p = \mathbf{sparse}\}$:

$$V_A = \sum_{\overline{s} \in \mathcal{S}_A^s} \mathcal{U}(\overline{s}) - \mathcal{L}(\overline{s}) + 1 \qquad N_A = \sum_{\langle S, \vec{s}, p \rangle \in \mathcal{S}_A^s} |S|$$

Given these numbers, the following declarations are generated, where ρ_A denotes the approximated density of the implicitly sparse matrix A and TYPE denotes the basis type of the enveloping data structure A:

```
INTEGER   NP_A, SZ_A
PARAMETER (NP_A = V_A, SZ_A = INT(ρ_A · N_A) + 2 * NP_A)

TYPE      VAL_A(SZ_A)
INTEGER   IND_A(SZ_A), LOW_A(NP_A), HGH_A(NP_A), LST_A
```

In this manner, a pool of V_A sparse vectors is obtained for matrix A, where the eth element in the arrays LOW_A and HGH_A are used to locate the entries and indices of the eth sparse vector in the parallel arrays VAL_A and IND_A. Because at most $\rho_A \cdot N_A$ entries must be stored, the size of these parallel arrays is set accordingly, with some additional working space as suggested in [77, 80, 235].

A unique base-location is used as label *base*$(\overline{s})$ of each storage summary $\overline{s} \in \mathcal{S}_A^s$ using the following algorithm:

```
bs := 1;
forall s̄ ∈ S_A^s do
   base(s̄) := bs;
   bs     += ( U(s̄) − L(s̄) + 1 );
  endif
enddo
```

Appropriate initialization code is generated at the beginning of the program, where each entry a_{ij} with $(i,j) \in S$ for some $\overline{s} = \langle S, (s_1, s_2)^T, \mathbf{sparse} \rangle \in \mathcal{S}_A$ is stored in the eth sparse vector of this pool, where e is defined as follows

$$e = \mathit{base}(\overline{s}) - \mathcal{L}(\overline{s}) + s_2 \cdot i - s_1 \cdot j \tag{7.7}$$

If $\theta(\overline{s})$ indicates that row-indexed storage is appropriate, where θ is defined as in the previous section, then the row index i is associated with this entry. Otherwise the column index j is associated with this entry. Hence, different index information may be associated with the entries in different sparse regions. In contrast with occurrences that induce accesses to zero or dense regions, however, code generation for the remaining occurrences is less straightforward (see chapter 8).

Example: Consider the following annotations in which the enveloping data structure `C`, the density $\rho_C = 0.125$, and some nonzero structure properties of an 8×8 implicitly sparse matrix C are declared:

```
      REAL C(8,8)
C_SPARSE(C : _DENSITY(0.125))
C_SPARSE(C : _SPARSE( 1 <= I - J <= 4, 2 <= I <= 5)(1,1))
C_SPARSE(C : _SPARSE( 1 <= I - J <= 7, 6 <= I <= 8)(0,1))
C_SPARSE(C : _SPARSE(-7 <= I - J <= 0)             (1,0))
```

Under the assumption that we actually obtain the set $\Sigma_C = \{S_1, S_2, S_3\}$, where the three simple sections are equal to the simple sections arising in the previous annotations, the storage summary set $\mathcal{S}_C = \{\overline{s}_1, \overline{s}_2, \overline{s}_3\}$ results for the following storage summaries:

					$\mathcal{L}(\overline{s}_i)$	$\mathcal{U}(\overline{s}_i)$	$\mathit{base}(\overline{s}_i)$
$\overline{s}_1$	$=$	$\langle S_1,$	$(1,1)^T,$	$\mathbf{sparse}\rangle$	1	4	1
$\overline{s}_2$	$=$	$\langle S_2,$	$(0,1)^T,$	$\mathbf{sparse}\rangle$	6	8	5
$\overline{s}_3$	$=$	$\langle S_3,$	$(-1,0)^T,$	$\mathbf{sparse}\rangle$	1	8	8

Hence, we obtain $V_C = 15$ and $N_C = 64$, which gives rise to the following declarations of the pool of sparse vectors:

```
REAL    VAL_C(38)
INTEGER IND_C(38), LOW_C(15), HGH_C(15), LST_C
```

Row index information is stored for the last eight vectors in this pool, based on column-wise storage patterns. Column index information is stored for the remaining vectors that are based on diagonal- and row-wise storage patterns.

If $(i,j) \in S$ holds for a storage summary $\langle S, \vec{s}, \mathbf{sparse} \rangle \in \mathcal{S}_C$ and an entry c_{ij}, then this entry is stored in the eth sparse vector of the pool, where e is defined as in (7.7). Hence, as illustrated in figure 7.10, for this data structure the value of e is determined as follows:

$$e = \begin{cases} i - j & \textbf{if } (i,j) \in S_1 \\ i - 1 & \textbf{if } (i,j) \in S_2 \\ 7 + j & \textbf{if } (i,j) \in S_3 \end{cases}$$

For instance, c_{41}, c_{55}, c_{81}, and c_{18} belong to the 3rd, 12th, 7th, and 15th sparse vector of the pool that is used to dynamically store the entries of the implicitly sparse matrix C.

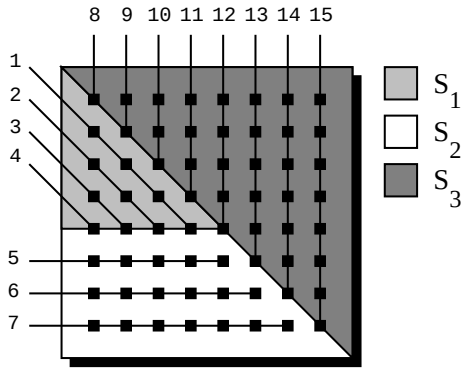


Figure 7.10: Pool of Sparse Vectors

COMMON-Storage

All variables implementing the sparse storage scheme that is selected for an implicitly sparse matrix A with enveloping data structure A are made available to all appropriate subroutines and functions using COMMON-storage. These variables are placed in a single named COMMON-block, as is illustrated below:

```
COMMON /A/ DN1_A, DN2_A, ..., VAL_A, IND_A, LOW_A, HGH_A, LST_A
```

This COMMON-statement together with the appropriate declarations of the variables are generated in the main program and in all clones in which A is uniquely associated with a formal argument (the occurrences of which are handled as occurrences of A).

Chapter 8

Phase 3: Sparse Code Generation

In the third and final phase, the actual data structure transformations are applied by converting the (possibly adapted) dense code into a form that operates on the selected sparse storage schemes, thereby using overhead reducing techniques as much as possible. A small number of general primitives are supplied in a separate library, so that constructs that would otherwise appear frequently in the generated sparse code can be replaced by calls to the appropriate primitives. This approach tends to reduce the size of the generated sparse code whereas, on the other hand, the primitives are chosen general enough to leave the sparse compiler with enough flexibility in selecting data structures. Identifiers of all subroutines and functions in this library, as well as all compiler generated identifiers contain underscores to prevent conflicts with identifiers in the original dense program.[1]

In this chapter, we first present the primitives that are supplied in the library. Thereafter, the generation of sparse code is presented in detail. Because the programmer remains unaware of the actual sparse storage scheme that is selected for each implicitly sparse matrix, the sparse compiler is also responsible for generating appropriate initialization code, which is the final topic of this chapter.

8.1 The Library

The sparse compiler performs a source-to-source translation of a dense program into semantically equivalent sparse code. The resulting sparse program together with a library containing some useful primitives that may be used in this program are supplied to a conventional FORTRAN compiler for the desired target architecture, as depicted below:

To leave the sparse compiler with sufficient flexibility in the data structure selection, only a small number of general primitives are supplied in this library. Supplying *some* primitives in a library, however, tends to reduce the size of the generated sparse program because constructs that otherwise would appear frequently can be replaced by calls to the appropriate primitives. In addition, several versions of the library that are fully hand-optimized for different target architectures can be constructed in advance, which improves the efficiency of all sparse codes that are generated for one of these architectures.

[1] Here, we assume that identifiers in the original dense program adhere the ANSI FORTRAN 77-standard, whereas the compiler used to compile the generated sparse code supports the use of underscores in identifiers.

Even without fully hand-optimized versions, this separate compilation approach is useful to reduce compilation-time, because after the library has been compiled for a specific target architecture, the resulting object file may be linked with every sparse program that is generated for this architecture.

In the context of sparse computations, the use of a library has been advocated by others. In the sparse extensions to BLAS [68, 69], a number of basic sparse operations are identified and standardized in a library to improve the readability, portability, and efficiency of sparse codes. This sparse BLAS, developed for an environment in which the symbolic and numerical operations are completely separated, focuses on the actual *operations* on sparse vectors (e.g. sparse dot product). The library of the sparse compiler, on the other hand, mainly deals with basic *manipulation* of sparse vectors (e.g. insertion of an entry). The sparse compiler implements the actual operations on sparse vectors by generating the appropriate constructs, although some of these constructs could be replaced by calls to sparse BLAS routines in a future implementation. By focusing on a small number of general primitives for manipulating sparse vectors, we leave the compiler with sufficient flexibility to use a pool of sparse vectors as dynamic storage for the sparse regions in each implicitly sparse matrix, where the layout of vectors may be different for each region. Moreover, we also prevent the situation in which each primitive must be implemented for a vast amount of existing storage schemes for sparse matrices. In SPARSKIT [185] this latter situation is partly avoided by limiting the number of sparse storage schemes that are used internally and providing a set of conversion routines from and to a single storage scheme.

8.1.1 Ceiling and Floor Functions

After applying the reshaping method, ceiling and floor functions may arise in the loop bounds of the target loop. Because these functions are not available as intrinsic functions, an implementation of these functions is supplied in the library. Each expression containing a ceiling or floor function is implemented using one of the following function calls:

```
⌈ d / n ⌉  ≡  CEIL__(d, n)      ⌊ d / n ⌋  ≡  FLOOR__(d, n)
```

Straightforward implementations of these functions are shown below:

```
INTEGER FUNCTION CEIL__(D, N)
INTEGER D, N

CEIL__ = D / N
IF (MOD(D,N).NE.0) THEN
  IF ((D.GT.0).EQV.(N.GT.0)) THEN
    CEIL__ = CEIL__ + 1
  ENDIF
ENDIF
RETURN
END
```

```
INTEGER FUNCTION FLOOR__(D, N)
INTEGER D, N

FLOOR__ = D / N
IF (MOD(D,N).NE.0) THEN
  IF ((D.LT.0).NEQV.(N.LT.0)) THEN
    FLOOR__ = FLOOR__ - 1
  ENDIF
ENDIF
RETURN
END
```

Together with the FORTRAN intrinsic integer functions `MAX0` and `MIN0` operating on an arbitrary number of arguments, the functions `CEIL_` and `FLOOR_` can be used in the generated code. For notational convenience, however, the mathematical notation for floor and ceiling functions is used in most examples.

Example: Consider the following representation of the innermost loop bounds in a double loop with loop index $\vec{\mathtt{I}} = (\mathtt{I}, \mathtt{J})^T$:

$$\begin{pmatrix} 1 & 2 \\ -1 & 3 \\ 1 & -4 \\ 1 & -3 \end{pmatrix} \vec{\mathtt{I}} \leq \begin{pmatrix} 10 \\ 5 \\ 0 \\ 1 \end{pmatrix}$$

This system gives rise to the following innermost DO-loop:

```
DO J = MAX0(CEIL__(I,4), CEIL__(I-1,3)),  MIN0(FLOOR__(10-I,2), FLOOR__(5+I,3))
  ...
ENDDO
```

8.1.2 Sparse Primitives

As shown in table 8.1, the library also supplies a number of primitives that can be used to manipulate sparse vectors in a pool. Inspired on the BLAS convention (see e.g [69, 70]), the first underscore in each identifier may be replaced by any type specification characters in $\{\mathtt{I}, \mathtt{S}, \mathtt{D}, \mathtt{C}\}$ to define INTEGER, REAL, DOUBLE PRECISION, or COMPLEX as basis type of the entries. In contrast with the scatter and gather primitives defined in sparse BLAS [69], primitives _SCT__ and _GTH__ also support the so-called switch technique [169, ch1].

Initialization

The entries in the sparse regions of an implicitly sparse $m \times n$ matrix A are stored in a pool consisting of V_A sparse vectors. Since the layout of these sparse vectors may differ over the regions, row index information is required for some sparse vectors, whereas column index information is required for others. Hence, *although this kind of storage organization differs from conventional general sparse row- or column-wise storage schemes, in essence this pool can be thought of as implementing general sparse row-wise storage of a* $V_A \times \max(m, n)$ sparse matrix. This implies that initialization methods for general sparse row-wise storage schemes [78, p30-31][164, p15-22][235, p31-34] are applicable.

The primitive _INI__ provides an initialization method for the pool of sparse vectors that assumes that the entries in the sparse regions of A are available at locations $2, \ldots,$LST_A of three parallel arrays VAL_A, IND_A, and TMP__. The first two arrays contain the numerical value and appropriate index information of each entry. These arrays actually form a part of the sparse storage scheme that has been selected for the implicitly sparse matrix A. Array TMP__ is only used temporarily to hold the number of the sparse vector to which each entry belongs and can be used for other purposes after initialization.

The pool of sparse vectors selected as dynamic storage scheme of an implicitly matrix A with enveloping data structure A is initialized using the following subroutine call:

```
CALL _INI__(VAL_A, TMP__, IND_A, LOW_A, HGH_A, NP_A, SZ_A, LST_A)
```

The implementation of subroutine SINI__ is presented below. The first location is always used as location $\perp$, which has the property that $\mathtt{VAL}(\perp) = 0$. Furthermore, array LOW is temporarily used to count the number of entries in each Ith sparse vector:

Primitive	Short Description
_INI__()	Initialization of dynamic storage
_INS__()	Insertion in a sparse vector
_SCT__()	Expansion of a sparse vector into a dense format
_GTH__()	Compression of a dense format into a sparse vector
LKP__()	Lookup in a sparse vector (independent of basis type)

Table 8.1: Sparse Primitives

```
SUBROUTINE SINI__(VAL, TMP, IND, LOW, HGH, NP, SZ, LST)
INTEGER NP, SZ, LST, I, J, K
INTEGER TMP(SZ), IND(SZ), LOW(NP), HGH(NP)
REAL    VAL(SZ), V

VAL(1) = 0.0
DO I = 1, NP
  LOW(I) = 0
ENDDO
DO K = 2, LST
  I      = TMP(K)
  LOW(I) = LOW(I) + 1
ENDDO
...
```

Thereafter, we assign the appropriate values to the elements of HGH, whereas each element of LOW is set to the first location *after* the locations reserved for each individual sparse vector:

```
LOW(1) = LOW(1) + 2
HGH(1) = LOW(1) - 1
DO I = 2, NP
  LOW(I) = LOW(I) + LOW(I-1)
  HGH(I) = LOW(I) - 1
ENDDO
...
```

Finally, entries and index information are moved to the appropriate locations using the following code which, although a WHILE-loop is nested within the DO-loop, has a running time that is proportional to the number of entries in the sparse regions of the corresponding implicitly sparse matrix:

```
DO K = 2, LST
  IF (IND(K).GT.0) THEN
    I      = TMP(K)
    LOW(I) = LOW(I) - 1
    J      = LOW(I)

    DO WHILE (J.NE.K)
      V = VAL(J)
      I = IND(J)
      VAL(J) =   VAL(K)
      IND(J) = - IND(K)
      VAL(K) = V
      IND(K) = I

      I      = TMP(J)
      LOW(I) = LOW(I) - 1
      J      = LOW(I)
    ENDDO
  ELSE
   IND(K) = - IND(K)
  ENDIF
ENDDO
...
```

Recall that the individual elements of array IND can be used to store row indices for some sparse vectors and column indices for other sparse vectors. The index information in the remaining part of array IND is reset to indicate that these locations are free.

```
DO K = LST+1, SZ
  IND(K) = 0
ENDDO
RETURN
END
```

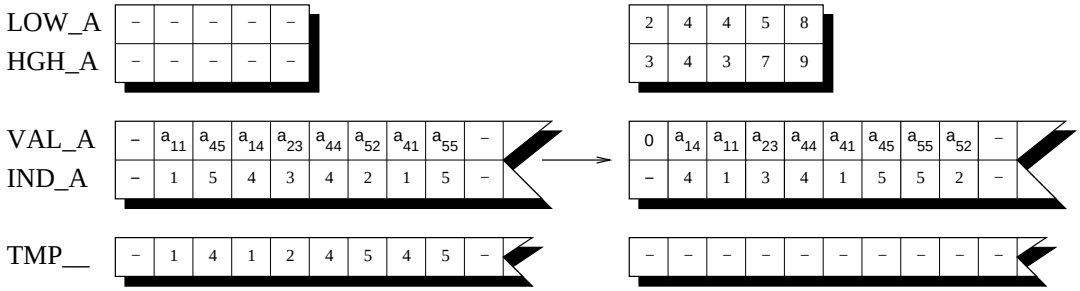


Figure 8.1: Initialization

If the Eth sparse vector is empty, then HGH_A(E)=LOW_A(E)-1 holds after initialization. In figure 8.1, for example, we show possible contents of the arrays before and after initialization in case the pool is used to implement general sparse row-wise storage for the following sparse matrix:

$$\begin{pmatrix} a_{11} & & & a_{14} & \\ & & a_{23} & & \\ & & & & \\ a_{41} & & & a_{44} & a_{45} \\ & a_{52} & & & a_{55} \end{pmatrix}$$

Although array TMP__ can be re-used for initializing the pools belonging to several implicitly sparse matrices, and can also be re-used for e.g. recording permutations applied to some of these matrices thereafter, the use of this array increases the storage requirements of the application. Therefore, alternative initialization methods, usually trading storage requirements for computational time, should be incorporated in a future implementation.

Lookup

A search for a particular element with index information F in the Eth sparse vector in a pool used to store the entries in the sparse regions of an implicitly sparse matrix with enveloping data structure A is performed using the following CALL-statement:

```
CALL LKP__(IND_A, LOW_A(E), HGH_A(E), F)
```

The entries in this sparse vector are scanned until either the index information matches F, or the entries in this vector have been exhausted. In the latter case, location $\perp$ is returned to indicate that the searched element is zero:

```
      INTEGER FUNCTION LKP__(IND, LOW, HGH, F)
      INTEGER IND(*), LOW, HGH, F

      DO LKP__ = LOW, HGH
        IF (IND(LKP__).EQ.F) GOTO 10
      ENDDO
      LKP__ = 1
10    RETURN
      END
```

Insertion

Insertion of a new entry with associated index information F in the Eth sparse vector of a pool that has been selected for an implicitly sparse matrix with enveloping data structure A is performed using the following call:

```
CALL _INS__(VAL_A, IND_A, LOW_A, HGH_A, E, NP_A, SZ_A, LST_A, AD, F)
```

The numerical value of the new entry is set to zero by this call. However, the location of this entry is returned in AD, so that the appropriate value can be assigned to this entry afterwards (viz. VAL_A(AD)=...). Below, we present an implementation of the subroutine SINS_ assuming that no entry with index F is present yet in the Eth sparse vector [164, p25-33][235, p16-21]

If there is some directly surrounding free space for the Eth sparse vector, this space is used to store the new entry:

```
      SUBROUTINE SINS__(VAL, IND, LOW, HGH, E, NP, SZ, LST, AD, F)
      INTEGER E, NP, SZ, LST, AD, F, I, J, L
      INTEGER LOW(NP), HGH(NP), IND(SZ)
      REAL    VAL(SZ)

      I = 0
5     IF ( (HGH(E).LT.SZ) .AND. (IND(HGH(E)+1).EQ.0) ) THEN
        HGH(E)  = HGH(E) + 1
        AD      = HGH(E)
        VAL(AD) = 0.0
        IND(AD) = F

        IF (LST.LT.AD) LST = AD

      ELSEIF ((LOW(E).GT.2).AND.(IND(LOW(E)-1).EQ.0)) THEN
        LOW(E)  = LOW(E) - 1
        AD      = LOW(E)
        VAL(AD) = 0.0
        IND(AD) = F
      ...
```

If surrounding free space is not available, then an attempt is made to move the whole sparse vector together with the inserted entry to the end of the data structure, where presumably most free space resides. The old locations of the sparse vector are marked as free:

```
      ELSEIF ((SZ-LST).GT.(HGH(E)-LOW(E)+1)) THEN
                                  ! Data Movement (begin)
        AD = LST + 1

        DO I = LOW(E), HGH(E)
           VAL(AD) = VAL(I)
           IND(AD) = IND(I)
           IND(I)  = 0
           AD      = AD + 1
        ENDDO
        VAL(AD) = 0.0
        IND(AD) = F

        LOW(E)  = LST + 1
        HGH(E)  = AD
        LST     = AD
      ...                         ! Data Movement (end)
```

In figure 8.2, we illustrate the data movement that occurs if an entry a_{31} is inserted in the 3th sparse vector of a pool that implements general row-wise storage of a sparse matrix A.

If this data movement cannot be done, a so-called left-compression is performed. Since such a left-compression is relatively expensive, sufficient working space (or 'elbow room') must be supplied to prevent the situation in which a left compression has to be applied many times during program execution. First, the variable I, used as a flag in this branch, is tested to prevent a second application of left compression during the same insertion:

```
      ELSE              ! Left Compression (begin)
        IF (I.EQ.1) THEN
          PRINT *, 'Out of Memory'
          STOP
        ENDIF
        ...
```

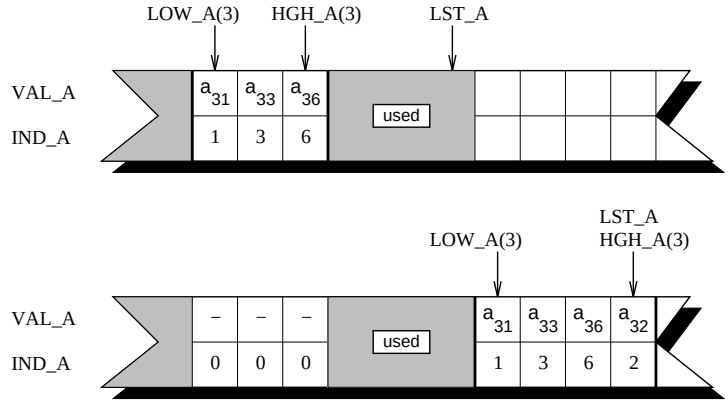


Figure 8.2: Data Movement

The compression starts by marking the index of the first entry of every `I`th sparse vector in the pool that is non-empty with the negative value `-I`. The value of the index is saved in the corresponding `LOW`-pointer:

```
DO I = 1, NP
  IF (HGH(I).GE.LOW(I)) THEN
    J             = IND( LOW(I) )
    IND( LOW(I) ) = - I
    LOW(I)        = J
  ENDIF
ENDDO
```

Thereafter, the actual left compression is performed:

```
J = 2
DO I = 2, LST
  IF (IND(I).GT.0) THEN
    VAL(J) = VAL(I)
    IND(J) = IND(I)
    J      = J + 1
  ELSEIF (IND(I).LT.0) THEN
    L      = - IND(I)
    VAL(J) = VAL(I)
    IND(J) = LOW(L)
    LOW(L) = J
    HGH(L) = J + HGH(L) - I
    J      = J + 1
  ENDIF
ENDDO
```

A very important property of this implementation of left compression is *that the relative order of entries in each sparse vector is preserved.* This property is exploited to account for data movement in the implementation of guard encapsulation (see section 8.2.3). Finally, the remaining locations are marked as free and the last used location is recorded. Thereafter, the insertion is tried again:

```
  DO I = J, LST
    IND(I) = 0
  ENDDO
  LST = J - 1
  I   = 1
  GOTO 5
ENDIF              ! Left Compression (end)
RETURN
END
```

Note that this approach fails if the sparse vector in which the insertion is done cannot be moved to the free space after the left compression, even if some free space remains. This is an extra argument for providing sufficient working space.

Expansion and Compression

The Eth sparse vector in a pool belonging to an implicitly sparse matrix with enveloping data structure A can be expanded into a dense one-dimensional array P and a switch array SWT with the following call:

```
CALL _SCT__(VAL_A, IND_A, LOW_A(E), HGH_A(E), P, SWT)
```

Likewise, this expanded vector is compressed into the pool again using the following call:

```
CALL _GTH__(VAL_A, IND_A, LOW_A(E), HGH_A(E), P, SWT)
```

In figure 8.3, the expansion and compression of the 3rd sparse vector in a pool that uses rowwise storage of an 8×8 sparse matrix A is illustrated. Implementations of both subroutines are shown below [78, ch2][169, ch1]:

```
SUBROUTINE SSCT__(VAL,IND,LOW,HGH,P,SWT)
INTEGER IND(*), LOW, HGH, I
REAL    VAL(*), P(*)
LOGICAL SWT(*)

DO I = LOW, HGH
    P( IND(I) ) = VAL(I)
  SWT( IND(I) ) = .TRUE.
ENDDO

RETURN
END
```

```
SUBROUTINE SGTH__(VAL,IND,LOW,HGH,P,SWT)
INTEGER IND(*), LOW, HGH, I
REAL    VAL(*), P(*)
LOGICAL SWT(*)

DO I = LOW, HGH
  VAL(I)       = P( IND(I) )
    P( IND(I) ) = 0.0
  SWT( IND(I) ) = .FALSE.
ENDDO
RETURN
END
```

All used elements in the dense array P and the switch array SWT are reset during the compression to support the subsequent expansions. In this manner, these arrays only have to be initialized once at the beginning of the program (see section 8.3).

8.2 Actual Sparse Code Generation

Due to the way in which the representative simple sections of each implicitly sparse matrix A have been constructed, for each occurrence of the corresponding data structure A with access summary $\langle X, \vec{x}^{\,n} \rangle \in \mathcal{X}_A$, there is a storage summary $\overline{s} = \langle S, \vec{s}, p \rangle \in \mathcal{S}_A$ such that $X \subseteq S$.

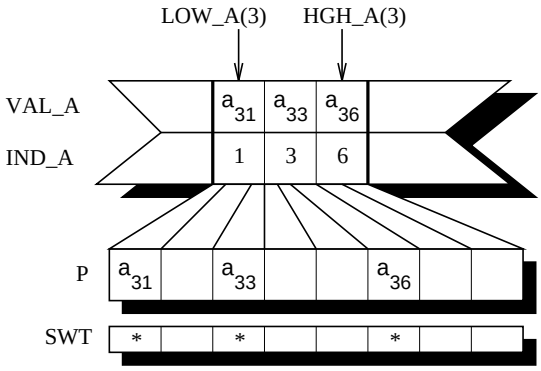


Figure 8.3: Expansion and Compression

In this case, we refer to the storage summary $\overline{s}$ as the storage summary that **matches** the occurrence of A. Since $p \in \{\mathbf{zero}, \mathbf{dense}, \mathbf{sparse}\}$, we can distinguish between occurrences of enveloping data structures accessing a part of the array corresponding to a zero, dense, or sparse region in an implicitly sparse matrix. We refer to these occurrences as zero, dense, and sparse occurrences, respectively.

In this section, we present a method to generate sparse code by applying the appropriate data structure transformations to these occurrences. The method assumes that subscript bounds are not violated in the original dense program, because otherwise subscripts of static dense storage or the pointers used to locate sparse vectors in the pool may also be violated in the generated sparse code. Since for some dense programs, out-of-bounds addressing is actually intended, the semantics of the program could be affected. Therefore, dense programs in which potential subscript violations arise are not converted by the prototype sparse compiler.

8.2.1 Zero and Dense Occurrences

Suppose that the storage summary $\overline{s} = \langle S, (s_1, s_2)^T, p\rangle \in \mathcal{S}_A$ with $p \in \{\mathbf{zero}, \mathbf{dense}\}$ matches an arbitrary occurrence `A(E1,E2)` in the program, where `E1` and `E2` denote the (possibly inadmissible) subscripts. If $p = \mathbf{zero}$, then we replace the occurrence with a zero constant of the appropriate type, depending on the basis type of the enveloping data structure A:

basis type	constant
INTEGER	0
REAL	0.0
DOUBLE PRECISION	0.0D
COMPLEX	CMPLX(0.0,0.0)

If $p = \mathbf{dense}$, then we replace the occurrence with the following expression, where the second subscript is omitted if $\mathcal{L}(\overline{s}) = \mathcal{U}(\overline{s})$:

```
DNk_A(E, s2 * E1 - s1 * E2)
```

In this replacement, $k = lab(\overline{s})$ and subscript E is defined as follows:

$$\mathtt{E} = \begin{cases} \mathtt{E1} & \textbf{if } \theta(\overline{s}) = \text{row-indexed} \\ \mathtt{E2} & \textbf{otherwise} \end{cases}$$

After all such replacements are done in a statement, the conditions of this statement and surrounding IF-statements and DO-loops are re-computed. If the condition of a statement becomes '**false**' (which is only possible if this statement does not call any function with side-effects), then this statement is eliminated from the program (recall that we always assume that the final value of a loop index is not used after a DO-loop).

Example: As an extreme example, if we know that $A = 0$ holds for an implicitly sparse matrix with enveloping data structure A, then the following fragment computing $\vec{b} = A\vec{x}$ is converted as shown below:

```
      REAL A(M,N), B(M), X(N)
C_SPARSE(A : _ZERO() )
      ...
      DO I = 1, M
        B(I) = 0.0
        DO J = 1, N
          B(I) = B(I) + A(I,J) * X(J)
        ENDDO
      ENDDO
```

→

```
      REAL B(M), X(N)
      ...
      DO I = 1, M
        B(I) = 0.0
      ENDDO
```

The storage summary set $\mathcal{S}_A = \{\langle S, \vec{0}, \mathbf{zero}\rangle\}$ results, where S contains the whole index set of A. Consequently, the occurrence of A is replaced by '0.0', which converts the conditions of the resulting assignment statement and surrounding J-loop into '**false**'. By cleaning up the code accordingly, the fragment is automatically converted into code that performs the appropriate operation $\vec{b} = A\vec{x} = \vec{0}$.

Example: Replacement may also alter the condition of a statement in another manner. For example, if besides the implicitly sparse matrix of the previous example, we also have an implicitly sparse matrix B with enveloping data structure B, then the following conversion is done (the sparse occurrence `B(I,J)` will be handled as explained in subsequent sections):

```
DO I = 1, M                                        DO I = 1, M
  DO J = 1, N                                        DO J = 1, N
    X = X + 3 * (A(I,J) + B(I,I))        →             X = X + 3 * (0.0 + B(I,I))
  ENDDO                                              ENDDO
ENDDO                                              ENDDO
```

This affects the condition of the assignment statement as follows, which implies that guard encapsulation may become feasible because the guard '$(\mathtt{I},\mathtt{I}) \in E(B)$' dominates the condition afterwards:

$$(\mathtt{I},\mathtt{J}) \in E(A) \vee (\mathtt{I},\mathtt{I}) \in E(B) \quad \rightarrow \quad (\mathtt{I},\mathtt{I}) \in E(B)$$

Example: A similar, but less trivial, example is shown below. The condition that has been associated with the original IF-statement is '$(\mathtt{I},\mathtt{J}) \in E(A) \vee (\mathtt{I},\mathtt{J}) \in E(B) \vee (2,2) \in E(A)$'. After replacing both occurrences of A by zero, however, we have E.t=**false** for the boolean expression, while the condition of the last assignment statement becomes '**false**'. Hence, the condition of the whole IF-statement changes into '$(\mathtt{I},\mathtt{J}) \in E(B)$':

```
IF (A(I,J).NE.0.0) THEN
  NNZ = NNZ + 1
ELSE
  ACC = ACC + B(I,J)
  A(2,2) = A(2,2) * 3.0
ENDDO
```

→

```
IF (0.0.NE.0.0) THEN   ← (I,J) ∈ E(B)
  NNZ = NNZ + 1        ← true
ELSE
  ACC = ACC + B(I,J)   ← (I,J) ∈ E(B)
ENDDO
```

In fact, the whole construct could be replaced by the ELSE-branch.

8.2.2 Preparatory Pass over Sparse Occurrences

Before any code is generated for the sparse occurrences of the enveloping data structure A of each implicitly sparse matrix A, a preparatory pass over the sparse occurrences with admissible subscripts is made to determine which occurrences may be involved in a guard encapsulation or access pattern expansion. During examination of an occurrence of array A, we refer to this occurrence as the **candidate**.

Suppose that $\overline{s} = \langle S, (s_1, s_2)^T, \mathbf{sparse}\rangle \in \mathcal{S}_A$ matches a candidate with access summary $\langle X, \vec{x}^{\,n}\rangle \in \mathcal{X}_A$ and admissible subscripts $F(\vec{\mathtt{I}}) = \vec{v} + W\vec{\mathtt{I}}$ in a loop with index vector $\vec{\mathtt{I}}$ and iteration space $IS \subseteq \mathcal{Z}^d$. Then, we define the e-tag of the candidate as follows, where *base*$(\overline{s})$ contains the base-location of the access summary in the pool of sparse vectors:

$$e(\vec{\mathtt{I}}) = \mathit{base}(\overline{s}) - \mathcal{L}(\overline{s}) + (+s_2, -s_1) \cdot F(\vec{\mathtt{I}})$$

The f-tag of the candidate is defined as shown below:

$$f(\vec{\mathtt{I}}) = \begin{cases} (1,0) \cdot F(\vec{\mathtt{I}}) & \textbf{if } \theta(\overline{s}) = \text{row-indexed} \\ (0,1) \cdot F(\vec{\mathtt{I}}) & \textbf{otherwise} \end{cases}$$

During each iteration $\vec{\mathtt{I}} = \vec{i}$, an element in the $e(\vec{i})$th sparse vector of the pool with associated index information $f(\vec{i})$ will be accessed. Obviously, we could generate lookup code directly, but this would yield very inefficient sparse code and should only be used as a last resort.

If the *effective* access patterns of the candidate and the storage patterns are consistent, i.e. $\vec{x}^{\,n} = (s_1, s_2)^T$, then either guard encapsulation or access pattern expansion may be feasible, because entries along each $e(\vec{\mathtt{I}})$th sparse vector can be generated efficiently together with the appropriate index information (cf. constraints (a) and (b) at page 84). It can be easily verified that in this case there is a $1 \leq p \leq d$ such that both the e-tag and f-tag can be expressed as follows, where $f_p \neq 0$:

$$\left\{ \begin{array}{lcl} e(\vec{\mathtt{I}}) & = & e_0 + e_1 \cdot \mathtt{I}_1 + \ldots + e_{p-1} \cdot \mathtt{I}_{p-1} \\ f(\vec{\mathtt{I}}) & = & f_0 + f_1 \cdot \mathtt{I}_1 + \ldots + f_{p-1} \cdot \mathtt{I}_{p-1} + f_p \cdot \mathtt{I}_p \end{array} \right. \tag{8.1}$$

Candidates for Guard Encapsulation

If guard '$F(\vec{\mathtt{I}}) \in E(A)$' dominates the loop-body of the $\mathtt{I}_p$-loop (see section 5.3.2), and this DO-loop is a stride-1 DO-loop with admissible loop bounds that has not been involved in a guard encapsulation before, then the candidate and the $\mathtt{I}_p$-loop can be involved in a guard encapsulation if the following two constraints are also satisfied.

To prevent the requirement for *ordered* storage , we require that the iterations of this DO-loop may be executed in arbitrary order (cf. constraint (c) at page 84):

- No data dependence is carried by the $\mathtt{I}_p$-loop and no exit branch [234, p238-241] or STOP-statement can be executed in the loop-body of this DO-loop.

Additionally, to simplify code generation and, again, to prevent the need for ordered storage, we impose the following constraint:

- During each fixed iteration $\mathtt{I}_1 = i_1, \ldots, \mathtt{I}_{p-1} = i_{p-1}$, insertions in the $e(\vec{\mathtt{I}})$th sparse vector of the pool cannot occur.

If the first constraint is violated, the sparse compiler inquires the programmer whether the prohibitive loop-carried data dependences may be ignored (cf. section 4.3.1).

Verifying the second constraint requires more effort. All occurrences of the same enveloping data structure A appearing at the left-hand side of assignment statements in the loop-body of the $\mathtt{I}_p$-loop are examined. If the subscripts of such a left-hand side occurrence are structurally equivalent to the subscripts $F(\vec{\mathtt{I}})$ of the candidate, which means that coefficients of loop indices in the common nesting depth of the two statements in which the occurrences appear are identical whereas all other coefficients are zero, then this left-hand side occurrence cannot induce insertion (viz. the corresponding guard dominates the loop-body). Likewise, if the storage summary that matches the left-hand side occurrence differs from $\overline{s}$, the last constraint is still satisfied, although insertions in *other* sparse vectors of the same pool may occur in this case.

Otherwise, a more expensive test is required. First, we construct the e-tag $e'(\vec{\mathtt{J}})$ of the left-hand side occurrence, where $\vec{\mathtt{J}}$ denotes the index vector of the loop in which this occurrence appears, together with a (conservative) representation $A\vec{\mathtt{J}} \leq \vec{b}$ of the iteration space of this loop, where inadmissible loop bounds are handled by leaving the corresponding loop indices unbounded. Because the first p components of $\vec{\mathtt{I}}$ and $\vec{\mathtt{J}}$ are equal and the e-tag $e(\vec{\mathtt{I}})$ of the candidate only depends on indices $\mathtt{I}_1, \ldots, \mathtt{I}_{p-1}$, the last constraint may be violated if the following equation has an integer solution for $A\vec{\mathtt{J}} \leq \vec{b}$:[2]

[2] This test strongly resembles the test performed during data dependence analysis while testing for the data direction vector $(=, \ldots, =, *, \ldots, *)$, although here the granularity is increased to sparse vectors rather than individual elements.

$$e(\vec{\mathtt{I}}) = e'(\vec{\mathtt{J}})$$

Fourier-Motzkin elimination is used to test the consistency of the system of inequalities arising from the representation of the iteration space and this equation (rewritten into $e(\vec{\mathtt{I}}) \leq e'(\vec{\mathtt{J}})$ and $e'(\vec{\mathtt{J}}) \leq e(\vec{\mathtt{I}})$). If this system is consistent or if the e-tag of the left-hand side occurrence cannot be constructed due to inadmissible subscripts, we conservatively assume that the last constraint is violated and disable guard encapsulation. If Fourier-Motzkin elimination reveals that the system is inconsistent, we may safely conclude that the last condition is not violated, although we still assume that insertions in other sparse vectors of the same pool may occur.

If all constraints are satisfied, then we record that encapsulation of the dominating guard is feasible for the $\mathtt{I}_p$-loop and the candidate (and all occurrences of enveloping data structure A in the loop-body having structurally equivalent subscripts). If insertions in *other* sparse vectors of the same pool may occur, this is also recorded because eventually this must be dealt with in the actual implementation of guard encapsulation.

Example: Consider the following example, in which an annotation is used to enforce the selection of general sparse row-wise storage for an implicitly sparse matrix A:

```
      REAL A(100,100)
C_SPARSE(A: _SPARSE( )(0,1))
      ...
      DO I = 1, 50
        DO J = I+1, 100
          X(J) = X(J) + A(I,J)          ← (I,J) ∈ E(A)
          DO K = 1, 100                 ← (I,J) ∈ E(A)
            A(J,K) = A(J,K) + A(I,J)    ← (I,J) ∈ E(A)
          ENDDO
          A(I,J) = A(I,J) * 2.0         ← (I,J) ∈ E(A)
        ENDDO
      ENDDO
```

The true and, hence effective access patterns of the first occurrence of A are consistent with the storage patterns. When this occurrence becomes the candidate, the following e-tag and f-tag are constructed (viz. $p = 2$), where $\vec{\mathtt{I}} = (\mathtt{I}, \mathtt{J})^T$:[3]

$$\begin{aligned} e(\vec{\mathtt{I}}) &= 1 \cdot \mathtt{I} \\ f(\vec{\mathtt{I}}) &= 0 \cdot \mathtt{I} + 1 \cdot \mathtt{J} \end{aligned}$$

Because guard '$(\mathtt{I}, \mathtt{J}) \in E(A)$' dominates the loop-body of the J-loop and no data dependences are carried by this DO-loop, we test whether insertion in the Ith sparse vector of the pool (i.e. the Ith row in this case) may occur during a fixed iteration $\mathtt{I} = i$. Therefore, the occurrences A(J,K) and A(I,J) appearing at the left-hand side of assignment statements in the J-loop are examined. Because the subscripts of the latter are structurally equivalent to the subscripts of the candidate, only the former is further examined. The following system of linear inequalities is constructed, where $\vec{\mathtt{J}} = (\mathtt{I}, \mathtt{J}, \mathtt{K})^T$ and $e'(\vec{\mathtt{J}}) = \mathtt{J}$:

$$\left.\begin{array}{c} \left(\begin{array}{ccc|c} 0 & 0 & 1 & 100 \\ 0 & 0 & -1 & -1 \\ 0 & 1 & 0 & 100 \\ 1 & -1 & 0 & -1 \\ 1 & 0 & 0 & 50 \\ -1 & 0 & 0 & -1 \end{array}\right) \end{array}\right\} A\vec{\mathtt{J}} \leq \vec{b}$$

$$\left.\begin{array}{c} \left(\begin{array}{ccc|c} -1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \end{array}\right) \end{array}\right\} e(\vec{\mathtt{I}}) = e'(\vec{\mathtt{J}})$$

[3] Note that if the second occurrence A(I,J) would become the candidate, we would obtain a structurally equivalent e-tag and f-tag for $\vec{\mathtt{I}} = (\mathtt{I}, \mathtt{J}, \mathtt{K})^T$.

Fourier-Motzkin elimination reveals the inconsistency of this system:

$$\ldots \rightarrow \left(\begin{array}{cc|c} 0 & 1 & 100 \\ -1 & 1 & 0 \\ 1 & -1 & -1 \\ 1 & -1 & 0 \\ -1 & 0 & -1 \\ 0 & 0 & 99 \\ 1 & 0 & 50 \end{array}\right) \rightarrow \left(\begin{array}{c|c} 1 & 99 \\ 1 & 100 \\ 1 & 50 \\ -1 & -1 \\ 0 & 0 \\ 0 & 99 \\ 0 & -1 \end{array}\right) \rightarrow \left(\begin{array}{c} 98 \\ 99 \\ 49 \\ 0 \\ -99 \\ -1 \end{array}\right)$$

Consequently, guard encapsulation is feasible, although we must account for possible insertions in other sparse vectors during the actual implementation. Note that if the execution set of the J-loop would be, for instance, $[\mathtt{I}, 100]$, then the system would be consistent and guard encapsulation would become disabled because insertions in the Ith sparse vector could occur during a fixed iteration $\mathtt{I} = i$.

Example: Although in the following example, guard '$(\mathtt{I}, \mathtt{J}) \in E(C)$' dominates the loop-body of the J-loop and the effective row-wise access patterns of occurrence C(I,J) are consistent with the storage patterns, guard encapsulation is disabled by the possibility of executing a STOP-statement in Q if we want to preserve the exact behavior of this program:

```
      PROGRAM MAIN                          SUBROUTINE P(X)
      REAL C(10,10), D(10)                  REAL X
C_SPARSE(C : _SPARSE()(0,1))                CALL Q(X)
      ...                                   RETURN
      DO I = 1, 10                          END
        DO J = 1, 10
          IF (C(I,J).NE.0.0) THEN
            CALL P(C(I,J))                  SUBROUTINE Q(Y)
            C(I,J) = 1 / D(J)               REAL Y
          ENDIF                             IF (Y.GE.500.0) THEN
        ENDDO                                 STOP
      ENDDO                                 ENDIF
      ...                                   RETURN
      END                                   END
```

If, for example, C(1,1)=500, C(1,2)=1.0, and D(2)=0, then division by zero could occur if iterations of the J-loop become reordered, whereas the original program terminates without any exception.

Candidates for Access Pattern Expansion

If, for any reason, guard encapsulation is not feasible for a candidate, then, in principle access pattern expansion is possible for this occurrence. Again, however, we impose an additional constraint to simplify code generation.

Let $0 \leq q < p$ denote the index of the last nonzero coefficient of the e-tag in (8.1), i.e. $e_q \neq 0$ and $e_i = 0$ for $q < i \leq d$. Then, a scatter and gather operation is generated at nesting depth q just before and after the $\mathtt{I}_{q+1}$-loop if the following constraint is satisfied:

- During each fixed iteration $\mathtt{I}_1 = i_1, \ldots, \mathtt{I}_q = i_q$, accesses induced by every individual occurrence of A are confined to either (i) the $e(\vec{\mathtt{I}})$th sparse vector of the pool, or (ii) other sparse vectors in the pool.

The constraint is verified as follows. All occurrences of A in the loop-body of the $\mathtt{I}_{q+1}$-loop for which the matching storage summary is identical to the storage summary that matches the candidate are examined.

First, the e-tag $e'(\vec{\mathtt{J}})$ of such an occurrence is constructed, where $\vec{\mathtt{J}}$ denotes the index vector of the loop in which this occurrence appears. If this e-tag is structurally equivalent to $e(\vec{\mathtt{I}})$, situation (i) occurs. Otherwise, a representation $A\vec{\mathtt{J}} \leq \vec{b}$ of the iteration space of the loop with index vector $\vec{\mathtt{J}}$ is constructed, in which loop indices with inadmissible loop bounds are left unbounded. Because the first q components of $\vec{\mathtt{I}}$ and $\vec{\mathtt{J}}$ are equal and $e(\vec{\mathtt{I}})$ only depends on $\mathtt{I}_1, \ldots, \mathtt{I}_q$, the constraint may be violated if the following equation has an integer solution for $A\vec{\mathtt{J}} \leq \vec{b}$,

$$e(\vec{\mathtt{I}}) = e'(\vec{\mathtt{J}})$$

Fourier-Motzkin elimination is used to test the consistency of the system of inequalities arising from the representation of the iteration space and this equation. If the system is consistent, or the e-tag $e'(\vec{\mathtt{J}})$ cannot be constructed due to inadmissible subscripts, we conservatively assume that the constraint is violated which disables access pattern expansion. In all other cases, we record that access pattern expansion at nesting depth q is feasible for the candidate (and for all occurrences in the loop-body of the $\mathtt{I}_{q+1}$-loop with a e-tag that is structurally equivalent to $e(\vec{\mathtt{I}})$).
Example: In the example of the previous section, occurrence `A(J,K)` with row-wise access patterns and the following e-tag and f-tag for $\vec{\mathtt{I}} = (\mathtt{I}, \mathtt{J}, \mathtt{K})^T$ may be involved in an access pattern expansion at nesting depth $q = 2$:

$$\begin{aligned} e(\vec{\mathtt{I}}) &= 0 \cdot \mathtt{I} + 1 \cdot \mathtt{J} \\ f(\vec{\mathtt{I}}) &= 0 \cdot \mathtt{I} + 0 \cdot \mathtt{J} + 1 \cdot \mathtt{K} \end{aligned}$$

Therefore, the other occurrences `A(J,K)` and `A(I,J)` are examined. Because the e-tag of the former is structurally equivalent to $e(\vec{\mathtt{I}})$, we only have to test whether $e(\vec{\mathtt{I}}) = e'(\vec{\mathtt{J}})$ may hold during any fixed iteration $\mathtt{I} = i$ and $\mathtt{J} = j$, where $\vec{\mathtt{J}} = (\mathtt{I}, \mathtt{J}, \mathtt{K})^T$ and $e'(\vec{\mathtt{J}}) = \mathtt{I}$ denotes the e-tag of `A(I,J)`. Clearly, this gives rise to the same inconsistent system of inequalities as was examined in the previous section. Consequently, expansion of the `J`th sparse vector (i.e. the `J`th row in this case) at nesting depth $q = 2$ is feasible for the occurrences `A(J,K)`.

8.2.3 Sparse Occurrences

After the preparatory pass over the sparse occurrences of all enveloping data structures in a program has been made, the actual sparse code is generated.

Implementation of Guard Encapsulation

If guard encapsulation is feasible for an an occurrence `A(`$F(\vec{\mathtt{I}})$`)` with e-tag $e(\vec{\mathtt{I}})$ and f-tag $f(\vec{\mathtt{I}})$ having the form (8.1) and a particular $\mathtt{I}_p$-loop with admissible loop bounds $\mathtt{L}_p$ and $\mathtt{U}_p$, then, depending on whether insertions in other sparse vectors of the pool are possible or not, the $\mathtt{I}_p$-loop is replaced by one of the following constructs, where $\mathtt{I}_p\_$ and possibly $\mathtt{LEN\_I}_p$ are two new locally declared scalar integer variables:

relative-addressing:

```
IF (Lp.LE.Up) THEN
  LEN_Ip = HGH_(e(I)) - LOW(e(I))
  DO Ip_ = 0, LEN_Ip
    Ip = IND_A(LOW_A(e(I)) + Ip_)
+       - f0 + f1*I1 - ... - fp-1*Ip-1
  ...
  ENDDO
ENDIF
```

absolute-addressing:

```
IF (Lp.LE.Up) THEN
  DO Ip_ = LOW_A(e(I)), HGH_A(e(I))
    Ip = IND_A(Ip_)
+       - f0 + f1*I1 - ... - fp-1*Ip-1
    ...
  ENDDO
ENDIF
```

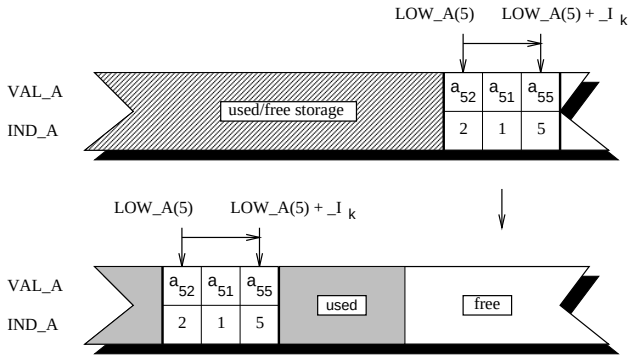


Figure 8.4: Left Compression (due to insertion in *another* sparse vector)

The outermost IF-statement is used to prevent the execution of this construct for zero trip loops. If the minimum value of the expression '$\mathtt{U}_p - \mathtt{L}_p$' is non-negative (which can be determined by applying the method presented at page 121 to the loop bounds in the original code), then this IF-statement is omitted. Relative-addressing induces slightly more overhead, but correctly accounts for any data movement caused by a left compression, because the relative order of entries is preserved by this operation (see primitive '_INS_' in section 8.1.2). For example, as illustrated in figure 8.4, while iterating over the entries in the 5th row of general row-wise storage, insertion in any other sparse row of the pool causing data movement is correctly dealt with using relative-addressing.

Within the loop-body of one of these two constructs, we test whether a generated entry actually belongs to an entry along the appropriate effective access pattern by testing inclusion of the restored loop index in the original execution set $[\mathtt{L}_p, \mathtt{U}_p]$ as follows:

```
IF (MOD(I_p, f_p) = 0) THEN
  I_p = I_p / f_p
  IF ((L_p.LE.I_p).AND.(I_p.LE.U_p)) THEN
      ...
  ENDIF
ENDIF
```

If $f_p = +1$ or $f_p = -1$, then the MOD-test and the integer division are omitted, although the sign of the restored loop index must still be reversed in the latter case. Furthermore, if inequality $\mathtt{L}_p \leq \mathtt{I}_p$ or $\mathtt{I}_p \leq \mathtt{U}_p$ is redundant with respect to the system obtained by substituting subscripts $F(\vec{\mathtt{I}})$ for (i, j) in the inequalities defining the simple section S of the matching storage summary $\langle S, \vec{s}, \mathbf{sparse} \rangle$ (tested with Fourier-Motzkin elimination according to proposition 2.1) then the corresponding lower or upper bound test is omitted since this implies that the test succeeds for every generated entry. Note that because bounds $\mathtt{L}_p$ and $\mathtt{U}_p$ are admissible, executing a possibly remaining IF-statement in each iteration is free of any side-effects.

Finally, at the position of the dots, we generate the loop-body of the original $\mathtt{I}_p$-loop in which every occurrence of the enveloping data structure A of which the subscripts are structurally equivalent to $F(\vec{\mathtt{I}})$ is replaced by one of the following expressions:

relative-addressing:	absolute-addressing:
VAL_A(LOW_A($e(\vec{\mathtt{I}})$) + $\mathtt{I}_{p\_}$)	VAL_A($\mathtt{I}_{p\_}$)

Example: Consider the following example, where an annotation is used to enforce the selection of general sparse row-wise storage for a 100×100 implicitly sparse matrix A with enveloping data structure A. Obviously, the guard '$(\mathtt{I}, 2 * \mathtt{J} - 1) \in E(A)$' dominates the loop-body of the J-loop, and guard encapsulation is feasible for A(I,2*J-1) with the tags $e(\vec{\mathtt{I}}) = \mathtt{I}$ and $f(\vec{\mathtt{I}}) = 2 * \mathtt{J} - 1$. Since insertions do not occur, absolute-addressing is used:

```
      REAL A(100,100)
C_SPARSE(A: _SPARSE( )(0,1))
      ...
      DO I = 1, 45
        DO J = I, I+5
          A(I,2*J-1) = A(I,2*J-1) * J
        ENDDO
      ENDDO
```

→

```
...
DO I = 1, 45
  DO J_ = LOW_A(I), HIGH_A(I)
    J = IND_A(J_) + 1
    IF (MOD(J, 2) .EQ. 0) THEN
      J = J / 2
      IF ((I.LE.J).AND.(J.LE.I+5)) THEN
        VAL_A(J_) = VAL_A(J_) * J
      ENDIF
    ENDIF
  ENDDO
ENDDO
```

Example: Consider the following example in which the elements in the lower triangular part of a 10×10 implicitly sparse matrix B are accumulated in the scalar variable LW:

```
      REAL B(10,10)
C_SPARSE(B : ...)
      DO I = 1, 10
        DO J = 1, I
          LW = LW + B(I,J)
        ENDDO
      ENDDO
```

The simple section X associated with the occurrence of B has the following form:

$$X = \{(i,j) \in \mathcal{Z}^2 \mid (1,1,2,0)^T \leq \mathcal{M}(i,j)^T \leq (10,10,20,9)^T\}$$

If the data dependences caused by the accumulation may be ignored, then one of the following fragments is generated, depending on whether we enforce the selection of, as illustrated in figure 8.5, general sparse row-wise or lower triangular sparse row-rise storage for B:

General Sparse Row-Wise:

```
DO I = 1, 10
  DO J_ = LOW_B(I), HGH_B(I)
    J = IND_B(J_)
    IF (J.LE.I) THEN
      LW = LW + VAL_B(J_)
    ENDIF
  ENDDO
ENDDO
```

Lower Triangular Sparse Row-Wise:

```
DO I = 1, 10
  DO J_ = LOW_B(I), HGH_B(I)
    J = IND_B(J_)        ! could be omitted
    LW = LW + VAL_B(J_)
  ENDDO
ENDDO
```

For general sparse row-wise storage, the test '`((1.LE.J).AND.(J.LE.I))`' can be simplified into '`(J.LE.I)`' with respect to the system obtained by substituting `(I,J)` for (i,j) in the simple section of the matching storage summary $\langle S, \vec{s}, \mathbf{sparse} \rangle$, where S describes the whole index set of B (viz. $X \subseteq S$, but $X \neq S$):

$$S = \{(i,j) \in \mathcal{Z}^2 \mid (1,1,2,-9)^T \leq \mathcal{M}(i,j)^T \leq (10,10,20,9)^T\}$$

On the other hand, for lower triangular sparse row-wise storage, the whole test may be omitted because both inequalities are redundant with respect to the system obtained by substituting `(I,J)` for (i,j) in the simple section of the matching storage summary $\langle S, \vec{s}, \mathbf{sparse} \rangle$, where S now contains the index set of the lower triangular part of B (viz. $X = S$). For example, `J` $\leq$ `I` may be omitted because Fourier-Motzkin elimination reveals the inconsistency of the following system $A(\mathtt{I},\mathtt{J})^T \leq \vec{b}$ (viz. negating `J` $\leq$ `I` for integer-valued variables yields `I` $-$ `J` ≤ -1):

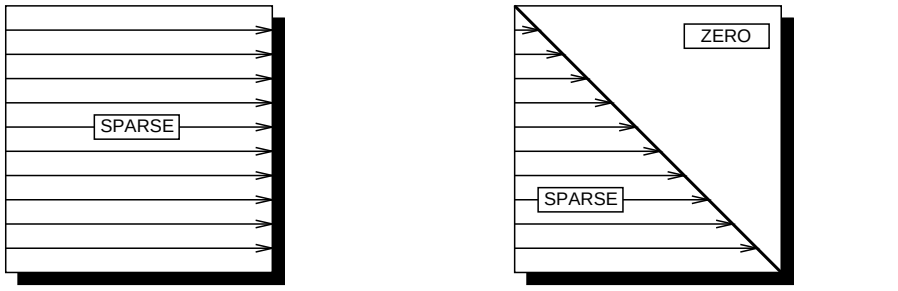


Figure 8.5: Possible Sparse Storage Schemes

$$\left(\begin{array}{rr|r} 0 & 1 & 10 \\ 1 & 1 & 20 \\ -1 & 1 & 0 \\ \hline 1 & -1 & -1 \\ \hline 0 & -1 & -1 \\ -1 & -1 & -2 \\ 1 & -1 & 9 \\ 1 & 0 & 10 \\ -1 & 0 & -1 \end{array}\right) \rightarrow \ldots \rightarrow \left(\begin{array}{c} \vdots \\ -1 \\ \vdots \end{array}\right)$$

In contrast, even if this storage scheme is selected for B in the following fragment, the test '`(J.LE.K)`' remains required after guard encapsulation, because each fixed iteration `K`$= k$ determines which part of the lower triangular is actually accessed:

```
DO I = 1, 10
  DO K = 1, I
    DO J = 1, K
      D(I,K) = D(I,K) + B(I,J)
    ENDDO
  ENDDO
ENDDO
```

→

```
DO I = 1, 10
  DO K = 1, I
    DO J_ = LOW_B(I), HGH_B(I)
      J = IND_B(J_)
      IF (J.LE.K) THEN
        D(I,K) = D(I,K) + VAL_B(J_)
      ENDIF
    ENDDO
  ENDDO
ENDDO
```

Implementation of Access Pattern Expansion

If access pattern expansion at nesting depth q if feasible for an an occurrence `A(`$F(\vec{\mathtt{I}})$`)` with e-tag $e(\vec{\mathtt{I}})$ and f-tag $f(\vec{\mathtt{I}})$ having the form (8.1), where $e_i = 0$ for $q < i < p$, then just before and after the $\mathtt{I}_{q+1}$-loop, the following CALL-statements are generated:

```
CALL _SCT_(VAL_A, IND_A, LOW_A(e(I)), HGH_A(e(I)), _AP_k, SWT_k)
DO I_{q+1} = L_{q+1}, U_{q+1}
  ...
ENDDO
CALL _GTH_(VAL_A, IND_A, LOW_A(e(I)), HGH_A(e(I)), _AP_k, SWT_k)
```

The label k is selected such that, on one hand, only a limited number of different arrays `_AP_`k and `SWT_`k are required, whereas, on the other hand, conflicts between storage required for simultaneously active access pattern expansions are avoided.

Each first underscore is replaced by a specification character in $\{\mathtt{I}, \mathtt{S}, \mathtt{D}, \mathtt{C}\}$, depending on the basis type of A. If storage summary $\overline{s} = \langle S, \vec{s}, \mathbf{sparse} \rangle$ matches the occurrence $\mathtt{A}(F(\vec{\mathtt{I}}))$, then the following declarations are generated, where TYPE denotes the basis type of the enveloping data structure A:

```
TYPE    _AP_k(1:U)
LOGICAL SWT_k(1:U)
```

The upper bound U is defined in terms of the boundary values in $\vec{\tau} \in \mathcal{Z}^4$ of $S \subseteq \mathcal{Z}^2$:

$$\mathtt{U} = \begin{cases} \tau_1 & \textbf{if } \theta(\overline{s}) = \text{row-indexed} \\ \tau_2 & \textbf{otherwise} \end{cases}$$

If the same identifiers would be generated several times (because label k may be re-used), then the maximum of all upper bounds is used in a single pair of declarations. In this manner, the storage required for access patterns expansions that cannot be simultaneously active may be re-used, thereby reducing the storage requirements and initialization time of the whole application. Eventually, all the arrays are placed in a single named COMMON-block, as illustrated below:

```
COMMON /STOR__/ SAP_10, SWT_10, SAP_20, SWT_20, ...
```

This COMMON-statement and the corresponding declarations are generated in the main program and in each clone in which access pattern expansion may occur.

Within the loop-body of the $\mathtt{I}_{q+1}$-loop, the following steps are applied to each occurrence of A with an e-tag $e'(\vec{\mathtt{J}})$ that is structurally equivalent to $e(\vec{\mathtt{I}})$, and an arbitrary f-tag $f'(\vec{\mathtt{J}})$. If the occurrence appears at the left-hand side of an assignment statement, this statement is replaced by the following construct, where the (possibly converted) right-hand side expression of the statement appears at the dots and L_ is a locally declared (dummy) scalar integer variable:

```
IF (.NOT. SWT_k(f'(J))) THEN
  CALL SINS__(VAL_A, IND_A, LOW_A, HGH_A, e'(J), NP_A, SZ_A, LST_A, L_, f'(J))
  SWT_k(f'(J)) = .TRUE.
ENDIF
_AP_k(f'(J)) = ...
```

Otherwise, the occurrence is simply replaced by '_AP_k(f'(J))'. Additionally, if the guard of the occurrence dominates the loop-body of the $\mathtt{I}_{q+1}$-loop, then this loop-body may be executed conditionally under the following test, which implies that the IF-statement shown above may be omitted for a left-hand side occurrence:

```
IF (SWT_k(f'(J))) THEN
  ...
ENDIF
```

Example: In the following example, occurrence A(I,I) is not considered as a candidate because it has diagonal-wise effective access patterns, whereas an annotation is used to enforce the selection of general sparse row-wise storage. However, access pattern expansion at nesting depth 1 is feasible for occurrence A(I,J), which also affects the former occurrence:

```
      REAL A(100,100), D(100,100)
C_SPARSE(A : _SPARSE()(0,1))
      ...
      DO I = 1, 100
        DO J = 1, 100
          D(I,J) = A(I,I) * A(I,J)
        ENDDO
      ENDDO
```

→

```
      ...
      REAL    SAP_10(1:100)
      LOGICAL SWT_10(1:100)
      COMMON  /STOR__/ SAP_10, SWT_10
      ...
      DO I = 1, 100
        CALL SSCT__(VAL_A, IND_A, LOW_A(I),
     +              HGH_A(I), SAP_10, SWT_10)
        DO J = 1, 100
          D(I,J) = SAP_10(I) * SAP_10(J)
        ENDDO
        CALL SGTH__(VAL_A, IND_A, LOW_A(I),
                    HGH_A(I), SAP_10, SWT_10)
      ENDDO
```

Example: Consider the first example of section 8.2.2 again. As stated before, guard encapsulation is feasible for occurrence `A(I,J)` (although insertions in other sparse vectors of the same pool may occur). Access pattern expansion at nesting depth 2 is feasible for `A(J,K)`. Therefore, eventually the following code is generated:

```
DO I = 1, 50
  LEN_J = HGH_A(I) - LOW_A(I)
  DO J_ = 0, LEN_J
    J = IND_A(LOW_A(I) + J_)
    IF (I+1.LE.J) THEN
      X(J) = X(J) + VAL_A(LOW_A(I) + J_)
      CALL SSCT__(VAL_A, IND_A, LOW_A(J), HGH_A(J), SAP_20, SWT_20)
      DO K = 1, 100
        IF (.NOT. SWT_20(K)) THEN
          SWT_20(K) = .TRUE.
          CALL SINS__(VAL_A, IND_A, LOW_A, HGH_A, J, 100, 1200, LST_A, L_, K)
        ENDIF
        SAP_20(K) = SAP_20(K) + VAL_A(LOW_A(I) + J_)
      ENDDO
      CALL SGTH__(VAL_A, IND_A, LOW_A(J), HGH_A(J), SAP_20, SWT_20)
      VAL_A(LOW_A(I) + J_) = VAL_A(LOW_A(I) + J_) * 2.0
    ENDIF
  ENDDO
ENDDO
```

Remaining Sparse Occurrences

The remaining sparse occurrences of the enveloping data structure `A` of an implicitly sparse matrix A in the program are handled as follows.

Suppose that $\overline{s} = \langle S, \vec{s}, \mathbf{sparse} \rangle \in \mathcal{S}_A$ matches a remaining sparse occurrence `A(E1,E2)`, where the subscripts are possibly inadmissible. An occurrence at the left-hand side of an assignment statement is replaced by the following construct, where the (possibly converted) right-hand side expression appear at the dots and `L_` is a new locally declared scalar integer variable:

```
L_ = LKP__(IND_A, LOW_A(E), HGH_A(E), F)
IF (L_ .EQ. ⊥) THEN
  CALL SINS__(VAL_A, IND_A, LOW_A, HGH_A, E, NP_A, SZ_A, LST_A, L_, F)
ENDIF
VAL_A(L_) = ...
```

In this construct, we define `F` as the compound expression '$base(\overline{s}) - \mathcal{L}(\overline{s}) + s_2 * \mathtt{E1} - s_1 * \mathtt{E2}$. Likewise, the expression `E` is defined as follows:

$$\mathtt{E} = \begin{cases} \mathtt{E1} & \text{if } \theta(\overline{s}) = \text{row-indexed} \\ \mathtt{E2} & \textbf{otherwise} \end{cases}$$

All occurrences with identical subscripts appearing at the right-hand side of the assignment statement are also replaced by the expression `VAL_A(L_)`.

Because subroutine `SINS` places a zero constant at the position of a new entry, these right-hand side occurrences correctly evaluate to zero in case of an insertion.

All other occurrences of an enveloping data structure `A` appearing at the right-hand side of assignment statements or appearing in arbitrary expressions of other statements are replaced by the following function call under the same definitions of `E` and `F` (but, of course, possibly with different subscripts `E1` and `E2`):

```
VAL_A( LKP__(IND_A, LOW_A(E), HGH_A(E), F) )
```

Because VAL_A($\perp$) $= 0$, this construct correctly accounts for the fact that the value of a non-entry is zero. Obviously, these replacements should only be used as a last resort by the compiler (i.e. in case overhead reducing techniques are not applicable). No attempts are made to account for any remaining conditions, because the potential gains of skipping the actual statements are probably small with respect to the incurred lookup overhead (although executing the first construct conditionally could reduce the amount of creation). In fact, frequent application of these replacements indicates that the conflicts in a program have not been resolved very well.
Example: Consider the following fragment, in which annotations are used to enforce the selection of general sparse row- and column-wise storage for the implicitly sparse matrices A and B having array A and B respectively as enveloping data structures:

```
      REAL A(15,15), B(20,20)
C_SPARSE(A : _SPARSE()(0,1))
C_SPARSE(B : _SPARSE()(1,0))
      ...
      A(2,3) = 7.0 * B(15,4) + 1.0 - A(2,3)
```

Because occurrences at nesting depth 0 cannot be involved in any overhead reducing technique, eventually the scalar statement is replaced by the following construct:

```
L_ = LKP__(IND_A, LOW_A(2), HGH_A(2), 3)
IF (L_ .EQ. ⊥) THEN
  CALL SINS__(VAL_A, IND_A, LOW_A, HGH_A, 2, NP_A, SZ_A, LST_A, L_, 3)
ENDIF
VAL_A(L_) = 7.0 * VAL_B(LKP__(IND_B, LOW_B(4), HGH_B(4), 15)) + 1.0 - VAL_A(L_)
```

8.3 Initialization Code Generation

While developing and testing the original dense program, the programmer can focus on the actual algorithms and very simple initialization code can be used for all enveloping data structures to test the program on some small dense matrices. Once the program has been debugged, all initialization code involving enveloping data structures is eliminated. The remaining dense program is used as input for the sparse compiler. After sparse storage schemes have been selected for all implicitly sparse matrices, the sparse compiler generates appropriate initialization code at the beginning of the main program, expecting all matrices in coordinate scheme. In this manner, the actual sparse storage schemes that are used in the generated program are kept completely transparent to the programmer.

8.3.1 Resetting Static Dense Storage and Switch Arrays

For each storage summary $\overline{s} = \langle S, \vec{s}, \mathbf{dense} \rangle \in \mathcal{S}_A$, the following code is used to reset all elements in the corresponding static dense storage, where $k = lab(\overline{s})$ and the outermost DO-loop and second subscript are omitted if $\mathcal{L}(\overline{s}) = \mathcal{U}(\overline{s})$ holds:

```
DO J_ = ℒ(s̄), 𝒰(s̄)
  DO I_ = L, U
    DNk_A(I_,J_) = 0.0
  ENDDO              ↑ zero constant of appropriate type
ENDDO
```

In this construct, the loop bounds L and U are defined as follows, where $\sigma \in \mathcal{Z}^4$ and $\tau \in \mathcal{Z}^4$ denote the boundary values of the simple section $S \subseteq \mathcal{Z}^2$:

$$(\mathtt{L}, \mathtt{U}) = \begin{cases} (\sigma_1, \tau_1) & \textbf{if } \theta(\overline{s}) = \text{row-indexed} \\ (\sigma_2, \tau_2) & \textbf{otherwise} \end{cases}$$

Likewise, for each pair _AP_k/SWT_k in the named COMMON-block 'STOR__' (used to support expansion and compression), the following code is generated, where U denotes the maximum upper bound recorded for each pair:

```
DO I_ = 1, U
  SWT_k = .FALSE.
  _AP_k = 0.0
ENDDO        ↑ zero constant of appropriate type
```

8.3.2 File Input

For each implicitly sparse matrix A with enveloping data structure A in the program, the following construct is generated:

```
      LST_A = 1
      OPEN (UNIT=1, FILE='file_name', STATUS='OLD')
      READ (1,*) M_, N_, NNZ_
      DO K_ = 1, NNZ_
        READ (1,*) I_, J_, V_
C Insert A(I_,J_) = V_ in Selected Data Structure
        ...
      ENDDO
      CLOSE (UNIT = 1)
```

In case a pool of sparse vectors is used as dynamic storage for the entries in the sparse regions of A, the scalar 'LST_A' is initialized to 1 because the first location in the parallel arrays implementing this pool is used as location $\perp$.

Thereafter, a construct that reads the matrix from file in coordinate scheme is generated, where the string 'file_name' is the file name defined for A (a similar fragment with 'A(I_,J_)=V_' can be manually inserted in the original dense program for testing purposes). If no file has been specified, the sparse compiler inquires the programmer for a file name.

At the dots, a multi-way IF-statement appears that determines the action required for each entry. Suppose we have the following set:

$$\mathcal{S}_A^{s,d} = \{\langle S, \vec{s}, p\rangle \mid p = \mathbf{sparse} \vee p = \mathbf{dense}\} = \{S_1, \ldots, S_k\}$$

Then, the following IF-statement is generated, in which the condition of the last branch is omitted, and the whole statement is omitted if $|\mathcal{S}_A^{s,d}| = 1$:

```
IF (I_,J_) ∈ S_1 THEN
  ...
ELSEIF (I_,J_) ∈ S_2 THEN
  ⋮
ELSE IF (I_,J_) ∈ S_{k-1} THEN
  ...
ELSE
  ...
ENDIF
```

Testing inclusion in each individual simple section $S \subseteq \mathcal{Z}^2$ with boundary values $\sigma \in \mathcal{Z}^4$ and $\tau \in \mathcal{Z}^4$ can be done as follows:

```
 IF ( (σ1.LE.I_   ).AND.(I_   .LE.τ1).AND.(σ2.LE.   J_).AND.(   J_.LE.τ2).AND.
+     (σ3.LE.I_+J_).AND.(I_+J_.LE.τ3).AND.(σ4.LE.I_-J_).AND.(I_-J_.LE.τ4) ) THEN
  ...
```

This inclusion test may induce substantial testing overhead. However, usually the condition can be simplified by omitting tests on lower or upper bounds that coincide with the minimum or maximum possible value of the expressions I_, J_, I_+J_ and I_-J_ under the equivalence (I_, J_) $\in [1, m] \times [1, n]$ or that are implied by inequalities that have already been generated. Moreover, a non-redundant ith pair of tests can be replaced by a single test for equivalence if $\sigma_i = \tau_i$ (e.g. (I_.EQ.σ_1)).

Dense Branch

In a branch corresponding to a storage summary $\overline{s} = \langle S, (s_1, s_2)^T, \mathbf{dense} \rangle$, the following construct is generated where `F=I_` if $\theta(\overline{s})$ = row-indexed and `F=J_` otherwise:

```
DNk_A(F, s2 * I_ - s1 * J_) = V_
```

Sparse Branch

In a branch corresponding to a storage summary $\overline{s} = \langle S, (s_1, s_2)^T, \mathbf{sparse} \rangle$, the following construct is generated, where `F=I_` if $\theta(\overline{s})$ = row-indexed and `F=J_` otherwise:

```
LST_A        = LST_A + 1
VAL_A(LST_A) = V_
IND_A(LST_A) = F
TMP__(LST_A) = base(s) - L(s) + s2 * I_ - s1 * J_
```

In this manner, eventually the parallel arrays `VAL_A` and `IND_A` together with the temporary array `TMP__` of the same size contain the numerical value, the index information and the number of the sparse vector for each entry in a sparse region of A.

If a pool of sparse vectors is used as dynamic storage of the sparse regions of A, then the following call is generated after the multi-way IF-statement to initialize this pool, where the first underscore is replaced by the appropriate type specification in $\{$`I`, `S`, `D`, `C`$\}$, depending on the basis type of the enveloping data structure `A`:

```
CALL _INI_(VAL_A, TMP__, IND_A, LOW_A, HGH_A, NP_A, SZ_A, LST_A)
```

Example: Consider the following annotations for a 1000×1000 implicitly sparse matrix A with enveloping data structure `A`:

```
      INTEGER   N
      PARAMETER (N=1000)
      REAL      A(N,N)

C_SPARSE(A : _DENSITY(0.01))
C_SPARSE(A : _FILE('mat.cs'))
C_SPARSE(A : _SPARSE(1-N <= I-J <=  -1)(0,1))
C_SPARSE(A : _DENSE (  0 <= I-J <=   0)(1,1))
C_SPARSE(A : _SPARSE(  1 <= I-J <= N-1)(1,0))
```

These annotations enforce the selection of a storage scheme in which the main diagonal is stored in static dense storage, whereas a pool of column- and row-wise sparse vectors is used to dynamically store entries in the strict lower and strict upper triangular part of A.

If we assume that we actually obtain the set $\Sigma_A = \{S_1, S_2, S_3\}$ in which the simple sections describe the index sets of the strict upper and strict lower triangular part and the main diagonal of A respectively, then we obtain the storage summary set $\mathcal{S}_A = \{\overline{s}_1, \overline{s}_2, \overline{s}_3\}$ with the following storage summaries:

				$\mathcal{L}(\overline{s}_i)$	$\mathcal{U}(\overline{s}_i)$	$base(\overline{s}_i)$	$lab(\overline{s}_i)$
$\overline{s}_1$	$=$	$\langle S_1, \ (0,1)^T, \ \mathbf{sparse} \rangle$		1	999	1	–
$\overline{s}_2$	$=$	$\langle S_2, \ (-1,0)^T, \ \mathbf{sparse} \rangle$		1	999	1000	–
$\overline{s}_3$	$=$	$\langle S_3, \ (1,1)^T, \ \mathbf{dense} \rangle$		0	0	–	1

Because $N_A = |S_1| + |S_2| = 999000$, and $V_A = 1998$, we set `NP_A=1998` and `SZ_A=13986` (viz. $0.01 \cdot 999000 + 2 \cdot 1998$). The following declarations are generated by the sparse compiler:

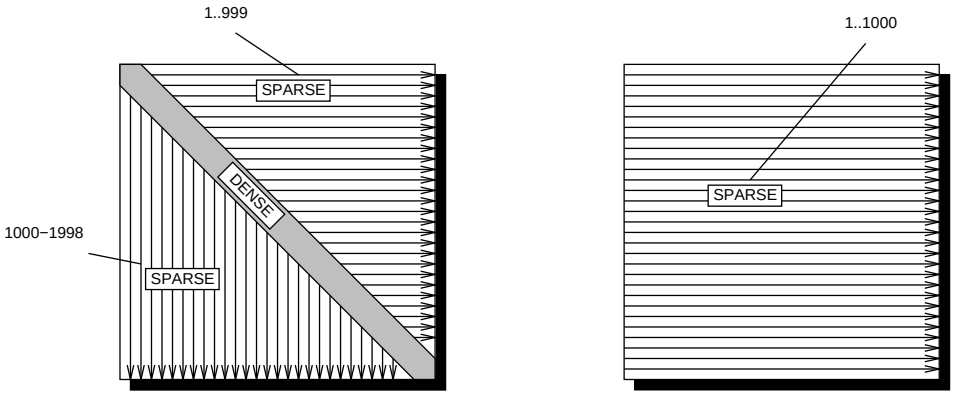


Figure 8.6: Two Different Sparse Storage Schemes

```
REAL    VAL_A(1:13986), DN1_A(1:1000), V_
INTEGER IND_A(1:13986), LOW_A(1:1998), HGH_A(1:1998), LST_A
INTEGER TMP__(1:13986), I_, J_, K_, M_, N_, NNZ_

COMMON  /A/  DN1_A, VAL_A, IND_A, LOW_A, HGH_A, LST_A
```

Array TMP__ and scalars V_, I_ J_, K_, M_, N_, and NNZ_ are temporarily required for the initialization. The other variables actually implement static dense storage for the main diagonal, and a pool of 1998 row- and column-wise sparse vectors as dynamic storage for the strict upper and strict lower triangular part, as illustrated in the first picture of figure 8.6. Therefore, these variables are placed in a named COMMON-block with label A, which will be made accessible in all clones in which the enveloping data structure is uniquely associated with a formal argument.

At the beginning of the program, the following DO-loop is generated to reset the elements of the static storage:

```
DO I_ = 1, 1000
  DN1_A(I_) = 0.0
ENDDO
```

Furthermore, because the matrix is stored in the file '`mat.cs`', the following code is generated which will initialize the sparse storage scheme selected for A at run-time:

```
LST_A = 1
OPEN (UNIT=1, FILE='mat.cs', STATUS='OLD')
READ (1,*) M_, N_, NNZ_
DO K_ = 1, NNZ_, 1
  READ (1,*) I_, J_, V_
  IF (((I_-J_).EQ.0)) THEN
    DN1_A(J_)   = V_
  ELSE IF (((I_-J_).LE.-1)) THEN
    LST_A       = LST_A + 1
    VAL_A(LST_A) = V_
    IND_A(LST_A) = J_
    TMP__(LST_A) = I_
  ELSE
    LST_A       = LST_A + 1
    VAL_A(LST_A) = V_
    IND_A(LST_A) = I_
    TMP__(LST_A) = J_ + 999
  ENDIF
ENDDO
CLOSE (UNIT=1)
CALL SINI_(VAL_A,TMP__,IND_A,LOW_A,HGH_A,1998,13986,LST_A)
```

A multi-way IF-statement is executed to determine which action is required for each entry. An entry a_{ij} with $i = j$ is placed in the jth location of array DN1_A.

Matrix	n	τ	Row-Wise	LDU	Read (HB)
jpwh_991	991	6027	1.1	1.1	0.5
gre_1107	1107	5664	1.1	1.1	0.5
orani678	2529	90158	12.9	13.3	7.8
lns_3937	3937	25407	4.0	4.1	2.3
psmigr_1	3140	543162	73.1	73.8	31.7

Table 8.2: Initialization Time in seconds on an HP 9000/720

An entry a_{ij} with either $i < j$ or $i > j$ is placed temporarily in coordinate-scheme like storage as an entry in the ith sparse vector with index information j or in the $(999 + j)$th sparse vector with index information i respectively, since $\theta(\overline{s}_1) =$ row-indexed and $\theta(\overline{s}_2) =$ column-indexed. After the file has been read completely, a call to `SINI_` converts this temporary storage into the selected storage scheme, after which array `TMP_` can be re-used for other initializations.

Note that for general sparse row-wise storage, enforced by the following annotation, only the second branch of the multi-way IF-statement would result:

```
C_SPARSE(A: SPARSE()(0,1))
```

In figure 8.2, we show the execution time in seconds on an HP 9000/720 for initializing general row-wise storage and the previous presented LDU-scheme. Moreover, in the last column, the time required to read the matrix from file using the column-wise Harwell-Boeing standard sparse matrix format [79] is shown. Obviously, although using coordinate scheme to initialize the matrices is substantially more expensive than using the column-wise Harwell-Boeing standard sparse matrix format, the execution time of from-file initialization of a general sparse row-wise storage and the execution time of initializing a more advanced sparse data structure are comparable.

Chapter 9

Initial Experimentation

To test the feasibility of automatically converting a dense program into semantically equivalent sparse code, the automatic data structure selection and transformation method has been actually incorporated in the prototype source to source restructuring compiler MT1 [24, 37, 45]. In this chapter, we present some qualitative and quantitative experiments that have been conducted with the prototype sparse compiler.

9.1 Qualitative Experiments

In this section, we take a closer look at some sparse constructs generated by the sparse compiler. First, we examine constructs for general sparse matrices. Thereafter, we show how characteristics of the nonzero structure can be accounted for. Finally, we illustrate how procedure cloning enables the application of program and data structure transformations.

9.1.1 Constructs for General Sparse Matrices

For general sparse matrices, we can distinguish between so-called static and simply dynamic operations, where the nonzero structures of all sparse matrices involved remains fixed, although the values of entries may change for the latter operations, and essentially dynamic operations, where nonzero structures may change [235, p10-12].

Static and Simply Dynamic Operations

In many static and simply dynamic operations, the loop-body of a loop only has to be executed for the entries of an implicitly sparse matrix. This situation occurs, for instance, in the following fragment in which the elements of an implicitly sparse matrix A with enveloping data structure A are scaled and the position and actual value of an element with largest absolute value are determined:

```
      REAL A(M,N)
C_SPARSE(A)
      ...
      DO I = 1, M
        DO J = 1, N
          A(I,J) = A(I,J) / 3.0              ← (I,J) ∈ E(A)
          IF (ABS(A(I,J).GT.ABS(MX))) THEN   ← (I,J) ∈ E(A)
            II = I                           ← true
            JJ = J                           ← true
            MX = A(I,J)                      ← true
          ENDIF
        ENDDO
      ENDDO
```

The first assignment statement only has to be executed for entries, since the division has no impact on zero elements. Hence, the sparse compiler associates condition '$(\mathtt{I},\mathtt{J}) \in E(A)$' with this assignment statement. Moreover, although none of the statements inside the one-way IF-statement can exploit sparsity, as reflected by the condition '**true**', the IF-statement as a whole only has to be executed for entries, because the condition of this IF-statement always fails for zero elements. Hence, if general sparse row-wise storage is selected for A and the programmer indicates that all loop-carried data dependences may be ignored (under the assumption that *any* element with largest absolute value may be found), then encapsulation of the guard '$(\mathtt{I},\mathtt{J}) \in E(A)$' in the execution set of the J-loop becomes feasible:

```
DO I = 1, M
  DO J_ = LOW_A(I), HGH_A(I)
    J = IND_A(J_)
    VAL_A(J_) = VAL_A(J_) / 3.0
    IF (ABS(VAL_A(J_).GT.ABS(MX))) THEN
      II = I
      JJ = J
      MX = VAL_A(J_)
    ENDIF
  ENDDO
ENDDO
```

After this conversion, the J_-loop iterates over all entries within each Ith row, of which the numerical values and column index information can be found at locations LOW_A(I)..HGH_A(I) in the parallel arrays VAL_A and IND_A respectively. Although the loop-body of the resulting loop is executed less frequently, and possibly in a different order because no ordering is imposed on the entries in each row, the semantics of the program are preserved.

As shown below, a dense implementation of the operation $\vec{b} \leftarrow \vec{b} + A\vec{x}$ with A implicitly sparse can be converted similarly if general sparse *column*-wise storage is selected for the matrix:

```
DO J = 1, N                                    DO J = 1, N
  DO I = 1, M                                    DO I_ = LOW_A(J), HGH_A(J)
    B(I) = B(I) + A(I,J) * X(J)      →             I = IND_A(I_)
  ENDDO                                          B(I) = B(I) + VAL_A(I_) * X(J)
ENDDO                                          ENDDO
                                               ENDDO
```

After this conversion, the I_-loops implements a sparse SAXPY ($\vec{y} \leftarrow \vec{y} + \alpha\vec{x}$, where $\vec{x}$ is sparse). Likewise, if loop interchanging is applied to the original loop and general sparse row-wise storage is selected for A, then the sparse compiler generates code that implements a sequence of sparse dot products ($w = \vec{x} \cdot \vec{y}$, where $\vec{x}$ is sparse), as will be shown in section 9.2.2. Hence, in a future implementation such constructs could be replaced by calls to primitives of the sparse extensions to BLAS [68] or directly by an efficient implementation (such as the GATHER-SAXPY-SCATTER implementation of sparse SAXPY for pipelined vector processors [65, 69, 76, 137, 184]).

The conversion into sparse code becomes more complex if the condition associated with a statement in a loop consists of a conjunction of guards, such as in the following example, where arrays A and B are used as enveloping data structure of implicitly sparse matrices A and B:

```
      REAL A(M,N), B(M,N)
C_SPARSE(A ; B)
      ...
      DO I = 1, M
        DO J = 1, N
          X = X + A(I,J) * B(I,J)     ← (I,J) ∈ E(A) ∧ (I,J) ∈ E(B)
        ENDDO
      ENDDO
```

If general sparse row-wise storage is selected for A and B and the data dependences caused by the accumulation may be ignored, then either the guard '$(\texttt{I}, \texttt{J}) \in E(A)$' or '$(\texttt{I}, \texttt{J}) \in E(B)$' can be encapsulated in the execution set of the J-loop, but not both.

To prevent the situation in which a lookup would have to be performed for each entry of either A or B, the sparse compiler uses expansion. For example, if in the generated sparse code, guard '$(\texttt{I}, \texttt{J}) \in E(A)$' is encapsulated in the execution set of the J-loop, the Ith row of B is expanded before operated upon:

```
DO I = 1, M
  CALL SSCT__(VAL_B ,IND_B, LOW_B(I), HGH_B(I), SAP_10, SWT_10)
  DO J_ = LOW_A(I), HGH_A(I)
    J = IND_A(J_)
    IF (SWT_10(J)) THEN
      X = X + VAL_A(J_) * SAP_10(J)
    ENDIF
  ENDDO
  CALL SGTH__(VAL_B ,IND_B, LOW_B(I), HGH_B(I), SAP_10, SWT_10)
ENDDO
```

Note that, although matrix B remains unaffected in this loop, the gather operation is generated after the J_-loop all the same. In this manner, used elements in array SAP_10 and the switch array SWT_10 are reset to enable each subsequent expansion. Because the time required to perform each scatter and gather operation is proportional to the number of entries in the corresponding row of B, and the initial costs of resetting the full-sized arrays SAP_10 and SWT_10 can be amortized over M expansions, a construct of which the execution time is proportional to the number of entries in A and B has been obtained.

A similar problem arises if the condition that is associated with a statement in a loop consists of a disjunction of guards, such as in the following implementation of $D \leftarrow D + A + B$, where we assume that D is used to store the elements of a dense matrix D:

```
      REAL A(M,N), B(M,N), D(M,N)
C_SPARSE(A ; B)
      ...
      DO I = 1, M
        DO J = 1, N
          D(I,J) = D(I,J) + A(I,J) + B(I,J)     ← (I,J) ∈ E(A) ∨ (I,J) ∈ E(B)
        ENDDO
      ENDDO
```

Because none of the guards dominates the condition, in this case guard encapsulation is even infeasible.[1] Clearly, performing a lookup in both A and B for each element of D would induce an unacceptable complexity. Fortunately, after this operation has been rewritten into the operations $D \leftarrow D + A$ and $D \leftarrow D + B$ using the transformations update expression splitting and loop distribution (see section 5.3.4), guard encapsulation becomes feasible if general sparse row-wise storage is selected for both A and B:

```
DO I = 1, M                                  DO I = 1, M
  DO J = 1, N                                  DO J_ = LOW_A(I), HGH_A(I)
    D(I,J) = D(I,J) + A(I,J)                     J = IND_A(J_)
  ENDDO                                          D(I,J) = D(I,J) + VAL_A(J_)
  DO J = 1, N                          →       ENDDO
    D(I,J) = D(I,J) + B(I,J)                   DO J_ = LOW_B(I), HGH_B(I)
  ENDDO                                          J = IND_B(J_)
ENDDO                                            D(I,J) = D(I,J) + VAL_B(J_)
                                               ENDDO
                                             ENDDO
```

[1] Encapsulation of a conjunction or disjunction of guards could be implemented efficiently if an ordering is imposed on the entries in each sparse vector using an **in-phase scan** [78, p20-21]. Because the selection of ordered sparse storage is not supported by the prototype sparse compiler, however, these constructs are not further considered.

Essentially Dynamic Operations

In the following two fragments, the nonzero structure of each row of A is obtained by respectively performing an or- and and-operation to the nonzero structure of the original row of A and the corresponding row of B, as illustrated in figure 9.1:

```
     REAL A(M,N), B(M,N)
C_SPARSE(A ; B)
      ...
     DO I = 1, M
       DO J = 1, N
         A(I,J) = A(I,J) + B(I,J)
       ENDDO
     ENDDO
```

```
      REAL A(M,N), B(M,N)
C_SPARSE(A ; B)
      ...
     DO I = 1, M
       DO J = 1, N
         A(I,J) = A(I,J) * B(I,J)
       ENDDO
     ENDDO
```

Because condition '$(\mathtt{I},\mathtt{J}) \in E(B)$' is associated with the assignment statement in the first loop, the sparse compiler implements $A \leftarrow A + B$ as follows if general sparse row-wise storage is selected for both implicitly sparse matrices:

```
DO I = 1, M
  CALL SSCT__(VAL_A, IND_A, LOW_A(I), HGH_A(I), SAP_10, SWT_10)
  DO J_ = LOW_B(I), HGH_B(I)
    J = IND_B(J_)
    IF (.NOT.SWT_10(J)) THEN
      SWT_10(J) = .TRUE.
      CALL SINS__(VAL_A, IND_A, LOW_A, HGH_A, I, NP_A, SZ_A, LST_A, L_, J)
    END IF
    SAP_10(J) = SAP_10(J) + VAL_B(J_)
  ENDDO
  CALL SGTH__(VAL_A, IND_A, LOW_A(I), HGH_A(I), SAP_10, SWT_10)
ENDDO
```

Consequently, the OR-operation is implemented by iterating over the entries in a row of B after the corresponding row of A has been expanded. The switch array `SWT_10` is used to determine where creation occurs. After all entries in a row of B have been considered, the entries in the expanded row of A are gathered back into a sparse vector. In fact, similar implementations are obtained for adding a number of implicitly sparse matrices if first update expression splitting and loop distribution are applied (cf. previous section). In essence, these automatically generated sparse implementations are similar to the code for adding sparse matrices found in [169, p242-247], although in the latter code, symbolic and numerical operations are separated.

Because condition '$(\mathtt{I},\mathtt{J}) \in E(A)$' is associated with the assignment statement in the second loop, the sparse compiler implements the scaling of matrix A with elements of B as follows if general sparse row-wise is selected for both A and B:

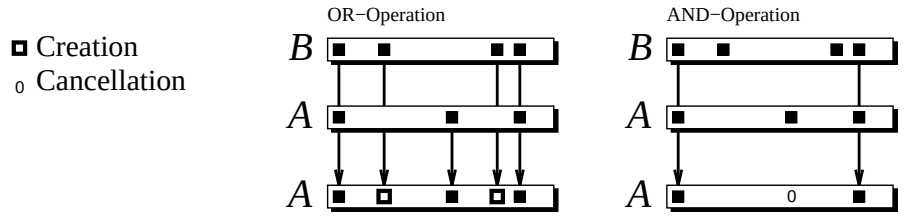


Figure 9.1: OR- and AND-operation

```
DO I = 1, M
  CALL SSCT__(VAL_B, IND_B, LOW_B(I), HGH_B(I), SAP_10, SWT_10)
  DO J_ = LOW_A(I), HGH_A(I)
    J = IND_A(J_)
    VAL_A(J_) = VAL_A(J_) * SAP_10(J)
  ENDDO
  CALL SGTH__(VAL_B, IND_B, LOW_B(I), HGH_B(I), SAP_10, SWT_10)
ENDDO
```

Hence, the situation is in fact handled as a simply dynamic operation by ignoring any cancellation caused by applying an AND-operation to the nonzero structures of a corresponding row of A and B (besides the fact that *exact* cancellation, where the subtraction of two entries is accidentally zero, is also ignored). A construct that resets some elements of the implicitly sparse matrix A is implemented similarly, i.e. the value of each entry in the regions that become zero is simply reset rather than deleting the entry explicitly, as shown below where we assume that $\mathtt{M} \leq \mathtt{N}$:

```
DO I = 1, M, 2
  DO J = 1, I
    A(I,J) = 0.0
  ENDDO
ENDDO
```

$\rightarrow$

```
DO I = 1, M, 2
  DO J_ = LOW_A(I), HGH_A(I)
   J = IND_A(J_)
   IF (J.LE.I) VAL_A(J_) = 0.0
  ENDDO
ENDDO
```

9.1.2 Characteristic of Nonzero Structures

Although in the previous section, efficient sparse code has been obtained for general sparse matrices, the prototype sparse compiler becomes more powerful if particular characteristics of the nonzero structure of implicitly sparse matrices are accounted for during this conversion.

Suppose that the operation $\vec{b} \leftarrow \vec{b} + A\vec{x}$ is applied to a 15×15 implicitly sparse matrix A having the nonzero structure shown in figure 9.2. If at compile-time the matrix is available on file, the nonzero structure analyzer of the sparse compiler can identify the zero and dense regions in this matrix, where the programmer is inquired whether the zero regions will be preserved at run-time. Likewise, this information can be supplied to the sparse compiler using the following annotations:

```
     REAL A(15,15)
C_SPARSE(A: _DENSE(15 <= I <= 15))
C_SPARSE(A: _DENSE( 1 <= I <= 14,  15 <=     J <= 15) )
C_SPARSE(A: _DENSE( 1 <= I <= 14,   0 <= I - J <=  0) )
C_SPARSE(A:  _ZERO( 2 <= I <= 14,   1 <= I - J <= 13) )
C_SPARSE(A:  _ZERO( 1 <= J <= 14, -13 <= I - J <= -1) )
```

Subsequently, iteration space partitioning is applied to the dense implementation to separate operations on zero regions from operations on dense regions:

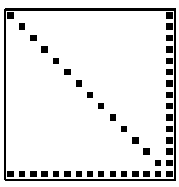


Figure 9.2: Nonzero Structure

```
DO I = 1, 15
  DO J = 1, 15
    B(I) = B(I) + A(I,J) * X(J)
  ENDDO
ENDDO
```

→

```
DO I = 1, 14
  DO J = 1, I-1
    B(I) = B(I) + A(I,J) * X(J)
  ENDDO
  B(I) = B(I) + A(I,I) * X(I)
  DO J = I+1, 14
    B(I) = B(I) + A(I,J) * X(J)
  ENDDO
  B(I) = B(I) + A(I,15) * X(15)
ENDDO
DO J = 1, 15
  B(15) = B(15) + A(15,J) * X(J)
ENDDO
```

Because this iteration space partitioning is successful, the sparse compiler decides to use the following static dense storage for the dense regions of A:

```
REAL       DN1_A(1:14), DN2_A(1:14), DN3_A(1:15)
COMMON /A/ DN1_A,       DN2_A,       DN3_A
```

Thereafter, all occurrences of A are either replaced by a zero constant or by an appropriate occurrence of this static dense storage. Finally, the condition of each statement in which such a replacement occurs is re-computed and redundant assignment statements and DO-loops are eliminated at compile-time:

```
DO I = 1, 14
  DO J = 1, I-1
    B(I) = B(I) + 0.0 * X(J)
  ENDDO
  B(I) = B(I) + DN2_A(I) * X(I)
  DO J = I+1, 14
    B(I) = B(I) + 0.0 * X(J)
  ENDDO
  B(I) = B(I) + DN1_A(I) * X(15)
ENDDO
DO J = 1, 15
  B(15) = B(15) + DN3_A(J) * X(J)
ENDDO
```

→

```
DO I = 1, 14
  B(I)  = B(I)  + DN2_A(I) * X(I)
  B(I)  = B(I)  + DN1_A(I) * X(15)
ENDDO
DO J = 1, 15
  B(15) = B(15) + DN3_A(J) * X(J)
ENDDO
```

Hence, the original dense implementation has been automatically converted into an implementation that is specially tailored for the particular sparse matrix of figure 9.2. Obviously, if the characteristics of the nonzero structure that can be exploited become more complex, the transformations required to do such a conversion also become more complex as more iteration space partitioning and access pattern reshaping becomes required. This strongly motivates the use of a sparse compiler to perform this conversion.

9.1.3 Subroutines and Functions

Procedure cloning enables the sparse compiler to apply program and data structure transformations to the code and formal arguments in all procedure clones without interfering with other uses of these subroutines and functions. Moreover, because procedure cloning usually improves the results of interprocedural constant propagation, loop bounds and subscript functions involving formal arguments may become admissible, which enables the application of more accurate program analysis and transformations.

Consider, for example, the following program in which annotations are used to inform the compiler about the fact that array A is used as enveloping data structure of an implicitly sparse matrix A in diagonal form, i.e. $a_{ij} \neq 0 \Rightarrow i = j$:

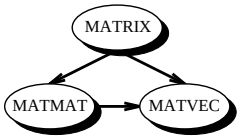


Figure 9.3: Call Graph of Program MATRIX

```
      PROGRAM MATRIX                          SUBROUTINE MATMAT(H, F, G, N)
                                              INTEGER N, I
      INTEGER    M, N                         REAL    H(N,N), F(N,N), G(N,N)
      PARAMETER (M = 50, N = 100)             DO I = 1, N
                                               CALL MATVEC(H, F(1,I), G(1,I), N)
      REAL       A(M,M), B(M,M), C(M,M)       ENDDO
      REAL       D(N,N), E(N,N)               RETURN
      REAL       X(N),   Y(N)                 END

C_SPARSE(A: _ZERO ( 1-M <= I - J <=  -1))     SUBROUTINE MATVEC(H, R, S, N)
C_SPARSE(A: _DENSE(   0 <= I - J <=   0))     INTEGER N, I, J
C_SPARSE(A: _ZERO (   1 <= I - J <= M-1))     REAL    H(N,N), R(N), S(N)
      ...                                     DO I = 1, N
      CALL MATMAT(A, B, C, M)                   DO J = 1, N
      CALL MATMAT(D, D, E, N)                     S(I) = S(I) + H(I,J) * R(J)
      CALL MATVEC(E, X, Y, N)                   ENDDO
      ...                                     ENDDO
                                              RETURN
      END                                     END
```

In this program, the call graph of which is shown in figure 9.3, the sparse compiler cannot bluntly apply data structure transformations to the formal arguments H in MATMAT and MATVEC, because these subroutines are also used to perform operations involving dense matrices. Therefore, the sparse compiler generates a clone MATMAT_A000 of the subroutine MATMAT in which A is uniquely associated with the formal argument H. Moreover, since the clone calls MATVEC with H as first actual argument, a clone MATVEC_A000 of MATVEC is also generated. The original subroutines are preserved to perform the operations $E \leftarrow E + DD$ and $\vec{y} \leftarrow \vec{y} + E\vec{x}$, whereas the clones are used to compute $C \leftarrow C + AB$.

If static dense storage is selected for the main diagonal of A, this data structure is placed in a named COMMON block, and the main program is converted as shown below, where the argument used to pass the whole implicitly sparse matrix has been eliminated:

```
PROGRAM MATRIX
...
REAL       DN1_A(1:50)
COMMON /A/ DN1_A
...
CALL MATMAT_A000(B, C, M)
CALL MATMAT     (D, D, E, N)
CALL MATVEC     (E, X, Y, N)
...
END
```

After iteration space partitioning has been applied to the procedure clones, and redundant assignment statements and DO-loops have been eliminated at compile-time, the sparse code shown below results. Interprocedural constant propagation has derived the value N=50 and the formal argument H has been either eliminated or replaced by an occurrence of the selected storage scheme, made available to the subroutines using the named COMMON-block:

```
SUBROUTINE MATMAT_A000(F, G, N)
INTEGER     N, I
REAL        F(50,50), G(50,50)
REAL        DN1_A(1: 50)
COMMON  /A/ DN1_A

DO I = 1, 50
  CALL MATVEC_A000(F(1,I), G(1,I), 50)
ENDDO
RETURN
END
```

```
SUBROUTINE MATVEC_A000(R, S, N)
INTEGER     N, I, J
REAL        R(50), S(50)
REAL        DN1_A(1:50)
COMMON  /A/ DN1_A

DO I = 1, 50
  S(I) = S(I) + DN1_A(I) * R(I)
ENDDO
RETURN
END
```

A subroutine computing the product of A with another matrix and a subroutine computing the product of A with a vector tailored for the specific nonzero structure of A has been derived automatically.

9.2 Quantitative Experiments

In this section, quantitative experiments are conducted with some small sparse programs that have been generated automatically by the prototype sparse compiler.

9.2.1 Preliminary Discussion

The experiments have been conducted on an HP 9000/720 and on one CPU of a Cray C98/4256. On both machines, all programs are compiled with the native FORTRAN compiler, where default optimizations and, for the latter, vectorization are enabled. Furthermore, experiments have been conducted with sparse matrices of the $E(n, c)$-class of [164, p6-11][235, p57-62], which are $n \times n$ matrices A having the following nonzero elements:

$$\begin{cases} a_{ii} & = +4.0 & i = 1, \ldots, n \\ a_{i,i+1} = a_{i+1,i} & = -1.0 & i = 1, \ldots, n-1 \\ a_{i,i+c} = a_{i+c,i} & = -1.0 & i = 1, \ldots, n-c \end{cases}$$

In fact, these matrices form simplifications of typical matrices arising in finite difference methods (cf. figure 6.10). In figure 9.4, the nonzero structure of $E(20, 5)$ is given. Although these matrices have a very simple nonzero structure, using this class enables us to test the generated sparse program for varying matrix sizes. Moreover, since for each n, at most 5 nonzero elements appear in each row, the execution time of an algorithm that fully exploits the sparsity of the matrix is expected to depend linearly on the order of the matrix. For a number of dense programs and varying values of n, a version for a general sparse row-wise matrix and for the matrix having the specific nonzero structure of matrices of the $E(n, 5)$-class are generated. Subsequently, the execution time of each version is measured using the appropriate matrix of the $E(n, 5)$-class. Note that since sparse row-wise versions can also be used for sparse matrices having an arbitrary nonzero structure, probably some performance must be traded for generality.

We can enforce the sparse compiler to select general sparse row-wise storage for an implicitly $n \times n$ sparse matrix A with enveloping data structure A by adding the following annotation to the declaration of this array, which simply states that the region consisting of the whole matrix is sparse and the preferred access direction of this region is $\vec{p} = (0, 1)^T$:

```
      REAL      A(N,N)
C_SPARSE(A : SPARSE()(0,1))
```

Likewise, we can supply the specific nonzero structure of a matrix of the $E(n, 5)$-class to the sparse compiler by replacing the previous annotation with the following annotations, in which the index set of each region is described in terms of a simple section:

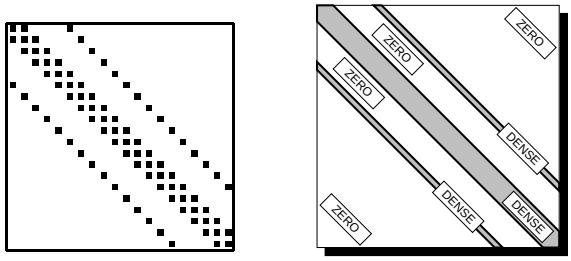


Figure 9.4: Nonzero Structure of $E(20, 5)$

```
C_SPARSE(A : _ZERO (1-N <= I-J <=  -6))
C_SPARSE(A : _DENSE( -5 <= I-J <=  -5))
C_SPARSE(A : _ZERO ( -4 <= I-J <=  -2))
C_SPARSE(A : _DENSE( -1 <= I-J <=   1))
C_SPARSE(A : _ZERO (  2 <= I-J <=   4))
C_SPARSE(A : _DENSE(  5 <= I-J <=   5))
C_SPARSE(A : _ZERO (  6 <= I-J <= N-1))
```

The sparse compiler sets the preferred access direction of all regions to $\vec{p} = (1, 1)^T$, which implies that attempts to enforce regular diagonal-wise access patterns for all occurrences of A will be made. This information can also be obtained automatically by the nonzero structure analyzer if, at compile-time, each specific matrix of the $E(n, 5)$-class is available on file.

9.2.2 Matrix times Vector

Computing the product of a matrix and a vector forms the basic computation of many iterative methods (cf. appendix A). Below, we present a dense implementation of $\vec{b} = A\vec{x}$, where the I-loop is placed innermost to enhance spatial locality or vector performance:

```
DO I = 1, N
  B(I) = 0.0
ENDDO
DO J = 1, N
  DO I = 1, N
    B(I) = B(I) + A(I,J) * X(J)
  ENDDO
ENDDO
```

Depending on whether the annotations enforcing either the selection of general sparse row-wise or static dense storage of nonzero diagonals for the implicitly sparse matrix A with enveloping data structure A are used, the sparse compiler converts the double loop shown above into one of the following fragments, where NP_A=N and SZ_A provides sufficient space for all entries:

Sparse:

```
REAL    VAL_A(1:SZ_A)
INTEGER IND_A(1:SZ_A)
INTEGER LOW_A(1:NP_A)
INTEGER HGH_A(1:NP_A), LST_A
COMMON  /A/ VAL_A, IND_A, ...
...
DO I = 1, N
  DO J_ = LOW_A(I), HGH_A(I)
    J = IND_A(J_)
    B(I) = B(I) + VAL_A(J_) * X(J)
  ENDDO
ENDDO
```

Diagonals:

```
REAL DN1_A(6:N), DN2_A(1:N-5)
REAL DN3_A(1:N,-1:1)
COMMON /A/ DN1_A, DN2_A, DN3_A
...
DO I = 6, N
  B(I) = B(I) + DN2_A(I-5) * X(I-5)
ENDDO
DO J = -1, 1
  DO I = MAX(1, 1-J), MIN(N, N-J)
    B(I) = B(I) + DN3_A(J+I,-J) * X(J+I)
  ENDDO
ENDDO
DO I = 1, N-5
  B(I) = B(I) + DN1_A(I+5) * X(I+5)
ENDDO
```

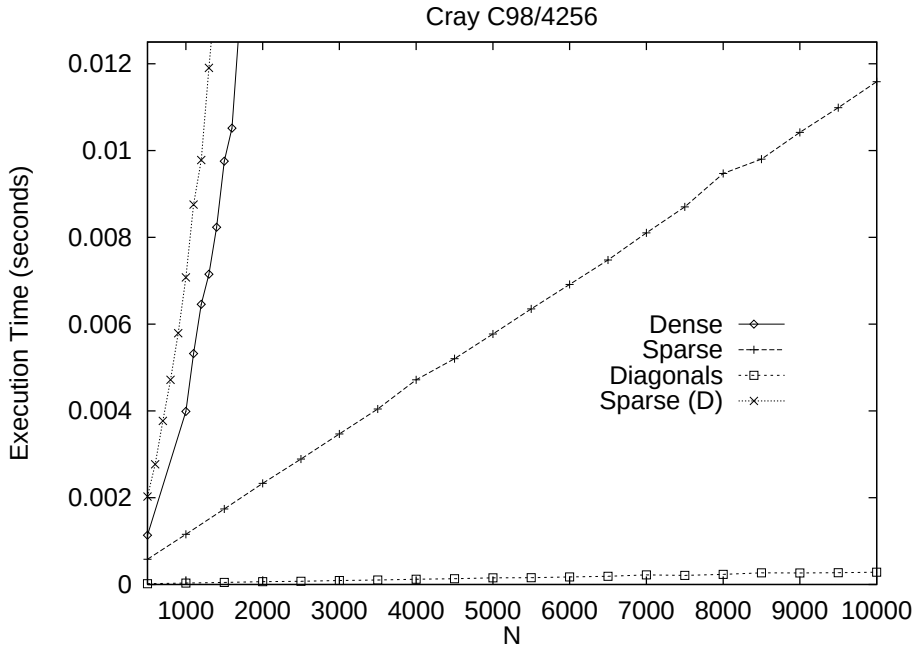


Figure 9.5: Computation of $\vec{b} \leftarrow A\vec{x}$

The first fragment results after loop interchanging has been applied by the reshaping method to enforce row-wise access patterns for the occurrence of the enveloping data structure A. Thereafter, a construct iterating over all entries in each Ith row is generated. The resulting code is equivalent to implementations found in e.g. [169, p248-249][184].

To obtain the second fragment, first the loop transformation defined by the following unimodular matrix is applied to enforce regular diagonal-wise access patterns:

$$U = \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix}$$

Thereafter, iteration space partitioning is used to separate operations on zero diagonals from the operations on nonzero diagonals. Because $a_{ij} \neq 0 \Rightarrow |i-j| \in \{0,1,5\}$, this implies that the execution set $[1-\mathtt{N}, \mathtt{N}-1]$ of the resulting outermost DO-loop is partitioned into the following sets, where DO-loops with a singleton execution set are unrolled:

$$[1-\mathtt{N}, -6], [-5, -5], [-4, -2], [-1, +1], [+2, +4], [+5, +5], [+6, \mathtt{N}-1]$$

Finally, occurrences of A in the resulting loops are either replaced by a zero constant or an appropriate occurrence of the static dense storage that has been selected for the nonzero diagonals, after which redundant assignment statements and DO-loops are eliminated at compile-time.

In figure 9.5, the execution times on the Cray of the original dense fragment, the general sparse code and the diagonal code are shown (labeled Dense, Sparse, and Diagonals respectively). Exploiting the sparsity decreases the execution time (and storage requirements) of the algorithm substantially, which has now become linearly dependent on the order of the sparse matrix. This reduction becomes more profound if the specific characteristics of the nonzero structure of the sparse matrices of the $E(n,5)$-class are exploited. The execution time of the general sparse code applied to a dense matrix is also shown (labeled Sparse(D)). In [2, 84, 180, 181], more advanced implementations of this algorithm are discussed.

9.2.3 Matrix times Matrix

Although the product of two matrices can be computed by repetitively calling a subroutine that computes the product of a matrix and a vector (see section 9.1.3), we can also perform the operation $C \leftarrow C + AB$ directly using the following dense implementation:

```
DO I = 1, N
  DO J = 1, N
    DO K = 1, N
      C(I,J) = C(I,J) + A(I,K) * B(K,J)
    ENDDO
  ENDDO
ENDDO
```

If the annotation enforcing general sparse-row wise storage for the implicitly sparse matrix A with enveloping data structure A is used, the reshaping method yields the following unimodular matrices:

$$U = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \quad \text{or} \quad U = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Hence, the sparse compiler can either interchange the J- and K-loop or use the original fragment to enforce row-wise access patterns for the occurrence of A. Thereafter, one of the following constructs results:

Row-wise1:

```
 DO I = 1, N
   DO K_ = LOW_A(I), HGH_A(I)
     K = IND_A(K_)
     DO J = 1, N
       C(I,J) = C(I,J) +
+           VAL_A(K_) * B(K,J)
     ENDDO
   ENDDO
 ENDDO
```

Row-wise2:

```
 DO I = 1, N
   DO J = 1, N
     DO K_ = LOW_A(I), HGH_A(I)
       K = IND_A(K_)
       C(I,J) = C(I,J) +
+           VAL_A(K_) * B(K,J)
     ENDDO
   ENDDO
ENDDO
```

If the specific nonzero structure of $E(n, 5)$ is exploited, the reshaping method is used to enforce regular diagonal-wise access patterns, which gives rise to the construction of the following unimodular matrices:

$$U = \begin{pmatrix} 0 & 1 & 0 \\ -1 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix} \quad \text{or} \quad U = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

Although scalar-wise *true* access patterns result for the occurrence of A after application of the loop transformation defined by the second matrix, the *effective* access patterns of this occurrence remain row-wise. Therefore, the loop transformation defined by the first matrix is applied. Thereafter, iteration space partitioning is used to separate operations on zero elements from operations on entries. Finally, redundant assignment statements and DO-loops are eliminated at compile-time and static dense storage is selected:

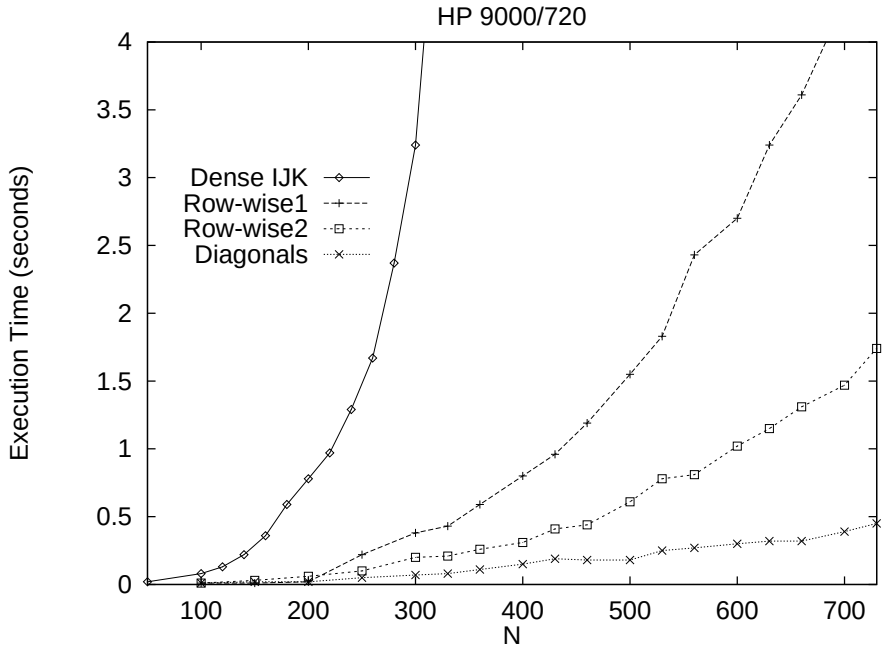


Figure 9.6: Computation of $C \leftarrow C + AB$

```
REAL       DN1_A(6:N), DN2_A(1:N-5), DN3_A(1:N,-1:1)
COMMON /A/ DN1_A,      DN2_A,        DN3_A
...
DO I = 1, N
  DO K = 1, N-5
    C(K+5,I) = C(K+5,I) + DN2_A(K) * B(K,I)
  ENDDO
  DO J = -1, 1
    DO K = MAX(1, J+1), MIN(N, J+N)
      C(K-J,I) = C(K-J,I) + DN3_A(K,-J) * B(K,I)
    ENDDO
  ENDDO
  DO K = 6, N
    C(K-5,I) = C(K-5,I) + DN1_A(K) * B(K,I)
  ENDDO
ENDDO
```

In figure 9.6 and 9.7 the execution times on both the HP and the Cray of these fragments are shown, where the dense version with the smallest execution time is used on both machines (version `IJK` and `KJI` on the HP and the Cray respectively). Again, exploiting sparsity reduces the execution time substantially, although the relative performance of the two general row-wise sparse versions differs on both machines. Exploiting all characteristics of the nonzero structure, however, yields the code with the least execution time on both machines.

Now, suppose that the arrays A, B, and C are used as enveloping data structure of three implicitly sparse matrices A, B, and C. In this case, the reshaping method of the sparse compiler enables us to explore all possible general sparse storage schemes. For example, we can explore whether the reshaping method can enforce row-wise access patterns for the occurrences of the arrays A and B, and column-wise access patterns for the occurrences of C using the following annotations:

```
      PARAMETER (N=...)
      REAL      A(N,N), B(N,N), C(N,N)
C_SPARSE( A : _SPARSE()(0,1) ; B : _SPARSE()(0,1) ; C : _SPARSE()(1,0) )
```

In figure 9.8, a tile is placed at every combination of row-, column-, and regular diagonal-wise access patterns that can be enforced for the occurrences of A, B, and C respectively using the reshaping method (assuming that data dependences caused by the accumulation may be ignored).

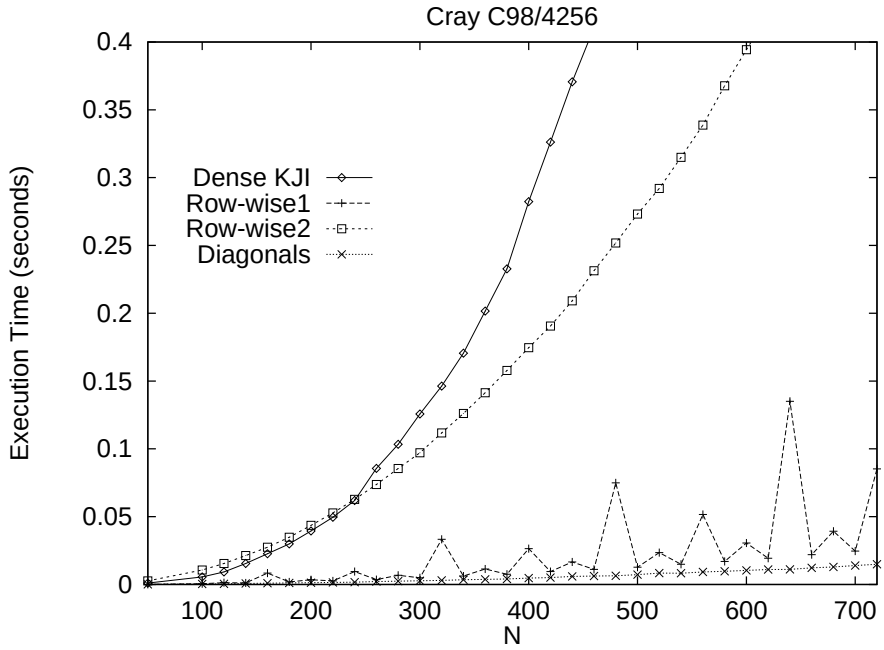


Figure 9.7: Computation of $C \leftarrow C + AB$

Not surprisingly, row-wise access patterns for the occurrences of A and C and column-wise access patterns for B are enforced by a loop transformation defined by $U = I$. Because the sparse storage schemes of the implicitly sparse matrices are selected accordingly, the following sparse code is generated, in which each Ith row of C and Jth column of B is expanded before operated upon:

```
DO I = 1, N
  CALL SSCT__(VAL_C ,IND_C, LOW_C(I), HGH_C(I), SAP_10, SWT_10)
  DO J = 1, N
    CALL SSCT__(VAL_B, IND_B, LOW_B(J), HGH_B(J), SAP_20, SWT_20)
    DO K_ = LOW_A(I), HGH_A(I)
      K = IND_A(K_)
      IF (SWT_20(K)) THEN
        IF (.NOT.(SWT_10(J))) THEN
          SWT_10(J) = .TRUE.
          CALL SINS__(VAL_C, IND_C, LOW_C, HGH_C, I, N, SZ_A, LST_C, L_, J)
        END IF
        SAP_10(J) = SAP_10(J) + VAL_A(K_) * SAP_20(K)
      END IF
    ENDDO
    CALL SGTH__(VAL_B, IND_B, LOW_B(J), HGH_B(J), SAP_20, SWT_20)
  ENDDO
  CALL SGTH__(VAL_C ,IND_C, LOW_C(I), HGH_C(I), SAP_10, SWT_10)
ENDDO
```

After the K-loop has been involved in guard encapsulation of '$(\mathtt{I}, \mathtt{K}) \in E(A)$', encapsulation of '$(\mathtt{K}, \mathtt{J}) \in E(B)$' becomes disabled. Hence, in this fragment only the sparsity of A is exploited to reduce the computational time.

As another example, if we enforce the selection of general sparse row-wise storage for A and B and general sparse column-wise storage for C, then the reshaping method fails because $\text{rank}(S) = 3$ for the objective matrix, as implied by the following integer echelon reduction:

$$E = RS^T = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

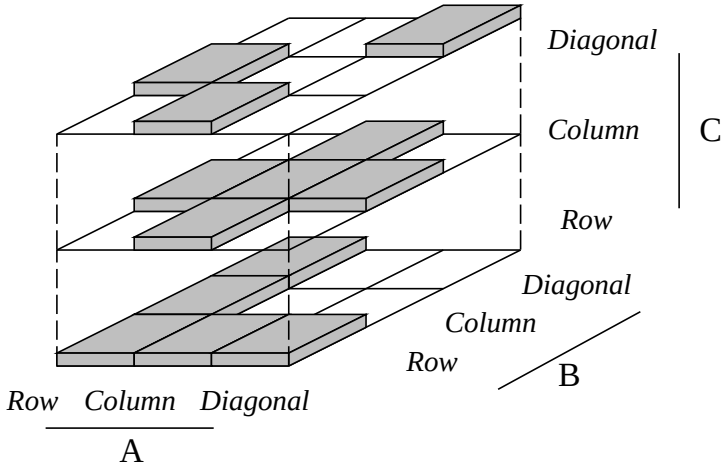


Figure 9.8: Reshaping Matrix Multiplication

Therefore, no tile appears at the corresponding row-row-column entry. If we inform the sparse compiler that occurrences of `C` may be ignored during reshaping, then the `J`- and `K`-loop are interchanged, and the following sparse code is generated:

```
DO I = 1, N
  DO K_ = LOW_A(I), HGH_A(I)
    K = IND_A(K_)
    DO J_ = LOW_B(K), HGH_B(K)
      J = IND_B(J_)
      L_ = LKP__(IND_C, LOW_C(J), HGH_C(J), I)
      IF ((L_.EQ.1)) THEN
        CALL SINS__(VAL_C, IND_C, LOW_C, HGH_C, J, N, SZ_C, LST_C, L_, I)
      END IF
      VAL_C(L_) = VAL_C(L_) + VAL_A(K_) * VAL_B(J_)
    ENDDO
  ENDDO
ENDDO
```

Because the true access patterns of the occurrences of `C` are row-wise, all overhead reducing techniques become disabled.

As a final example, we enforce the selection of general sparse row-wise storage for all implicitly sparse matrices. After access pattern reshaping, the following sparse code is generated, in which the sparsity of both B and C is fully exploited, and the `I`th row of C is expanded before operated upon:

```
DO I = 1, N
  CALL SSCT__(VAL_C, IND_C, LOW_C(I), HGH_C(I), SAP_10, SWT_10)
  DO K_ = LOW_A(I), HGH_A(I)
    K = IND_A(K_)
    DO J_ = LOW_B(K), HGH_B(K)
      J = IND_B(K_)
      IF (.NOT.(SWT_10(J))) THEN
        SWT_10(J) = .TRUE.
        CALL SINS__(VAL_C, IND_C, LOW_C, HGH_C, I, N, SZ_C, LST_C, L_, J)
      END IF
      SAP_10(J) = SAP_10(J) + VAL_A(K_) * VAL_B(J_)
    ENDDO
  ENDDO
  CALL SGTH__(VAL_C, IND_C, LOW_C(I), HGH_C(I), SAP_10, SWT_10)
ENDDO
```

In essence, this automatically generated sparse implementation is equivalent to the code for sparse matrix multiplication found in [106][169, p253-258], although in the latter code symbolic and numerical operations are separated. Another implementation is given in [154].

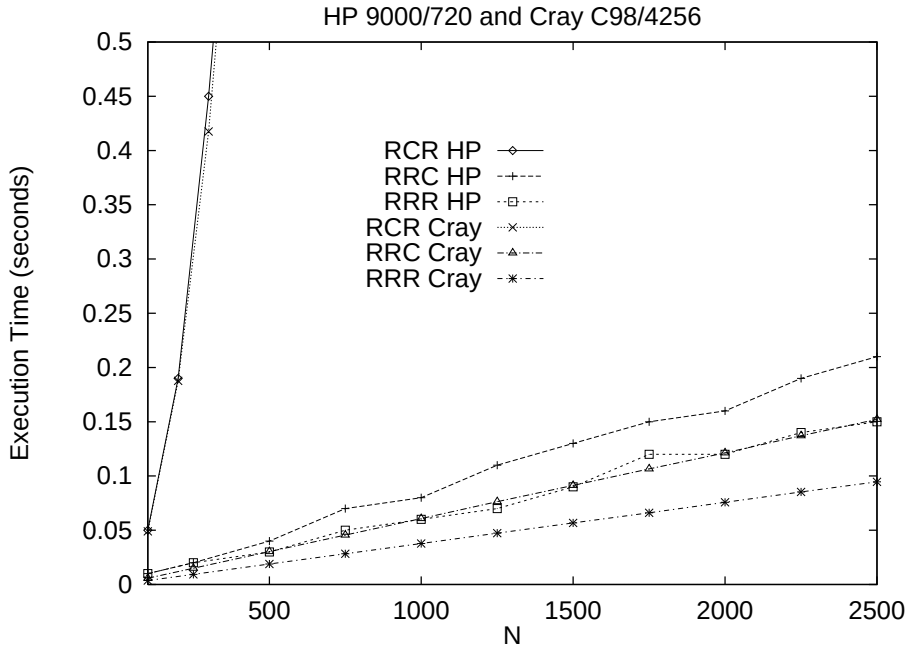


Figure 9.9: Computation of $C \leftarrow C + AB$

In figure 9.9, we present the execution times of the automatically generated versions on both the Cray and HP (labeled according to the kind of storage scheme selected for A, B, and C respectively, where e.g. RRR denotes general sparse row-wise storage of all matrices). The general sparse storage schemes of A and B are initialized to $E(n, 5)$ and we start with $C = 0$. Obviously, although the storage requirements of the first version are reduced, the computational time of this version is still unacceptable. The other fragments, however, fully exploit the sparsity of A and B to reduce the computational time, whereas the sparsity of all matrices is exploited to reduce the storage requirements. This experiment clearly illustrates the importance of selecting appropriate sparse storage schemes.

9.2.4 LU-Factorization

An important step in solving a linear system of equations $A\vec{x} = \vec{b}$ is the factorization of a square non-singular matrix A into a unit lower triangular matrix L and an upper triangular matrix U according to $A = LU$ (cf. appendix A). A dense implementation of LU-factorization without pivoting is shown below, where the array A that is initially used to store A becomes overwritten with the elements of the factors L and U:

```
DO K = 1, N-1
  DO I = K+1, N
    A1(I,K) = A2(I,K) / A3(K,K)
    DO J = K+1, N
      A4(I,J) = A5(I,J) - A6(I,K) * A7(K,J)
    ENDDO
  ENDDO
ENDDO
```

Because matrices of the $E(n, 5)$-class are positive definite, factorization without pivoting is stable. A straightforward way to exploit the sparsity of the matrix to reduce the computational time of the algorithm is to guard the loop-body of the `I`-loop with the test '`(A(I,K).NE.0.0)`'. However, in this manner, the storage requirements of the algorithm are not reduced.

If we use an annotation to enforce the selection of general sparse row-wise storage for the implicitly sparse matrix A with enveloping data structure A, then the reshaping method fails, and the following sparse code is generated:[2]

```
DO K = 1, N
  DO I = K+1, N
    CALL SSCT__(VAL_A, IND_A, LOW_A(I), HGH_A(I), SAP_20, SWT_20)
    IF (SWT_20(K)) THEN
      SAP_20(K) = SAP_20(K) / VAL_A(LKP__(IND_A, LOW_A(K), HGH_A(K), K))
      LEN_J = HGH_A(K) - LOW_A(K)
      DO J_ = 0, LEN_J
        J = IND_A(LOW_A(K) + J_)
        IF (K+1.LE.J) THEN
          IF (.NOT.SWT_20(J)) THEN
           SWT_20(J) = .TRUE.
           CALL SINS__(VAL_A, IND_A, LOW_A, HGH_A, I, NP_A, SZ_A, LST_A, L_, J)
          END IF
          SAP_20(J) = SAP_20(J) - SAP_20(K) * VAL_A(LOW_A(K) + J_)
        END IF
      ENDDO
    ENDIF
    CALL SGTH__(VAL_A, IND_A, LOW_A(I), HGH_A(I), SAP_20, SWT_20)
  ENDDO
ENDDO
```

However, if the programmer indicates that the access patterns of occurrence A(K,K) may be ignored during the reshaping, the I- and J-loop are interchanged to obtain row-wise access patterns for the three occurrences A(I,K). Thereafter, the following sparse code is generated:

```
DO I = 2, N
  CALL SSCT__(VAL_A1, IND_A, LOW_A(I), HGH_A(I), SAP_10, SWT_10)
  DO K = 1, I-1
    IF (SWT_10(K)) THEN
      SAP_10(K) = SAP_10(K) / VAL_A(LKP__(IND_A, LOW_A(K), HGH_A(K), K))
      LEN_J = HGH_A(K) - LOW_A(K)
      DO J_ = 0, LEN_J
        J = IND_A(LOW_A(K)+J_)
        IF (K+1.LE.J) THEN
          IF (.NOT.SWT_10(J)) THEN
            SWT_10(J) = .TRUE.
            CALL SINS__(VAL_A, IND_A, LOW_A, HGH_A, I, N, SZ_A, LST_A, L_, J)
          END IF
          SAP_10(J) = SAP_10(J) - SAP_10(K) * VAL_A(LOW_A(K+J_))
        END IF
      ENDDO
    END IF
  ENDDO
  CALL SGTH__(VAL_A, IND_A, LOW_A(I), HGH_A(I), SAP_10, SWT_10)
ENDDO
```

Guard '$(\mathtt{K},\mathtt{J}) \in E(A)$' has been encapsulated in the execution set of the J-loop because during each fixed iteration $\mathtt{I} = i$ and $\mathtt{K} = k$, only insertions in the jth row, where $i < k < j$ may occur. The entries are accessed relatively to the base location LOW_A(K) to correctly account for any data movement that may occur during insertions in other rows. Moreover, since all entries in a row of the sparse matrix are stored in a single sparse vector, the test '(K+1.LE.J)' is required to determine whether an entry must actually be operated upon. The possible insertions disable encapsulation of guard '$(\mathtt{I},\mathtt{K}) \in E(A)$' in the execution set of the K-loop, because we would like to iterate over all entries in a row in which insertions are performed (which could only be implemented if some kind of ordering would be imposed on the entries in each row).

Because fill-in occurs, the nonzero structure of the sparse matrix changes into the nonzero structure of the filled matrix, i.e. $L + U$.

[2] Here, we have manually moved to the scatter operation before the first assignment statement to improve the performance. Such optimizations can be easily incorporated in a future implementation.

In figure 9.10, the nonzero structure of the matrix $E(20,5)$ and the nonzero structure of the filled matrix arising after the factorization are shown. Therefore, rather than selecting general sparse row-wise storage, we can also inform the compiler about the fact that the implicitly sparse matrix A eventually becomes a band matrix with semi-bandwidths 5 using the following annotations (note that the zero regions outside the band will be preserved at run-time):

```
      PARAMETER (N = ...)
      REAL      A(N,N)
C_SPARSE(A : _ZERO (1-N <= I-J <=  -6))
C_SPARSE(A : _DENSE( -5 <= I-J <=   5))
C_SPARSE(A : _ZERO (  6 <= I-J <= N-1))
```

In this case, the sparse compiler automatically converts the original implementation of LU-factorization into the following band formulation of LU-factorization using iteration space partitioning and the compile-time elimination of redundant assignment statements and DO-loops:

```
REAL       DN1_A(1:N,-5:5)
COMMON /A/ DN1_A
...
DO K = 1, N-1
  DO I = K+1, MIN(N, K+5)
    DN1_A(K,I-K) = DN1_A(K,I-K) / DN1_A(K,0)
    DO J = K+1, MIN(N, K+5)
      DN1_A(J,I-J) = DN1_A(J,I-J) - DN1_A(K,I-K) * DN1_A(J,K-J)
    ENDDO
  ENDDO
ENDDO
```

In figure 9.11, we present the execution time of these versions of LU-factorization (Dense1 and Dense2 denote the original dense implementation and the dense implementation with a conditional statement respectively). Although the band implementation is clearly superior, the second general sparse version also outperforms the dense version that exploits the sparsity to reduce computational time. Because the size of the execution set of the K-loop has not been reduced using guard encapsulation, however, the execution time of general sparse row-wise version still grows at least quadratically in the order of the sparse matrix for larger matrices. See figure 9.13. This makes solving large sparse systems infeasible. Although we can use the band implementation for all matrices of the $E(n,5)$-class, we would also like to be able to generate a general sparse implementation that really exploits all zero elements to deal with other kinds of sparse matrices.

One step in the right direction is the observation that in the original fragment the strict lower triangular part is mainly accessed along columns, whereas the strict upper triangular part is only accessed along rows. Moreover, because the elements along the main diagonal are used as pivots, these elements must be nonzero. Hence, we can help the sparse compiler by supplying this information by means of the following annotations:

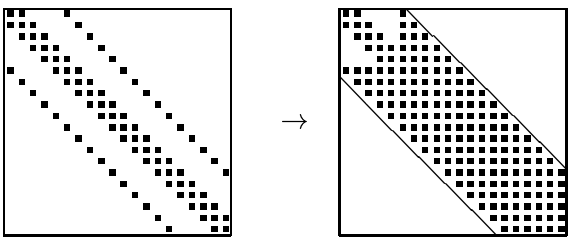


Figure 9.10: Nonzero Structure of $E(20,5)$ and the Filled Matrix

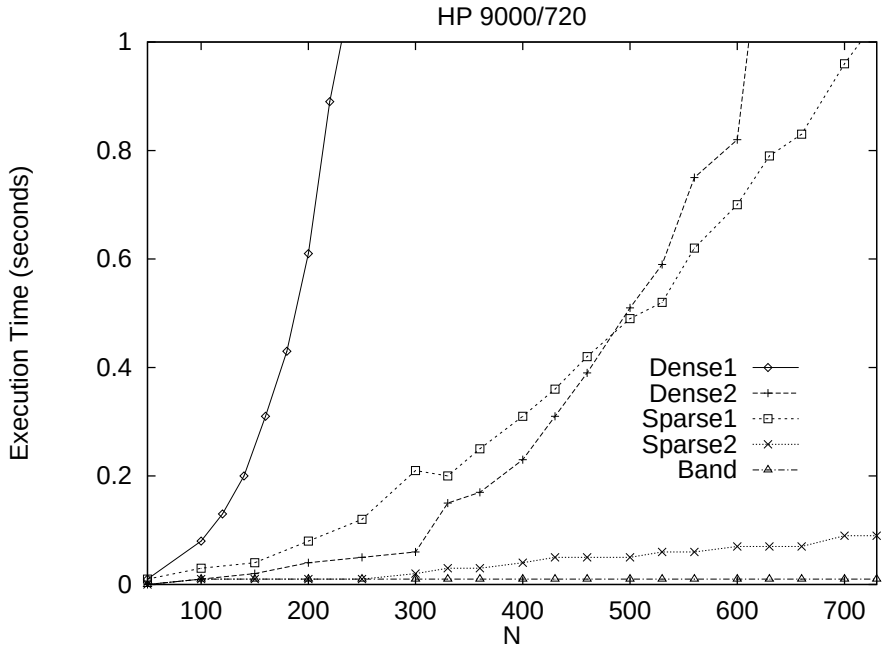


Figure 9.11: LU-Factorization

```
      PARAMETER (N = ...)
      REAL      A(N,N)
C_SPARSE(A : _SPARSE(1-N <= I-J <=  -1)(0,1))
C_SPARSE(A : _DENSE (  0 <= I-J <=   0)(1,1))
C_SPARSE(A : _SPARSE(  1 <= I-J <= N-1)(1,0))
```

First, iteration space partitioning is applied by the sparse compiler to isolate operations on the strict lower and strict upper triangular part and the main diagonal:

```
...                                          ...
DO J = K+1, N                                DO J = K+1, I-1
  A(I,J) = A(I,J) - A(I,K)*A(K,J)    I=J       A(I,J) = A(I,J) - A(I,K)*A(K,J)
ENDDO                                        ENDDO
...                                          A(I,I) = A(I,I) - A(I,K)*A(K,I)
                                             DO J = I+1, N
                                               A(I,J) = A(I,J) - A(I,K)*A(K,J)
                                             ENDDO
                                             ...
```

If the simple section associated with the occurrence with label i in the original fragment is denoted by $S_i \subseteq \mathcal{Z}^2$, then the effects of this iteration space partitioning are shown in figure 9.12. Note that independent of whether the iteration space partitioning is induced by the occurrence with label 4 or 5, eventually the incrementally constructed simple sections into which both S_4 and S_5 are partitioned become associated with the resulting duplicates of these occurrences according to the mechanism discussed in detail in section 7.2.3. Moreover, the redundant fragmentation of the other simple sections is simply ignored. In fact, even without annotations, the sparse compiler is able to detect the fact that separate storage of the strict triangular parts and the main diagonal of the matrix matches the kind of operations performed in the code [39]. Indeed, many general storage schemes for sparse matrices are based on this fragmentation [164, 185, 235, 236].

Thereafter, loop distribution is applied to the `I`-loop, after which loop interchanging is applied to obtain column-wise access of the strict lower triangular part:[3]

[3] In the current prototype sparse compiler, these particular transformations must be guided by the programmer. In principle, however, these transformations could be done automatically in a future implementation.

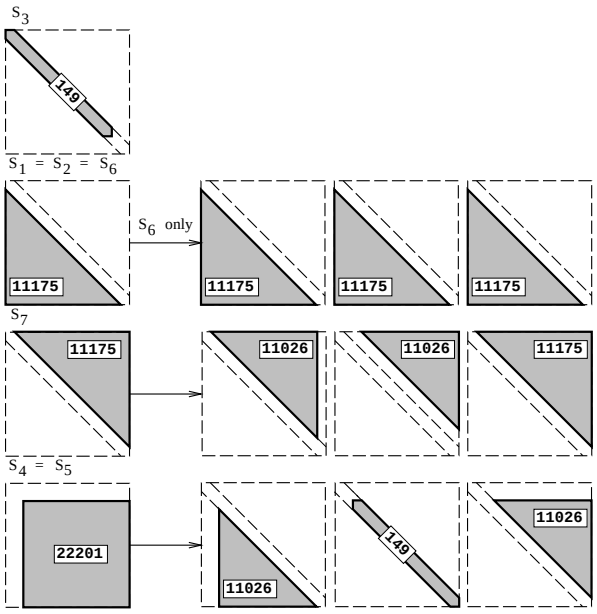


Figure 9.12: Fragmentation of LU-Factorization (`N=150`)

```
DO K = 1, N-1
  DO I = K+1, N
    A(I,K) = A(I,K) / A(K,K)
  ENDDO
  DO I = K+1, N                              DO J = K+1, N-1
    DO J = K+1, I-1                    →       DO I = J+1, N
      A(I,J) = A(I,J) - A(I,K) * A(K,J)
    ENDDO
  ENDDO
  DO I = K+1, N
    A(I,I) = A(I,I) - A(I,K) * A(K,I)
  ENDDO
  DO I = K+1, N
    DO J = I+1, N
      A(I,J) = A(I,J) - A(I,K) * A(K,J)
    ENDDO
  ENDDO
ENDDO
```

Finally, the compiler selects a sparse storage scheme in which the entries in the strict lower and upper triangular part of A are stored in separate sparse vectors, whereas static dense storage is used for the main diagonal of A. We refer to this storage scheme as the LDU-scheme (see also section 8.3). First, guard encapsulation to the first `I`-loop is done as shown below, because all entries in column `K` below the main diagonal are stored in the `K+N-1`th sparse vector of the pool:

```
DO K = 1, N-1
  DO I_ = LOW_A(K+N-1), HGH_A(K+N-1)
    I = IND_A(I_)
    VAL_A(I_) = VAL_A(I_) / DN1_A(K)
  ENDDO
  ...
```

Subsequently, the double loop performing the updates on the strict lower triangular part of A is converted into a construct that iterates over entries in the strict upper triangular part of the `K`th row (stored in the `K`th sparse vector) and entries in the strict lower triangular part of the `K`th column (stored in the `N-K+1`th sparse vector). In fact, we have altered the upper bound of the `J`-loop into `N` to reduce the number of resulting tests.

The sparse compiler generates code in which relative addressing is used to account for possible data movement. The J**th** column below the main diagonal is expanded before operated upon:

```
LEN_J = HGH_A(K) - LOW_A(K)
DO J_ = 0, LEN_J
  J = IND_A(LOW_A(K)+J_)
  CALL SSCT__(VAL_A, IND_A, LOW_A(J+N-1), HGH_A(J+N-1), SAP_20, SWT_20)
  IF (J+1.LE.100) THEN
    LEN_I = HGH_A(K+N-1) - LOW_A(K+N-1)
    DO I_ = 0, LEN_I, 1
      I = IND_A((LOW_A(K+N-1)+I_))
      IF (J+1.LE.I) THEN
        IF (.NOT.SWT_20(I)) THEN
          SWT_20(I) = .TRUE.
          CALL SINS__(VAL_A, IND_A, LOW_A, HGH_A, J+N-1, N, SZ_A, LST_A, L_, I)
        ENDIF
        SAP_20(I) = SAP_20(I) - VAL_A((LOW_A(K+N-1)+I_)) * VAL_A((LOW_A(K)+J_))
      ENDIF
    ENDDO
  ENDIF
  CALL SGTH__(VAL_A, IND_A, LOW_A(J+N-1), HGH_A(J+N-1), SAP_20, SWT_20)
ENDDO
```

Updating the elements along the main diagonal is implemented as follows:

```
CALL SSCT__(VAL_A, IND_A, LOW_A(K), HGH_A(K), SAP_10, SWT_10)
DO I_ = LOW_A(K+N-1), HGH_A(K+N-1)
  I = IND_A(I_)
  IF (SWT_10(I)) THEN
    DN1_A(I) = DN1_A(I) - VAL_A(I_) * SAP_10(I)
  ENDIF
ENDDO
CALL SGTH__(VAL_A, IND_A, LOW_A(K), HGH_A(K), SAP_10, SWT_10)
```

As partly illustrated below, the sparse code generated for the double loop that performs the updating of elements in the strict upper triangular part is very similar to the code that updates the strict lower triangular part:

```
LEN_I = HGH_A(K+N-1) - LOW_A(K+N-1)
DO I_ = 0, LEN_I
  I = IND_A(LOW_A(K)+N-1+I_)
  ...
    ...
  ...
ENDDO
ENDDO    ! K-loop
```

In figure 9.13, the execution time of the implementation operating upon the LDU-scheme is compared with the execution time of the general sparse row-wise and band version of LU-factorization. Although the LDU sparse version is slightly more expensive than the band version, these experiments indicate that a general sparse implementation that fully exploits the sparsity of A can be derived. In figure 9.14 and 9.15, the same experiments are done on the Cray (note that the dense versions run significantly faster than on the HP).

Another way to obtain an implementation of LU-factorization that fully exploits the sparsity of the matrix operated upon is to store the column nonzero structure of the matrix in combination with general sparse row-wise storage [105][235, ch2][236, 164]. Obviously, the LDU-scheme suffers from less overhead storage. Moreover, because matrices of the $E(n, 5)$-class are *symmetric* positive definite, we would rather like to use the Choleski factorization to solve a corresponding system of linear equations. Experiments with the sparse compiler indicate that a row- or column-oriented formulation of Choleski factorization (see e.g. [129, p201]) is directly converted into a version exploiting all zero elements. A symmetric implementation in which symbolic and numerical operations are separated can be found in [169, p258-268].

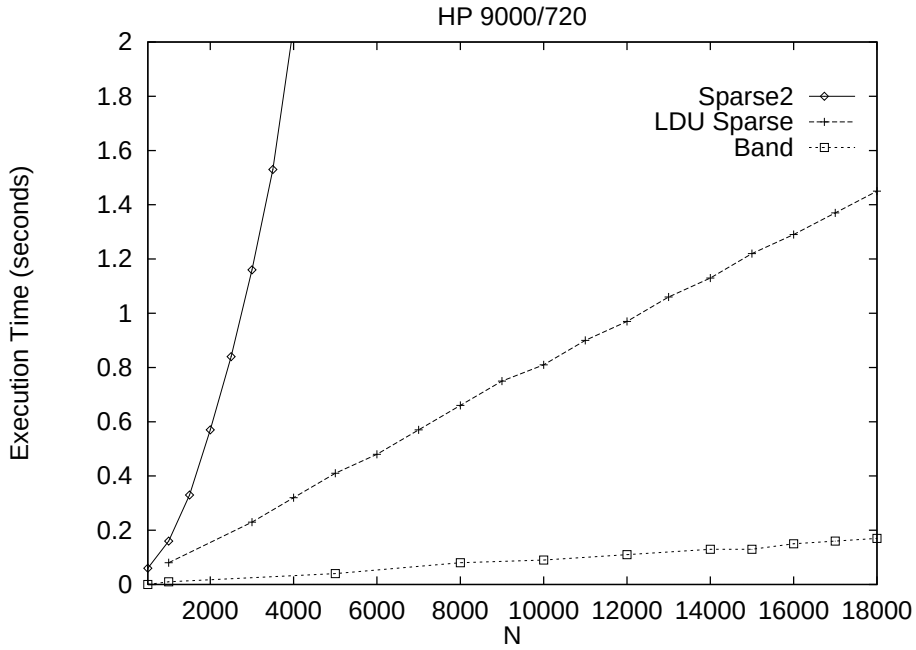


Figure 9.13: LU-Factorization

Other experiments presented in [35] indicate that code performing the factorization of a tridiagonal matrix that can be derived automatically is essentially identical to a hand-coded implementation [149]. Some remarks and implementation details about parallel direct solvers can be found in e.g. [129][168][235, ch10].

9.2.5 Forward and Back Substitution

After a matrix A has been factorized into $A = LU$, a system $A\vec{x} = \vec{b}$ is solved by forward substitution of the system $L\vec{c} = \vec{b}$, followed by back substitution of $U\vec{x} = \vec{c}$. Dense implementations of forward and back substitution, where an in-place conversion of the vector $\vec{b}$ into $\vec{x}$ is performed, are shown below:

Forward Substitution:

```
DO I = 2, N
  DO J = 1, I-1
    B(I) = B(I) - A(I,J) * B(J)
  ENDDO
ENDDO
```

Back Substitution:

```
DO I = N, 1, -1
  DO J = I+1, N
    B(I) = B(I) - A(I,J) * B(J)
  ENDDO
  B(I) = B(I) / A(I,I)
ENDDO
```

If an annotation enforcing general sparse row-wise storage of A is used, the sparse compiler converts these fragments into the following sparse codes:

Sparse Forward:

```
DO I = 2, N
  DO J_ = LOW_A(I), HGH_A(I)
    J = IND_A(J_)
    IF (J.LE.I-1) THEN
      B(I) = B(I) - VAL_A(J_) * B(J)
    END IF
  ENDDO
ENDDO
```

Sparse Back:

```
 DO I = N, 1, -1
   IF (I+1.LE.N) THEN
     DO J_ = LOW_A(I), HGH_A(I)
       J = IND_A(J_)
       IF (I+1.LE.J) THEN
         B(I) = B(I) - VAL_A(J_) * B(J)
       END IF
     ENDDO
   ENDIF
   B(I) = B(I) / VAL_A( LKP__(IND_A,
+              LOW_A(I), HGH_A(I), I) )
 ENDDO
```

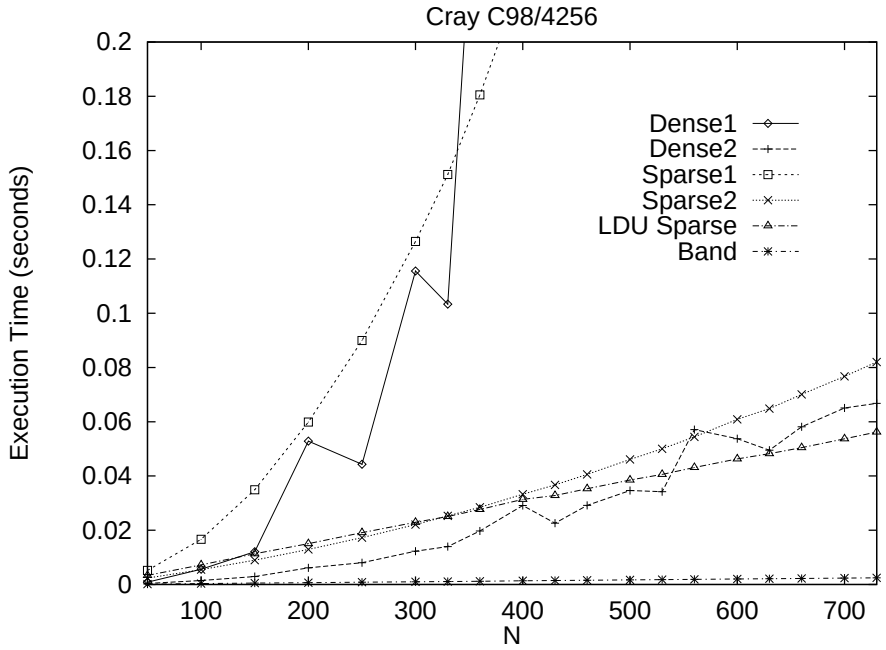


Figure 9.14: LU-Factorization

Note that although the resulting fragments strongly resemble the code generated for the product of a sparse matrix with a vector, there are some differences. First, a lookup is required in the back substitution, because the true diagonal-wise access patterns of `A(I,I)` are inconsistent with the way in which the entries of A are stored. Moreover, because the execution set of the `J`-loop is empty for `I=N`, the generated `J_`-loop is protected by the test '`(I+1.LE.N)`' to prevent erroneous accesses to entries in the `N`th row of the sparse matrix (although the test could be safely omitted for this particular example).[4] Finally, because *all* entries in a row are stored in a single sparse vector, the test '`(I+1.LE.J)`' remains required in the innermost DO-loop of both versions to distinguish between entries in the strict lower and upper triangular part of the matrix respectively. The fragments still exploit the sparsity of A, however, because in contrast with using the test '`(A(I,J).NE.0.0)`' in the dense case, the test in the sparse versions is only executed for entries.

If annotations enforcing the selection of the LDU-scheme are used, the sparse compiler applies loop interchanging to the double loop of forward substitution, and generates the following sparse codes:

LDU Forward:

```
DO J = 1, N-1
  DO I_ = LOW_A(J+N-1), HGH_A(J+N-1)
    I = IND_A(I_)
    B(I) = B(I) - VAL_A(I_) * B(J)
  ENDDO
ENDDO
```

LDU Back:

```
DO I = N, 1, -1
  IF (I+1.LT.N) THEN
    DO J_ = LOW_A(I), HGH_A(I)
      J = IND_A(J_)
      B(I) = B(I) - VAL_A(J_) * B(J)
    ENDDO
  ENDIF
  B(I) = B(I) / DN1_A(I)
ENDDO
```

Loop interchanging has, in fact, converted the inner product formulation of forward substitution into an outer product formulation [97, p25-28].

[4] The generation of this test can be avoided by peeling one iteration of the `I`-loop.

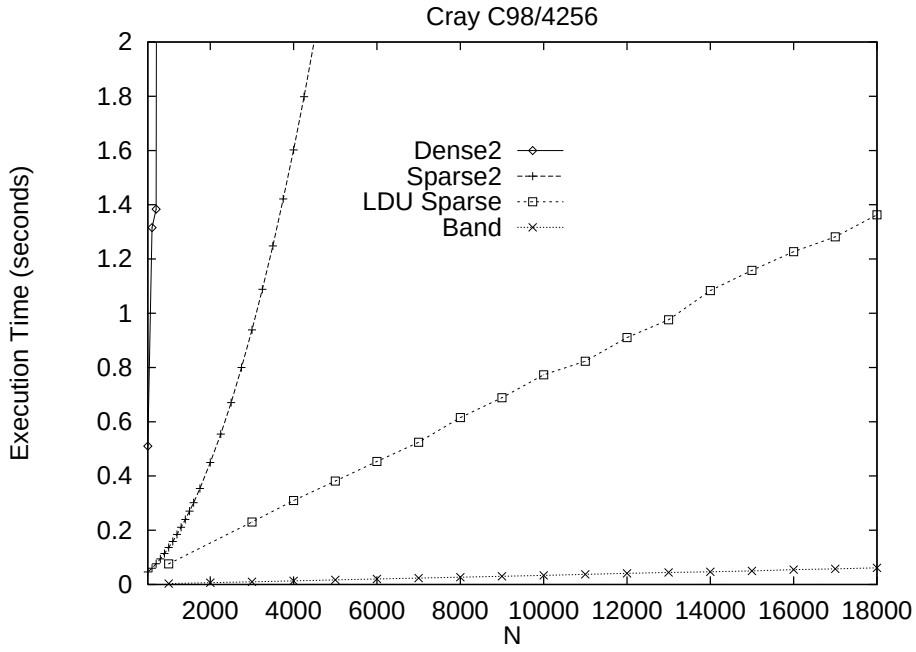


Figure 9.15: LU-Factorization

Now, no test is required in the body of the `J_`-loops because entries in the strict lower and strict upper triangular part of A are stored in separate sparse vectors. Moreover, because static dense storage is used for the main diagonal of A, the lookup in back substitution vanishes. However, in this case it is essential to protect the whole `J_`-loop of back substitution with the test '`(I+1.LE.N)`', because otherwise the `N`th sparse vector would be erroneously accessed, thereby inducing accesses to the first column of A

Finally, if we inform the sparse compiler about the specific band characteristics of matrices of the $E(n, 5)$-class after the factorization, the following band versions are generated automatically:

Band Forward:

```
DO I = 2, N
  DO J = MAX(1, I-5), I-1
    B(I) = B(I) - DN1_A(J,I-J) * B(J)
  ENDDO
ENDDO
```

Band Back:

```
DO I = N, 1, -1
  DO J = I+1, MIN(N, I+5)
    B(I) = B(I) - DN1_A(J,I-J) * B(J)
  ENDDO
  B(I) = B(I) / DN1_A(I,0)
ENDDO
```

Due to data dependences, the access patterns cannot be reshaped along the diagonals. However, iteration space partitioning and the compile-time elimination of redundant assignment statements and DO-loops is still applicable to the innermost DO-loop, which also reduces the amount of operations that must be executed. Similar implementations of forward and back substitution can found in SPARSKIT and SPARK [185, 186] and in [11][169, p268-270][184].

In figure 9.16, the execution times of all versions of forward and back substitution on the Cray are presented (dense forward and back substitution have almost identical execution times). Because for general sparse row-wise storage a lookup remains in the code of back substitution, this version has the largest execution time of all sparse versions. Nevertheless, it is clear that all sparse versions fully exploit sparsity.

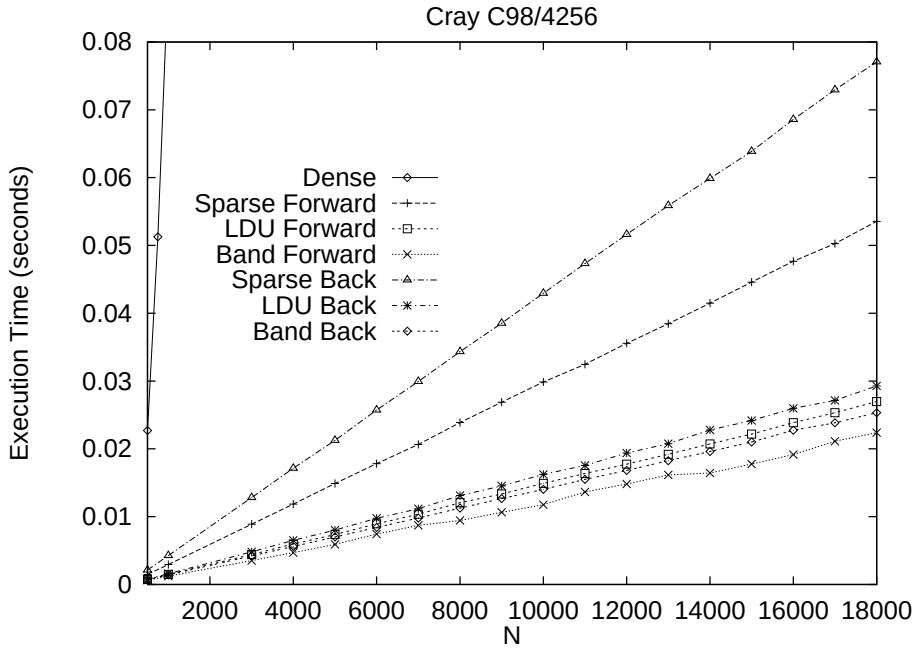


Figure 9.16: Forward and Back Substitution

9.2.6 A Non-Numerical Application

A convenient data structure to represent a weighted directed graph $G = (V, E)$, consisting of a finite set of vertices $V = \{v_1, \ldots, v_n\}$ that are labeled with the integers $1, \ldots, n$, a finite set of edges $E \subseteq V \times V$ and a mapping $f : E \to \mathcal{N}^+$ that assigns a weight to every edge, is a special kind of adjacency matrix, called a **weight matrix**. If v_i denotes the vertex with label i, then in this matrix W we set $w_{ij} = c$, if $(v_i, v_j) \in E$ and $f((v_i, v_j)) = c$, or $w_{ij} = 0$ otherwise. The weighted graph shown in figure 9.17, for instance, is represented by the given weight matrix:

$$W = \begin{pmatrix} 0 & 0 & 0 & 0 & 9 & 11 \\ 0 & 0 & 0 & 8 & 14 & 0 \\ 15 & 7 & 0 & 10 & 0 & 0 \\ 0 & 0 & 0 & 0 & 35 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 9 & 0 \end{pmatrix}$$

Although much storage is wasted because W is usually sparse, the advantage of this representation is that it can be easily manipulated.

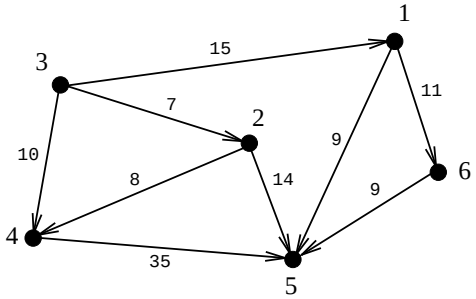


Figure 9.17: Weighted Graph

For example, below we present a straightforward implementation of finding a topological sort of a directed *acyclic* graph, which is a mapping $t : V \to \{1, \ldots, n\}$, such that $\forall (v_i, v_j) \in E : t(v_i) < t(v_j)$.

In the implementation, first the in-degree of every vertex is determined in array INDEG. Thereafter, the topological sort is computed in array TOP by using a stack to successively consider each next vertex with zero in-degree and adapting the in-degree of all neighbors:

```
PROGRAM TOPSORT
INTEGER   N, I, J, TOP
PARAMETER (N = ...)
INTEGER   W(N,N), INDEG(N), TOP(N)
INTEGER   STACK(N), SP

DO J = 1, N
  INDEG(J) = 0
  DO I = 1, N
    IF (W(I,J).NE.0) THEN
      INDEG(J) = INDEG(J) + 1
    ENDIF
  ENDDO
ENDDO
SP = 0
DO I = 1, N
  IF (INDEG(I).EQ.0) THEN
    CALL PUSH(STACK, SP, I)
  ENDIF
ENDDO
TP = 1
CALL POP(STACK, SP, I)
DO WHILE (I.GT.0)
  TOP(I) = TP
  TP     = TP + 1
  DO J = 1, N
    IF (W(I,J).NE.0) THEN
      INDEG(J) = INDEG(J) - 1
      IF (INDEG(J).EQ.0) THEN
        CALL PUSH(STACK, SP, J)
      ENDIF
    ENDIF
  ENDDO
  CALL POP(STACK, SP, I)
ENDDO

END
```

```
SUBROUTINE PUSH(STACK, SP, I)
INTEGER STACK(*), SP, I

SP        = SP + 1
STACK(SP) = I
RETURN
END

SUBROUTINE POP(STACK, SP, I)
INTEGER STACK(*), SP, I

IF (SP.GT.0) THEN
  I  = STACK(SP)
  SP = SP - 1
ELSE
  I  = 0
ENDIF
RETURN
END
```

For the previous graph, for instance, the mapping $t(1) = 4$, $t(2) = 2$, $t(3) = 1$, $t(4) = 3$, $t(5) = 6$, and $t(6) = 5$ is constructed.

Because matrix W is sparse and accessed along the rows within the WHILE-loop, an annotation that enforces the selection of general sparse row-wise storage for the implicitly sparse matrix W with enveloping data structure W is added to this program. Supported by loop distribution and loop interchanging, the following conversion is applied by the sparse compiler to the fragment computing the indegree, because the condition of the IF-statement always fails for zero elements, where NP_W=N and SZ_W provides sufficient space to store the entries of W:

```
DO J = 1, N
  INDEG(J) = 0
  DO I = 1, N
    IF (W(I,J).NE.0) THEN
      INDEG(J) = INDEG(J) + 1
    ENDIF
  ENDDO
ENDDO
...
```

→

```
DO J = 1, N
  INDEG(J) = 0
ENDDO
DO I = 1, N
  DO J_ = LOW_W(I), HGH_W(I)
    J = IND_W(J_)
    IF (VAL_W(J_).NE.0) THEN
      INDEG(J) = INDEG(J) + 1
    ENDIF
  ENDDO
ENDDO
...
```

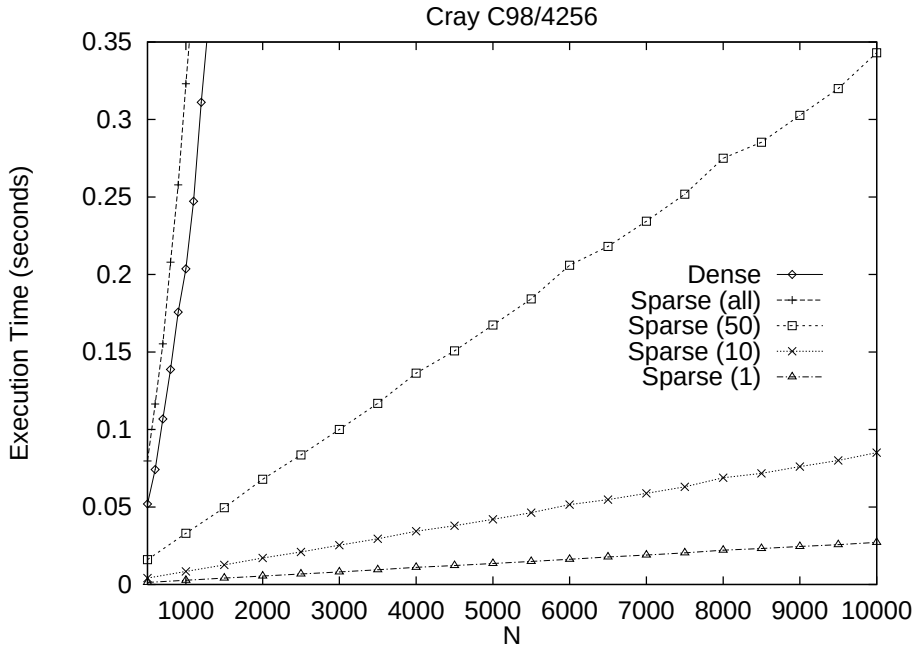


Figure 9.18: Topological Sorting

A similar transformation can be applied to the J-loop in the WHILE-loop if we inform the sparse compiler about the fact that I can be handled as a loop index and if we indicate that all data dependences caused by calling PUSH may be ignored because the order in which vertices are pushed on the stack is not essential for this algorithm. If in the original program, the graph would be represented by an adjacency matrix rather than a weight matrix, the array VAL_W would even become redundant after the conversion,

This clearly illustrates that an adjacency matrix representation of a graph can be automatically converted into the more economical adjacency structure [169, p10-13]. Hence, the storage requirements of the original program are reduced substantially, while the complexity of the algorithm is automatically reduced from $O(|V|^2)$ into $O(|V| + |E|)$, since each vertex and edge is considered only once.

In figure 9.18 we show the execution times of the original dense program and the generated sparse code on the Cray for varying values of n. The sparse code is applied to an $n \times n$ matrix representation of a complex chain in which there is an edge from each vertex to all previous vertices, to only the previous 50 or only the previous 10 vertices, and a simple chain in which there is only an edge from each vertex to the previous vertex (labeled Sparse (all), Sparse (50), Sparse (10) and Sparse (1) respectively). The chains are illustrated in figure 9.19.

Figure 9.19: Complex and Simple Chain

Chapter 10

Advanced Transformations

In previous chapters, we have demonstrated the feasibility of automatically converting a dense program into semantically equivalent sparse code. Experiments indicate that in many cases the sparse compiler is capable of transforming a particular dense fragment into code that fully exploits the sparsity of all implicitly sparse matrices to reduce both the storage requirements as well as computational time of the original implementation. Although more experiments and probably the development of more advanced transformations and strategies are required to determine whether a successful conversion is also feasible for large programs, the fact that in principle the complexity of writing sparse codes can be reduced substantially by dealing with the sparsity of matrices at the compilation level rather than at the programming level already seems to justify this new approach. So far, however, some other important issues have not been addressed.

First, because the sparse compiler is presented with the original dense program, the information obtained by data dependence analysis of this program is usually more accurate than the information that can be obtained by a compiler to which only the resulting sparse code is presented. Although this advantage already has been exploited to a certain extend, since accurate data dependence information enables more program transformations to support the selection of a suitable sparse storage scheme, data dependence information obtained by analyzing the original dense program can also be used to exploit implicit parallelism in the corresponding sparse code, as further discussed in section 10.1 of this chapter.

Second, because on one hand a reordering method may be required to preserve sparsity and, possibly, stability as well (cf. appendix A), which can be essential to keep solving a sparse problem feasible, whereas on the other hand it is difficult to express sparsity related decisions in the original dense code, some elementary support for the incorporation of both local strategies and a priori reordering methods in the generated sparse code is provided by the sparse compiler. The implementation of this support is explored in section 10.2 of this chapter.

10.1 Exploiting Parallelism in the Generated Sparse Code

Because the sparse compiler is presented with the original dense program, accurate data dependence analysis can be performed. This provides the sparse compiler with all information required for loop vectorization and concurrentization of the corresponding sparse code. However, because this information may be lost in an obscured sparse code, the information must be propagated to the FORTRAN compiler that produces machine code for a particular target machine. Moreover, the sparsity of particular matrices may even provide opportunities for loop concurrentization that are not present in the original dense code. In this section, we glance at both the direct exploitation of implicit parallelism and the exploitation of the kind of parallelism that is induced by sparsity.

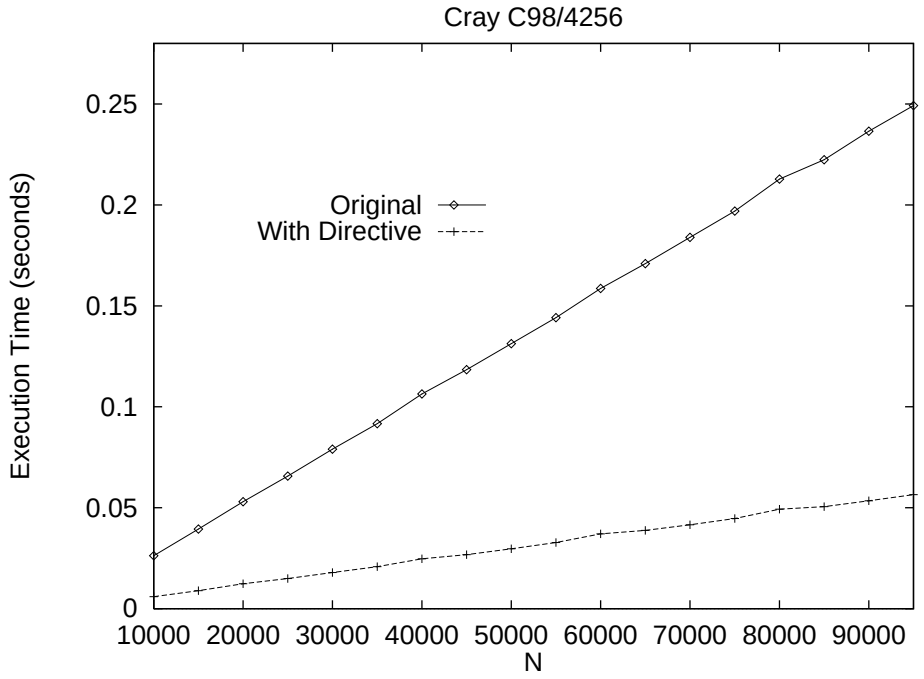


Figure 10.1: Vectorization of Sparse Forward Substitution

Because exploiting parallelism becomes complicated by the possibility of data movement in case insertions in a dynamic pool of sparse vectors may occur, we focus on fragments without the possibility of creation.

10.1.1 Direct Exploitation of Implicit Parallelism

The information that some DO-loops resulting after guard encapsulation can be vectorized may become lost in the resulting sparse code. For example, the native FORTRAN compiler of the Cray C98/4256 does not apply vectorization to the resulting sparse implementation of forward substitution in which the LDU-scheme is selected for the implicitly sparse matrix A, (see section 9.2.5), because flow dependences caused by array `B` must conservatively be assumed:

```
DO J = 1, N-1
  DO I_ = LOW_A(J+N-1), HGH_A(J+N-1)
    I = IND_A(I_)
    B(I) = B(I) + VAL_A(I_) * B(J)
  ENDDO
ENDDO
```

To prevent this loss of information, the sparse compiler may, dependent on the way information can be supplied to the FORTRAN compiler used to compile the generated sparse code, write the loop in vector syntax, generate a vectorizing directive before the `I_`-loop, or add an assertion stating that disjoint elements of `B` are referenced in each fixed iteration of the `J`-loop. For example, in figure 10.1, we show the execution time of the previous double loop on one CPU of the Cray C98/4256 for sparse matrices of varying size with 10 nonzero elements in most columns in case no information is propagated to the native compiler and in case the directive '`CDIR$ ivdep`' is added before the `I_`-loop. This experiment indicates that quite some performance can be gained by preserving essential information in the generated sparse code.

Similar arguments hold for loop concurrentization. In the sparse SAXPY implementation of $\vec{b} \leftarrow \vec{b} + A\vec{x}$, where general sparse column-wise storage is selected for A, concurrentization of the innermost DO-loop seems to be prohibited by output dependences caused by array `B`.

However, the sparse compiler may inform another FORTRAN compiler about the fact that all iterations of this DO-loop are independent using a DOALL-construct:

```
DO J = 1, N
  DOALL I_ = LOW_A(J), HGH_A(J)
    I    = IND_A(I)
    B(I) = B(I) + VAL_A(I_) * X(J)
  ENDDOALL
ENDDO
```

In a future implementation of the sparse compiler, more advanced program transformations and data structures could be incorporated to enhance the efficiency of the generated sparse code. For example, if the sparse compiler would be able to select a two-dimensional VAL_A/IND_A implementation of a pool of row-wise sparse vectors, loop interchanging could be applied to the automatically generated sequence of sparse dot products implementation of $\vec{b} \leftarrow \vec{b} + A\vec{x}$ to enhance the performance on pipelined vector processors [2, 11, 84]:

```
DO I = 1, M                                  DO J_ = 1, MAXNNZ
  DO J_ = 1, NNZ(I)                            DO I = 1, M
    J    = IND_A(I,J_)                  →        J    = IND_A(I,J_)
    B(I) = B(I) + VAL_A(I,J_) * X(J)             B(I) = B(I) + VAL_A(I,J_) * X(J)
  ENDDO                                        ENDDO
ENDDO                                        ENDDO
```

Here, MAXNNZ is equal to the maximum value of NNZ(I) and an appropriate padding is applied to the arrays. Note that if entries are sorted on column index information, effectively the extended column scheme (see section 4.1.3) has been derived automatically, which illustrates the potential of automatically converting a dense program into efficient sparse code.

10.1.2 Exploitation of Parallelism Induced by Sparsity

Because some data dependences arising in the original dense program may disappear in the corresponding sparse code, the sparsity of particular matrices may induce opportunities for concurrent execution that are not present in the original dense implementation. In this section, we glance at some methods to exploit such parallelism.

Elimination of Data Dependences

After a dense program has been converted into semantically equivalent sparse code, many statement instances arising in the original program are no longer executed in the sparse code. This implies that converting a dense program into sparse code also affects the data dependences that arise in the original program, because *a data dependence of which the sink or source statement instance is not executed disappears.*

For example, below we present the conditions associated with some scalar statements in a fragment in which array A is used as enveloping data structure of an implicitly sparse matrix A:

```
      REAL A(M,N)
C_SPARSE(A)
      ...
S1:   B(3) = ...                    ← true
S2:   ACC  = ACC + A(1,3) * B(3)    ← (1,3) ∈ E(A)
S3:   C(1) = ACC                    ← true
```

In this fragment, the data dependences $S_1 \delta S_2$ and $S_2 \delta S_3$ hold, reflecting the fact that these scalar statements must be executed serially. However, because the compiler associates the condition '$(1,3) \in E(A)$' with S_2, the data dependence chain $S_1 \rightarrow S_2 \rightarrow S_3$ is broken if a_{13} is not an entry.

Since S_1 and S_3 are independent, the elimination of the data dependences enables concurrent execution of these scalar statements. Consequently, the implicit synchronization from S_1 to S_2 and from S_2 to S_3 that is induced by the serial semantics of the program can be placed under control of the condition. Hence, if general sparse row-wise storage is selected for A, the compiler may generate the following sparse code, where a COBEGIN-construct [67][175, p65-66] is used to express concurrency at statement level:

```
   L_0 = LKP__(IND_A, LOW_A(1), HGH_A(1), 3)
   IF (L_0 .NE. ⊥) THEN
S1:  B(3) = ...
S2:  ACC  = ACC + VAL_A(L_0) * B(3)
S3:  C(1) = ACC
   ELSE
     COBEGIN
S1:    B(3) = ...
S3:    C(1) = ACC
     COEND
   ENDIF
```

If, at compile-time, either static dense storage is selected for the region in which element a_{13} resides, or we know that the element resides in a zero region, then only one of these branches has to be generated, because in these cases the condition associated with S_1 becomes either '**false**' or '**true**' respectively.

Because only a small reduction of execution time may be expected from such fine grain parallelism, in the next section we explore whether a similar technique can be used to enhance concurrency in loops.

Concurrency in Loops

If a loop is converted into sparse code, then, dependent on the nonzero structures of the sparse matrices involved in the computation, some data dependences arising in the original loop may disappear. As for the example in the previous section, only a small reduction of execution time may be expected from exploiting the fact that some loop-*independent* data dependences disappear, as this would enable the concurrent execution of some statements instances belonging to the same iteration. Clearly, exploiting the elimination of some loop-carried data dependences has more potential, since this can make different iterations of a loop completely independent.

To capture the data dependence structure of a loop explicitly in the program text, we assume that particular DO-loops in the original dense program have been converted into DOACROSS-loops by generating random synchronization that enforces all loop-carried data dependences (see section 3.2.2). For many loops, such synchronization enforces (nearly) serial execution. However, if the program is converted into sparse code, we can exploit the fact that some synchronization is not required if a loop-carried data dependence disappears due to the fact that an instance of a sink statement is not executed:

> If the sink statement of a static data dependence of which the underlying data dependences are carried by a DOACROSS-loop is under control of a condition, the corresponding **wait**-statement can be placed under control of this condition.

Likewise, if the *source* statement is under control of a condition, in principle we could place the **wait**-statement under control of the condition as well. However, because this requires the re-evaluation of conditions used in earlier iterations, usually it provides the sparse compiler with limited opportunities to actually eliminate the synchronization.

Under the assumption that bits may remain untested in the bit array implementation of random synchronization, it is not useful to place the corresponding **post**-statement under control of a condition of either the sink or source statement, because this operation is non-blocking.

	Dense Matrix				
I	2	3	4	5	6
	$S_1(2)$	wB2	wB3	wB4	wB5
t	$S_2(2)$	·	·	·	·
i	pB2	·	·	·	·
m		$S_1(3)$	·	·	·
e		$S_2(3)$	·	·	·
		pB3	·	·	·
↓			$S_1(4)$	·	·
			$S_2(4)$	·	·
			pB4	·	·
				$S_1(5)$	·
				$S_2(5)$	·
				pB5	·
					$S_1(6)$
					$S_2(6)$
					pB6

	$(4,4) \notin E(A)$ and $(6,6) \notin E(A)$				
I	2	3	4	5	6
	$S_1(2)$	wB2	$S_2(4)$	wB4	$S_2(6)$
t	$S_2(2)$	·	pB4	·	pB6
i	pB2	·		$S_1(5)$	
m		$S_1(3)$		$S_2(5)$	
e		$S_2(3)$		pB5	
		pB3			
↓					

Table 10.1: Elimination of Random Synchronization

Example: If the following DO-loop is converted into a concurrent loop, then the underlying data dependences of the static data dependence $S_2\delta_< S_1$ must be explicitly enforced by synchronization, whereas all underlying data dependences of $S_1\delta_= S_2$ are simply enforced by serial execution of the loop-body within every iteration:

```
   DO I = 2, N
S1:  C(I)   = C(I) + A(I,I) * B(I)
S2:  B(I+1) = C(I)
   ENDDO
```

→

```
DOACROSS I = 2, N
  wait(BSYNC,I-1)
  C(I)   = C(I) + A(I,I) * B(I)
  B(I+1) = C(I)
  post(BSYNC,I)
ENDDOACROSS
```

Obviously, synchronizing the lexically backward data dependence effectively serializes the loop in the dense case. However, if subsequently the sparse compiler is informed about the fact that array A is used as enveloping data structure of an implicitly sparse matrix A, then the compiler associates the condition '$(\mathtt{I},\mathtt{I}) \in E(A)$' with S_1.

Since the previous rule states that the **wait**-statement used to enforce the underlying data dependences of $S_2\delta_< S_1$ can also be placed under this condition, the sparse compiler can convert the DOACROSS-loop into the following sparse code in case general sparse row-wise storage is selected for A:

```
DOACROSS I = 2, N
  L_0 = LKP__(IND_A, LOW_A(I), HGH_A(I), I)
  IF (L_0 .NE. ⊥)
    wait(BSYNC,I-1)
    C(I) = C(I) + VAL_A(L_0) * B(I)
  ENDIF
  B(I+1) = C(I)
  post(BSYNC,I)
ENDDOACROSS
```

In table 10.1, the serial execution order for a dense matrix and the execution order for a sparse matrix with $(4,4) \notin E(A)$ and $(6,6) \notin E(A)$ are shown for N=6, where pBi and wBi are used as an abbreviation of **post**(BSYNC,i) and **wait**(BSYNC,i) respectively.

Example: The outermost DO-loop of the following implementation of forward substitution can be converted into a DOACROSS-loop as follows:

```
   DO I = 1, N                                  DOACROSS I = 1, N
     DO J = 1, I-1                                DO J = 1, I-1
S1:    X(I) = X(I) - A(I,J) * X(J)       →         wait(XSYNC,J)
     ENDDO                                        X(I) = X(I) - A(I,J) * X(J)
S2:  X(I) = X(I) / A(I,I)                       ENDDO
   ENDDO                                        X(I) = X(I) / A(I,I)
                                                post(XSYNC,I)
                                              ENDDOACROSS
```

Although underlying dependences of $S_2\delta_< S_1$ hold between a *single* source statement instance and sink statement instances in *all* following iterations, the synchronization variable XSYNC can still be implemented as a one-dimensional bit vector by using the fact that several **wait** instances may test the same bit.

Again, synchronization serializes the loop in the dense case. If the sparse compiler is informed about the fact that array A is used as enveloping data structure of an implicitly sparse matrix A, however, then the compiler associates the condition '$(\mathtt{I},\mathtt{J}) \in E(A)$' with S_1 and, hence, with the **wait**-statement. Consequently, because '$(\mathtt{I},\mathtt{J}) \in E(A)$' dominates all statements in the loop-body of the J-loop, the sparse compiler can generate the following sparse code if sparse row-wise storage and static dense storage is used for respectively entries in the strict lower triangular part and main diagonal of A:

```
X(1) = X(1) / DN1_A(1)
DOACROSS I = 2, N
  DO J_ = LOW_A(I), HGH_A(I)
    J = IND_A(J_)
    wait(XSYNC,J)
    X(I) = X(I) - VAL_A(J_) * X(J)
  ENDDO
  X(I) = X(I) / DN1_A(I)
  post(XSYNC,I)
ENDDOACROSS
```

Here, we see the full potential of combining sparse code generation with synchronization elimination because, in contrast with the previous example, all lookup overhead to determine whether synchronization and an arithmetic operation are required has been eliminated by guard encapsulation.

Note that, in this example, all underlying memory-based data dependences of $S_1\delta_{<,<}S_1$ are covered by synchronization of the underlying value-based data dependences of $S_1\delta_< S_2$ in both the dense and sparse fragment. In general, however, if the sparse compiler is used to generate conditions for random synchronization, both value and memory-based-data dependences should be explicitly enforced by random synchronization, whereas methods to eliminate redundant synchronization (see e.g. [141, 155, 156, 157]) should only be applied after all conditions have been generated.

Effective Exploitation of Concurrency

Although performing random synchronization conditionally may reveal much concurrency, this approach also increases run-time overhead and the demand for memory or special hardware resources. Moreover, the performance may drop if the data dependence structure induced by particular nonzero structures and the used scheduling policy do not match. However, suppose that we can partition the execution set of a DOACROSS-loop into the sets $I_1, \ldots, I_m$, such that during any iteration $i \in I_l$, random synchronization only tests bits that are set during an iteration $i \in I_{l'}$ with $l' < l$. Then, the DOACROSS-loop can be executed as a sequence of DOALL-loops as follows:

```
DO l = 1, m
  DOALL I ∈ I_l
   ...
  ENDDOALL
ENDDO
```

Some concurrency may be lost, because barrier synchronization after each DOALL-loop enforces that for $l > 1$, each iteration in I_l must wait for the completion of *all* iterations in I_{l-1}, even if the iterations on which it actually depends already have been executed. However, this concurrency may be traded for the reduction of synchronization overhead, while the efficiency of a DOALL-loop is less sensitive to the scheduling policy.

Below, we present the framework of serial pre-evaluation code that can be generated automatically by the sparse compiler to compute the sets $I_1, \ldots, I_m$ at run-time before the actual loop is executed (inspired on the pre-evaluation code found in the module UNARY of SPARSKIT [185]). This computation is based on a levelization of the data dependence structure arising in the loop. For a DOACROSS-loop with execution set $[\mathtt{L}, \mathtt{U}]$, first a level `L` is assigned to each iteration, after which the iterations are sorted on the `MLEV` resulting levels in an array `ISET` that is accessed through a pointer structure stored in `LVP`:

```
C Compute Levelization
      MLEV = 0
      DO I = L, U
         L = 1
         ... computation of L ...  ← (+)
         LEV(I) = L
         LVP(L) = LVP(L) + 1
         MLEV   = MAX(MLEV, L)
       ENDDO
C Compute Pointer Structure
      DO I = 2, MLEV+1
        LVP(I) = LVP(I) + LVP(I-1)
      ENDDO
C Sort Iterations
      DO I = U, L, -1
        L              = LEV(I)
        LVP(L)         = LVP(L) - 1
        ISET( LVP(L) ) = I
      ENDDO
```

The DOACROSS-loop itself is replaced by the following implementation of the sequence of DOALL-loops, where the original loop-body appears at the dots:

```
DO L = 1, MLEV
  DOALL II = LVP(L)+1, LVP(L+1)
    I = ISET(II)
    ...
  ENDDOALL
ENDDO
```

The actual computation of the level of each iteration that is generated at position $(+)$ depends on the synchronization statements appearing in the sparse code. For example, for the simple example of the previous section, the sparse compiler generates the following computation of `L`, in which out-of-bounds synchronization is explicitly protected:

```
L_0 = LKP__(IND_A, LOW_A(I), HGH_A(I), I)
IF (L_0 .NE. ⊥)
  IF (1.LE.I-1) L = MAX(L, LEV(I-1))
ENDIF
```

In figure 10.2, we show the resulting contents of arrays `LVP` and `ISET` for the given nonzero structure of the main diagonal of the implicitly sparse matrix A. Likewise, the level of each iteration for forward substitution is computed by generating the following code at position $(+)$:

```
DO J_ = LOW_A(I), HIGH_A(I)
  J = IND_A(J_)
  IF (2.LE.J) L = MAX(L, LEV(J))
ENDDO
```

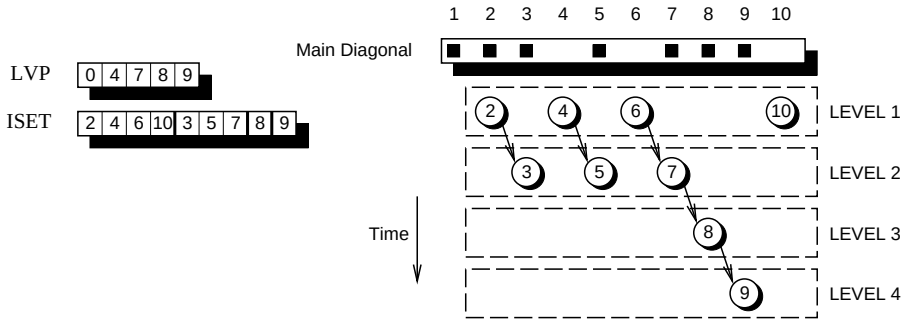


Figure 10.2: Levelization (simple example)

In figure 10.3, we show the results of the levelization for the given nonzero structure of the implicitly sparse matrix A. For this matrix, all iterations in the first level set $I_1 = \{1, 2, 3, 5\}$ can be executed independently. Thereafter, iterations with nonzero elements in only the columns defined by I_1 may start, which are the remaining iterations $I_2 = \{4, 6\}$. In figure 10.4, we present the execution time of the serial implementation of forward substitution, the execution time required for a levelization, and the wall clock time required to perform the sequence of DOALL-loops on four CPUs of a Cray C98/4256 for sparse matrices of varying size with arrow nonzero structures similar to the matrix arising from minimum degree in figure 4.13. Although a speedup of 3.5 is obtained in some cases, computing the levelization is almost as expensive as performing the actual computation. Hence, this approach is only useful if the time required to compute the levelization of the data dependence structure arising in a loop can be amortized over many concurrent executions of this loop or if the levelization itself is concurrentized (see discussion below).

Related Work on Run-Time Loop Concurrentization

Other work has addressed the run-time concurrentization of a loop in which the data dependence structure is determined by loop-invariant contents of arrays that are used as subscripts.

In [160], run-time disambiguation (determining whether references are independent) is used to enhance concurrency in a stream of instructions, although some remarks about generalizing the method to DO-loops is made. In subscript blocking [170, p66-81], the compiler generates pre–evaluation code that determines subsets of consecutive iterations that may be executed concurrently, whereas the original loop is converted into a sequence of DOALL-loops. This method, however, does not allow for reordering the individual iterations to enhance concurrency.

In [155][157, p115-121][233], a method that determines subsets of arbitrary independent iterations is presented, which suffers from substantial overhead because independent iterations are executed using a mask which is re-computed between the execution of successive subsets. In [11, 187, 184], the concurrentization of sparse triangular systems solvers is considered.

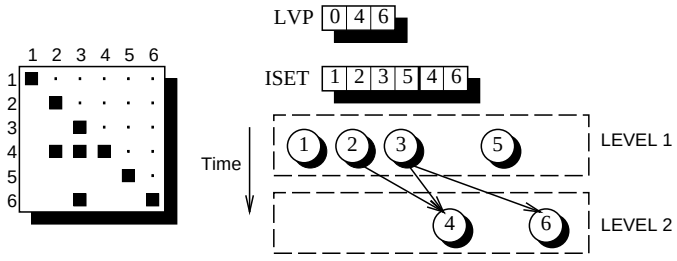


Figure 10.3: Levelization (Forward Substitution)

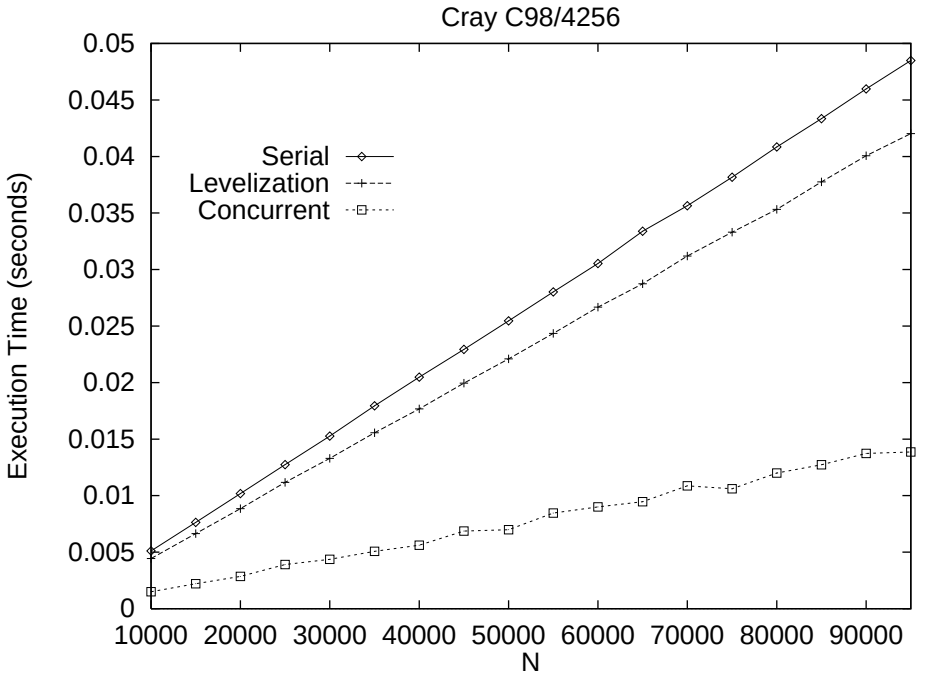


Figure 10.4: Concurrentization of Forward Substitution (four CPUs)

These methods are generalized in the run-time loop concurrentization method of [188, 189, 190, 206, 204]. Here, an **inspector** is generated before a loop that determines subsets of independent iterations, called wavefronts, which are executed by an **executor** as a sequence of DOALL-loops, or, alternatively, in a DOACROSS-like construct. To reduce the execution time of the inspector, which is proportional to the execution time of the original loop (cf. figure 10.4), the inspector itself may be concurrentized [126, 134, 189].

This can be done using sectioning [134], where each individual process computes the wavefronts for a contiguous range of iterations, after which the wavefronts of different processes are simply concatenated. Since this method does not necessarily have the fewest possible wavefronts, the results of sectioning may be used to re-execute the inspector concurrently, yielding the same wavefronts as would result using a serial inspector [134].

Note that these methods are only applicable to the actual sparse code and that all potential data dependences must conservatively be accounted for. In contrast, the sparse compiler is presented with the original dense code, so that program and data structure transformations may be applied before the actual sparse code is generated, while data dependence analysis, in general, yields more accurate information. For example, in the sparse implementation of forward substitution, the potential static data dependence $S_1 \overline{\delta}_< S_1$ must be accounted for, whereas the sparse compiler may safely discard this data dependence. In principle, however, the same concurrency is obtained and the sparse compiler can benefit from the results arising from further research in this area.

10.2 Towards Incorporating Reordering Methods

The sparse compiler provides some elementary support for the incorporation of reordering methods, suited for the incorporation of local strategies and a priori reordering methods for general sparse matrices (cf. appendix A).

10.2.1 Recording a Permutation

The programmer can specify that an $m \times n$ implicitly sparse matrix A with enveloping data structure A will be permuted into PAQ at run-time using the following annotation:

```
     REAL A(M,N)
C_SPARSE(A : _PERM)
```

This annotation indicates that any a priori reordering method may be applied to A before this implicitly sparse matrix is initialized, whereas particular row and column interchanges may be applied to A at the position of an interchange annotation involving the array A. The actual implementation of permutations is kept transparent to the programmer. The sparse compiler is responsible for generating code in which permutations are applied and recorded. As far as the programmer is concerned, all programming can be done on the enveloping data structure A as if elements are physically moved in this two-dimensional array, i.e. if at a particular moment A is permuted into PAQ, then the programmer may assume that A contains the elements of PAQ.

Because permuting A may affect properties of the nonzero structure of this matrix in an unpredictable manner, exploiting peculiarities of the nonzero structure seems to be difficult. Therefore, in the current prototype sparse compiler, only general sparse storage schemes can be selected for A. If general sparse row-wise storage is selected, then the sparse compiler adds the following permutation arrays to the declarations of the sparse storage scheme (general sparse column-wise storage is implemented analogously):

```
INTEGER ROW_A(1:M), COL_A(1:N), INVCOL_A(1:N)
COMMON  /A/ ..., ROW_A, COL_A, INVCOL_A
```

These permutation arrays are used to record the permutation matrices P, Q, and Q^T according to the method presented in [78, 169, p34-35], i.e. the arrays have the contents $P(1, \ldots, m)^T$, $(1, \ldots, n)Q$, and $(1, \ldots, n)Q^T$ respectively. The initial contents of the first two permutation arrays must be specified in the file that is used to initialize A at run-time. Given such an implicitly sparse matrix A with enveloping data structure A, the sparse compiler adds the following fragment to the initialization code:

```
OPEN (UNIT=1, FILE='file_name', STATUS='OLD')
DO I_ = 1, M
  READ (1,*) ROW_A(I_)
ENDDO
DO I_ = 1, N
  READ (1,*) COL_A(I_)
ENDDO
...
CLOSE (UNIT = 1)
```

At the dots, the matrix is initialized as discussed earlier. However, after the whole matrix has been initialized, the following code is executed to perform the row interchanges by simply applying P to the row pointers, where the temporary integer array TMP_ is used:

Permute LOW_A:

```
DO I_ = 1, M
  TMP__(I_) = LOW_A( ROW_A(I_) )
ENDDO
DO I_ = 1, M
  LOW_A(I_) = TMP__(I)
ENDDO
```

Permute HGH_A:

```
DO I_ = 1, M
  TMP__(I_) = HGH_A( ROW_A(I_) )
ENDDO
DO I_ = 1, M
  HGH_A(I_) = TMP__(I)
ENDDO
```

However, in row-wise storage, permuting the columns is not so straightforward. Therefore, in addition to recording Q, permutation matrix Q^T is also recorded. The following fragment is generated to initialize the array INVCOL_A:

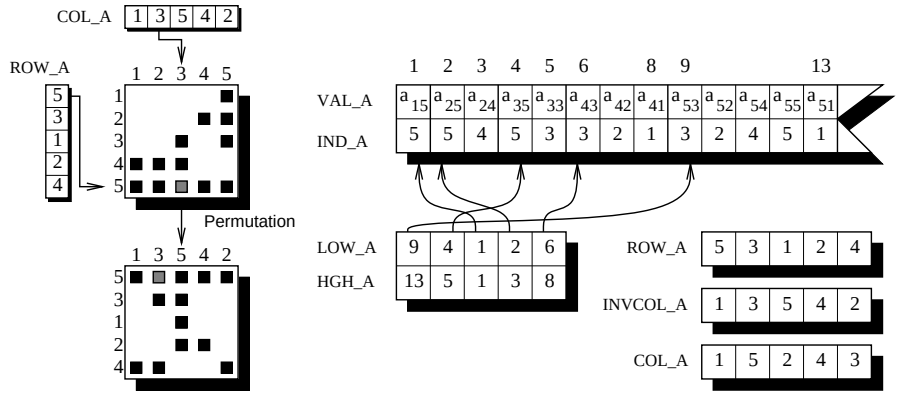


Figure 10.5: Permutation

```
DO I_ = 1, N
  INVCOL_A( COL_A(I_) ) = I_
ENDDO
```

Now, the sparse compiler may generate sparse code as before, thereby accounting for the fact that an entry in the permuted matrix PAQ with row index i and column index j is stored in the ith sparse vector with index information `COL_A`(j) whereas, conversely, an entry stored in the ith sparse vector with index information j corresponds to an entry in the ith row and `INVCOL_A`(j) column of the permuted matrix. In this manner, any a priori reordering method can be incorporated by computing this reordering just before the generated sparse code is executed, after which the contents of the corresponding permutation arrays are added to the file in which the implicitly sparse matrix resides in coordinate scheme.

Example: Below, we give an example in which an a priori reordering method is applied to the 5×5 implicitly matrix stored in the file '`mat1.cs`'. The effects of permuting A by physically moving the elements are illustrated in figure 10.5, where, for instance, the entry with row index 1 and column index 2 in the permuted matrix corresponds to a_{53} in the original matrix. The way in which the permutation is recorded in general sparse row-wise storage is also illustrated in the figure.

```
      PROGRAM PERMUTE

      INTEGER    N
      PARAMETER (N = 5)
      REAL       A(N,N)
C_SPARSE(A : _FILE('mat1.cs') )
C_SPARSE(A : _PERM)
      ...
      END
```

contents of file '`mat1.cs`'

```
5 3 1 2 4 ← permutation arrays
1 3 5 4 2
5 5       ← coordinate scheme
13
1 5 1.5 2 4 2.4 2 5 2.5 3 3 3.3
3 5 3.5 4 1 4.1 4 3 4.3 4 2 4.2
5 1 5.1 5 3 5.3 5 2 5.2 5 4 5.4 5 5 5.5
```

Note that, alternatively, a file in which the *permuted* matrix is stored in coordinate scheme could be generated and used as input for the generated sparse program. However, in this case, particular input and output data should be permuted accordingly. For example, while solving a linear system of inequalities with a permuted coefficient matrix PAQ, all right-hand sides must be permuted according to P, whereas the resulting solutions must be permuted according to Q. The permutation mechanism of the sparse compiler enables the programmer to deal with these issues only once, at the expense of some run-time overhead and the obligation to add the correct induction annotations to the original dense program. In addition, the permutation mechanism of the sparse compiler enables the incorporation of local strategies.

10.2.2 Implementation of Induction Annotations

The implementation of induction annotation is rather straightforward. Because a row permutation matrix P of an implicitly sparse matrix with enveloping data structure A is recorded with the contents $P(1, \ldots, m)$ in the permutation array ROW_A, the induction annotations involving this row permutation matrix are implemented as follows, where array PERM_A is a locally declared temporary array having the same basis type as A:[1]

```
action: X < _ROW(A)                 action: X > _ROW(A)

DO I_ = 1, M                        DO I_ = 1, M
  PERM_A(I_) = X( ROW_A(I_) )         PERM_A(I_) = X(I_)
ENDDO                               ENDDO
DO I_ = 1, M                        DO I_ = 1, M
  X(I_) = PERM_A(I_)                  X( ROW_A(I_) ) = PERM_A(I_)
ENDDO                               ENDDO
```

Likewise, because a column permutation matrix Q associated with an implicitly sparse matrix A with enveloping data structure A is recorded with the contents $(1, \ldots, n)Q$ in the permutation array COL_A, the induction annotations involving this column permutation matrix are implemented as follows:

```
action: Y < _COLUMN(A)            action: Y > _COLUMN(A)

DO I_ = 1, N                      DO I = 1, N
  PERM_A(I_) = Y( COL_A(I_) )       PERM_A(I_) = Y(I_)
ENDDO                             ENDDO
DO I_ = 1, N                      DO I_ = 1, N
  Y(I_) = PERM_A(I_)                Y( COL_A(I_) ) = PERM_A(I_)
ENDDO                             ENDDO
```

10.2.3 Implementation of Interchange Annotations

The programmer can specify that for an implicitly sparse matrix A with enveloping data structure A an arbitrary row and column in the range $[\mathtt{LR}, \mathtt{UR}]$ and $[\mathtt{LC}, \mathtt{UC}]$ respectively may be interchanged with the Rth row and Cth column, using the following interchange annotation:

```
C_INTERCHANGE(A, LR:UR > R, LC:UC > C)
```

Rather than directly specifying the criteria for a local strategy that must be used to determine a row and column, the sparse compiler may select these criteria after analyzing the program. After a particular local strategy has been selected, the sparse compiler is also responsible for generating code that implements the selected local strategy. In this code, at run-time desired row and column interchanges are determined and applied. In this manner, the sparse compiler is not restricted to selecting already existing local strategies, but may derive a local strategy that is suited for a particular fragment.

Applying the actual interchanges is straightforward. Interchanging two rows is simply performed by interchanging the corresponding row pointers, while corresponding elements in array ROW_A are interchanged to record the new permutation matrix P. Interchanging two columns is performed by altering the permutation arrays COL_A and INVCOL_A to record the new permutation matrices Q and Q^T:

```
Interchanging Rows I1 and I2:       Interchanging Columns J1 and J2:

SWAP( ROW_A(I1), ROW_A(I2) )        SWAP( COL_A(J1), COL_A(J2) )
SWAP( LOW_A(I1), LOW_A(I2) )        INVCOL_A( COL_A(J1) ) = J1
SWAP( HGH_A(I1), HGH_A(I2) )        INVCOL_A( COL_A(J2) ) = J2
```

[1] Alternatively, array X could be permuted in-place if the permutation would be stored as a sequence of interchanges [78, p34-35].

Below, an example is given in which we explore the steps that could be taken by the sparse compiler to incorporate a very simple local strategy. Obviously, as more advanced methods to select criteria for a local method have been developed and incorporated in the sparse compiler, a dense program that has been used earlier to obtain sparse code can be re-translated into more efficient sparse code without any modifications to the program.

Example: Consider the following implementation of LU-factorization, where annotations are used to indicate that any a priori reordering may be applied to the implicitly sparse matrix A with enveloping data structure A. Moreover, interchanging annotations are used to indicate that an arbitrary element in the active sub-matrix may be used as pivot:

```
      PROGRAM SOLVE                      SUBROUTINE FACT(A, N)
                                         INTEGER I, J, K, N
      INTEGER    N                       REAL    A(N,N)
      PARAMETER (N = ...)
      REAL      A(N,N), B(N)             DO K = 1, N - 1
C_SPARSE(A : _PERM)              C_INTERCHANGE(A, K:N > K, K:N > K)
      ...                                  DO I = K + 1, N
      CALL FACT(A,    N)                     A(I,K) = A(I,K) / A(K,K)
      CALL FORW(A, B, N)                     DO J = K + 1, N
      CALL BACK(A, B, N)                       A(I,J) = A(I,J) - A(I,K)*A(K,J)
      ...                                    ENDDO
      END                                  ENDDO
                                         ENDDO
                                         RETURN
                                         END
```

As far as the programmer is concerned, data in the enveloping data structure A is physically moved according to PAQ, so that eventually the enveloping data structure A is overwritten with elements of the factors L and U satisfying $PAQ = LU$ for the original matrix A. Consequently, forward and back substitution can be implemented as follows. A right-hand side vector $\vec{b}$ is permuted into $P\vec{b}$ before forward substitution, whereas the in-place computed solution $\vec{x}$ is permuted into $Q\vec{x}$ to obtain the desired solution after back substitution. Note that a system $AX = B$ can be solved by repetitively calling subroutines FORW and BACK with the columns of X and B after a factorization $PAQ = LU$ has been computed, which effectively solves $PAQQ^TX = PB$ (see appendix A for a detailed discussion of LU-factorization):

```
     SUBROUTINE FORW(A, B, N)            SUBROUTINE BACK(A, B, N)
     INTEGER I, J, N                     INTEGER I, J, N
     REAL    A(N,N), B(N)                REAL    A(N,N), B(N)
C_INDUCE B < _ROW(A)                     DO I = N, 1, -1
     DO I = 1, N                           DO J = I + 1, N
       DO J = 1, I - 1                       B(I) = B(I) - A(I,J) * B(J)
         B(I) = B(I) - A(I,J) * B(J)       ENDDO
       ENDDO                               B(I) = B(I) / A(I,I)
     ENDDO                               ENDDO
     RETURN                          C_INDUCE B > _COLUMN(A)
     END                                 RETURN
                                         END
```

Because the sparse compiler generates the clones FACT_A0, FORW_A00, and BACK_A00 in which A is uniquely associated with the formal argument A, all permutation annotations uniquely define the permutation matrices associated with the implicitly sparse matrix A.

Straightforward conversion of the factorization code into sparse code yields a slightly different version of the code presented in chapter 9. Now, however, permutations are accounted for:

```
DO K = 1, N
 ... interchanging code ...
 DO I = K+1, N
   CALL SSCT__(VAL_A, IND_A, LOW_A(I), HGH_A(I), SAP_20, SWT_20)
   IF (SWT_20(COL_A(K))) THEN
     SAP_20(COL_A(K)) = SAP_20(COL_A(K)) /
+          VAL_A(LKP__(IND_A, LOW_A(K), HGH_A_A(K), COL_A(K)))
     ...
```

```
      LEN_J = HGH_A(K) - LOW_A(K)
      DO J_ = 0, LEN_J
        J = INVCOL_A( IND_A(LOW_A(K)+J_) )
        IF (K+1.LE.J) THEN
          IF (.NOT.SWT_20(COL_A(J))) THEN
           SWT_20(COL_A(J)) = .TRUE.
           CALL SINS__(VAL_A,IND_A,LOW_A,HGH_A,I,NP_A,SZ_A,LST_A,L_,COL_A(J))
          END IF
          SAP_20(COL_A(J)) = SAP_20(COL_A(J)) - SAP_20(COL_A(K))*VAL_A(LOW_A(K)+J_)
        END IF
      ENDDO
    ENDIF
    CALL SGTH__(VAL_A, IND_A, LOW_A(I), HGH_A(I), SAP_20, SWT_20)
  ENDDO
ENDDO
```

As shown in boldface, permutation arrays are used to translate column indices of the stored sparse matrix into column indices of the permuted matrix. Because row interchanges are applied by interchanging the row pointers accordingly, this translation is not required for rows. Likewise, straightforward conversion of the implementation of forward and back substitution yields the following sparse fragments, in which the induction annotations are explicitly implemented:

```
DO I_ = 1, N
  PERM_A(I_) = B(ROW_A(I_))
ENDDO
DO I_ = 1, N
  B(I_) = PERM_A(I_)
ENDDO
DO I = 2, N
  DO J_ = LOW_A(I), HGH_A(I)
    J = INVCOL_A( IND_A(J_) )
    IF (J.LE.I-1) THEN
      B(I) = B(I) - VAL_A(J_) * B(J)
    ENDIF
  ENDDO
ENDDO
```

```
DO I = N, 1, -1
  IF (I+1.LE.N) THEN
    DO J_ = LOW_A(I), HGH_A(I)
      J = INVCOL_A( IND_A(J_) )
      IF (I+1.LE.J) THEN
        B(I) = B(I) - VAL_A(J_) * B(J)
      ENDIF
    ENDDO
  ENDIF
  B(I) = B(I) / VAL_A(LKP__(IND_A,
+    LOW_A(I), HGH_A(I), COL_A(I)))
ENDDO
DO I = 1, N
  PERM_A(I_) = B(I_)
ENDDO
DO I_ = 1, N
  B(COL_A(I_)) = PERM_A(I_)
ENDDO
```

Subsequently, the sparse compiler may select criteria to implement the interchange annotation. In fact, any local strategy varying from the minimum row in minimum column to the Markowitz or minimum deficiency strategy may arise, because the sparse compiler can determine the conditions under which creation occurs in the I- and J-loop.

For example, because any row in the range [K, N] may be interchanged with row K and because the expression 'HGH_A(K)-LOW_A(K)' provides a very rough measure of the amount of fill-in that could occur at each step K, the sparse compiler may decide to generate the following interchanging code in which a row that minimizes this expression is selected and interchanged with row K:

```
C DETERMINE ROW
      LEN = ∞
      DO I = K, N
        IF ((HGH_A(I) - LOW_A(I)).LT.LEN) THEN
          KK  = I
          LEN = HGH_A(I) - LOW_A(I)
        ENDIF
      ENDDO

C INTERCHANGE ROWS
      SWAP( ROW_A(K), ROW_A(KK) )
      SWAP( LOW_A(K), LOW_A(KK) )
      SWAP( HGH_A(K), HGH_A(KK) )
```

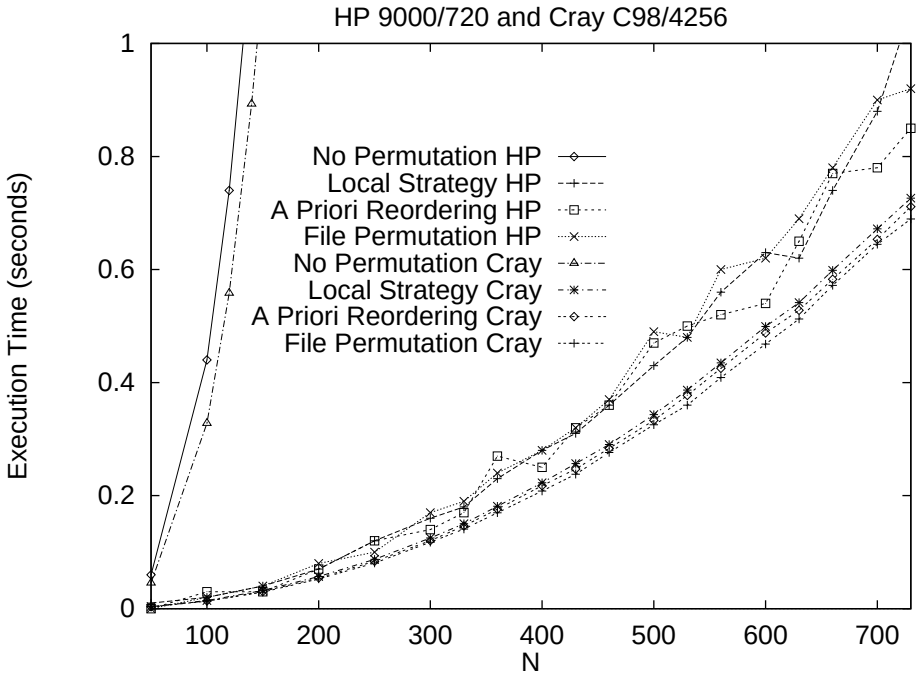


Figure 10.6: Solving $A\vec{x} = \vec{b}$

Furthermore, since element A(K,K) is involved in a division and because any column in the range [K, N] may be interchanged with column K and because the pivot is involved in a division, the sparse compiler may decide to generate code in which an element in the upper triangular part of the selected row with maximum absolute value is interchanged to the pivotal position. A possible implementation that assumes that such an element always can be found is shown below:

```
C DETERMINE COLUMN
      MX = -∞
      DO J_ = LOW_A(K), HGH_A(K)
        J = INVCOL_A(IND_A(J_))
        IF (K.LE.J) THEN
          IF (ABS(VAL_A(J_)).GT.MX) THEN
            KK = J
            MX = ABS(VAL_A(J_))
          ENDIF
        ENDIF
      ENDDO

C INTERCHANGE COLUMNS
     SWAP( COL_A(K), COL_A(KK) )
     INVCOL_A( COL_A(K)  ) = K
     INVCOL_A( COL_A(KK) ) = KK
```

In figure 10.6, we present the execution times of different versions to solve $A\vec{x} = \vec{b}$ on both the HP and one CPU of the Cray for sparse matrices of varying size with arrow nonzero structures similar to the first matrix in figure 4.13. Here, the execution times of the sparse implementations without any permutation overhead when applied to the non-permuted matrices and matrices permuted on file using an a priori computed minimum degree ordering are shown. Furthermore, the execution time of the version in which this a priori reordering is implemented using the permutation mechanism of the sparse compiler and the version with the previous presented local strategy are shown. Obviously, the incorporation of a reordering method enables a more effective exploitation of the sparsity of the coefficient matrix.

Unfortunately, however, due to the fact that in the current prototype only general sparse storage can be selected for implicitly sparse matrices that may be permuted, still a sparse implementation of LU-factorization results of which the execution time grows at least quadratically in the order of the coefficient matrix. Because this makes solving large sparse systems infeasible, in a future implementation, the incorporation of reordering methods should be combined with the selection of more advanced data structures in which, for example, the column structure of the matrix is stored as well [105, 235]. In addition, more advanced data structures may be required to support the selected local strategy in which, for example, linked lists sort rows and columns in increasing order of row and column count [74, 81]. Note that, in contrast with the previous example, in which an inefficient pivot selection method could be used due to the fact that an inefficient implementation of LU-factorization itself was used, in efficient implementations of LU-factorization, it becomes extremely important to perform the selection of the next pivot very efficiently as well.

Chapter 11

Conclusions

Although developing and maintaining sparse codes is a complex and cumbersome task, only limited compiler support for sparse matrix computations has been developed in the past. In this dissertation, we have tried to make a step towards solving this omission by proposing an alternative method to develop sparse codes. Rather than dealing with the sparsity at the programming level, as is done traditionally, the sparsity of implicitly sparse matrices is dealt with at the compilation level by a sparse compiler. In this approach, the programmer defines all operations on implicitly sparse matrices using simple two-dimensional arrays. The burden of sparse code generation is completely placed on the sparse compiler which selects a suitable sparse storage scheme for each implicitly sparse matrix and transforms the original dense code into sparse code that operates on these selected storage schemes, thereby reducing the storage requirements and computational time of the original dense program. In the introduction of chapter 4, several advantages of this approach have been discussed. Elaboration of these ideas has resulted in the development and implementation of a prototype sparse compiler. The automatic data structure selection and transformation method that is used by this sparse compiler has been presented in detail in this dissertation. In addition, some initial experiments have been conducted that indicate that in many cases the sparse compiler is capable of transforming a dense fragment into code that fully exploits the sparsity of some matrices to reduce both the storage requirements as well as computational time of the original implementation. Finally, we discussed the automatic exploitation of implicit parallelism in the generated sparse code and explored incorporating sparsity preserving reordering methods.

In this final chapter, we discuss the contributions of our research and shortcomings of the current prototype. Moreover, we glance at related work, and we state topics for future research.

11.1 Contributions of this Research

Restricted by time constraints inherent to a four year research project on one hand, but driven by the desire to actually implement the ideas proposed in the dissertation to demonstrate the feasibility of these ideas on the other hand, we decided to implement a prototype sparse compiler by incorporating the automatic data structure selection and transformation method in an existing prototype restructuring compiler MT1 [24, 37, 45]. The resulting prototype sparse compiler, in which not all issues that should be dealt with in a commercially acceptable compiler are accounted for [212], provides sufficient functionality to make a first claim about the feasibility of automatically generating sparse codes.

The experiments of chapter 9 indicate that in many cases the sparse compiler is able to convert a particular dense fragment into semantically equivalent code that fully exploits the sparsity of all implicitly sparse matrices to reduce both the storage requirements as well as computational time of the original implementation.

Although more experiments and probably the development of more advanced transformations and strategies to control these transformations are required to determine whether a successful conversion is also feasible for large programs, these results already indicate that a sparse compiler can be very useful during development of a sparse algorithm. The experiments also indicate that the selection of a sparse storage scheme for an implicitly sparse matrix can have a dramatic impact on the performance of the resulting sparse code. Very efficient sparse code can be obtained if characteristics of the nonzero structure of each implicitly sparse matrix are already know and accounted for at compile-time. Although, in practice, these characteristics will not be known until run-time, this indicates that in principle the sparse compiler can be used to generate multi-version code for an algorithm, in which each version is suited for a particular class of sparse matrices. At run-time, the characteristics of the nonzero structure of each implicitly sparse matrix are determined by means of efficient automatic nonzero structure analysis, after which the outcome of this analysis determines which version is most appropriate.

We also have shown that this approach provides the sparse compiler with more opportunities to exploit implicit parallelism and we have made a cautious step towards incorporating sparsity preserving reordering methods in the generated sparse code. Although definitely more research is required to make effective use of all these ideas, the fact that the complexity of writing sparse codes can be reduced substantially by dealing with the sparsity of matrices at the compilation level rather than at the programming level seems to justify this new approach. Moreover, the fact that the sparse compiler can be used to transform an adjacency matrix representation into the more economical adjacency structure representation, thereby reducing the complexity of the algorithm accordingly (see section 9.2.6), reveals the potential of going beyond numerical applications and to provide compiler support for other kinds of data structure transformations as well.

11.2 Shortcomings of the Prototype Sparse Compiler

While experimenting with the current prototype sparse compiler, some severe shortcomings were encountered that should be fixed before this sparse compiler can provide a serious full alternative to explicitly dealing with sparsity at the programming level. In this section, we discuss these shortcomings.

A first shortcoming of the prototype sparse compiler is that, although for many operations that occur frequently in numerical programs, at least one obvious dense implementation exists that is handled appropriately by the sparse compiler, there are also some operations for which any obvious dense implementation becomes translated into extremely inefficient sparse code. Consider, for example, the following dense implementation of matrix transposition (found in any textbook on programming, see e.g. [12]), in which annotations are used to enforce selection of the LDU-scheme for the implicitly sparse matrix A with enveloping data structure A:

```
      INTEGER I, J
      REAL    A(N,N), TMP

C_SPARSE(A : _SPARSE( 1 - N <= I - J <=   - 1)(0,1)))
C_SPARSE(A : _DENSE ( 0     <= I - J <=     0)(1,1)))
C_SPARSE(A : _SPARSE( 1     <= I - J <= N - 1)(1,0))
      ...
      DO I = 1, N-1
        DO J = I+1, N
          TMP    = A(I,J)   ← true
          A(I,J) = A(J,I)   ← (I,J) ∈ E(A) ∨ (J,I) ∈ E(A)
          A(J,I) = TMP      ← true
        ENDDO
      ENDDO
```

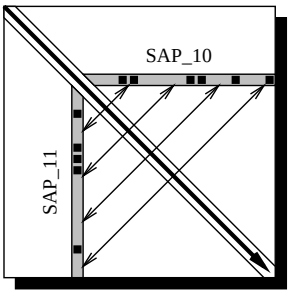


Figure 11.1: Inefficient Transposition

Because guard encapsulation is not applicable to this fragment, the sparse compiler decides to expand part of each `I`th row and column before the elements in these parts are interchanged, as illustrated in figure 11.1. Unfortunately, however, although the generated sparse code is semantically equivalent to the original dense code, the resulting code is embarrassingly inefficient because complete fill-in occurs for A, which clearly is an undesirable effect of transposing a sparse matrix:

```
DO I = 1, N-1
  CALL SSCT__(VAL_A, IND_A, LOW_A(I+N-1), HGH_A(I+N-1), SAP_11, SWT_11)
  CALL SSCT__(VAL_A, IND_A, LOW_A(I),     HGH_A(I),     SAP_10, SWT_10)
  DO J = I+1, N
    TMP = SAP_10(J)
    IF (.NOT.SWT_10(J)) THEN
      SWT_10(J) = .TRUE.
      CALL SINS__(VAL_A, IND_A, LOW_A, HGH_A, I,     NP_A, SZ_A, LST_A, L_, J)
    ENDIF
    SAP_10(J) = SAP_11(J)
    IF (.NOT.SWT_11(J)) THEN
      SWT_11(J) = .TRUE.
      CALL SINS__(VAL_A, IND_A, LOW_A, HGH_A, I+N-1, NP_A, SZ_A, LST_A, L_, J)
    ENDIF
    SAP_11(J) = TMP
  ENDDO
  CALL SGTH__(VAL_A, IND_A, LOW_A(I+N-1), HGH_A(I+N-1), SAP_11, SWT_11)
  CALL SGTH__(VAL_A, IND_A, LOW_A(I),     HGH_A(I),     SAP_10, SWT_10)
ENDDO
```

The inefficiency of this fragment is in strong contrast with the extremely efficient implementations of transposition that are possible for many sparse storage schemes. The LDU-scheme, for example, can simply be transposed by interchanging the pointers to the columns and rows of the strict lower and upper triangular part respectively. As already stated in chapter 4, efficient implementations to transpose a sparse matrix stored in a general sparse row-wise scheme can be found in [106][169, p236-239]. Similar problems are encountered for dense implementations in which, for example, row or column interchanges are explicitly implemented or where (parts of) matrices are copied to an implicitly sparse matrix. Hence, apparently there are some constructs that must be recognized explicitly by the sparse compiler to enable the generation of efficient sparse code.

Another problem is that for larger problems, conflicts may arise that cannot be resolved. Partly this is to blame on the way in which transformations have been implemented in the current prototype sparse compiler. For example, the reshaping method uses the unimodular framework, which is only applicable to perfectly nested sub-loops. For larger programs, it is more likely that unresolved conflicts remain. However, these are not the fundamental limitations of our approach, because more advanced transformations that widen the scope of application can be incorporated in a future implementation. More fundamentally is the strategy required to control the different program transformations, as is the case for conventional restructuring compilers. We leave finding such a strategy as an open problem.

Another drawback is that the prototype sparse compiler cannot handle existing sparse codes. Obviously, it would be very useful if a sparse compiler could provide some support to alter sparse storage schemes in such codes. Now, each algorithm must written as a dense algorithm first. Although this is much simpler than coding a sparse algorithm, this approach does not allow for reusing existing sparse codes.

Finally, the limited support to incorporate sparsity reordering methods currently provided by the prototype sparse compiler forms a fundamental shortcoming. This should be fixed in a future implementation, probably in combination with the ability to select more advanced data structures.

Despite all these shortcomings, we would like to stress that for many small fragments, the sparse compiler enabled us to rapidly obtain reasonably efficient sparse code which, if desired, could be further improved by hand. Consequently, even if not all shortcomings are solved in a future implementation, using a sparse compiler as a programming tool to develop new sparse algorithms seems to have some potential.

11.3 Related Work

As already discussed at the end of section 10.1, part of the compiler support for sparse matrix computations has been focused on the run-time concurrentization of loops in which the data dependence structure is determined by loop-invariant contents of arrays that are used as subscripts [126, 134][155][157, p115-121][160][170, p66-81][188, 189, 190, 204, 206]. Compiler support for cache optimization for band matrices is presented in [138].

Techniques required to make languages that facilitate data parallel programming also suitable for irregular and sparse computations have been studied in [205, 207, 208, 209]. In this approach, the compiler can automatically generate efficient code for distributed memory architectures. Since this issue has been left unexplored in this dissertation, no comparison can be given about the effectiveness of this approach in relationship with our approach. In contrast with our sparse compiler, however, the actual sparse storage schemes are not transparent to the programmer, but must be explicitly operated upon in original code, which complicates the development of the sparse application. Future efforts could be aimed at combining the advancements made in both areas.

In general, much effort has already been put in the development of sparse primitives. In chapter 9, some references to papers focusing on efficiently implementing particular sparse primitives have already been given. Some basic sparse operations are provided in the sparse extensions to BLAS [68, 69]. Other primitives are supplied in the basic tool-kit SPARSKIT [185]. Complete general-purpose packages to solve systems of linear equations are also available (e.g. MA18, MA27, MA28, SPARSPAK, Y12M [56, 78, 80, 81, 74, 94, 97, 169, 236, 164, 235]). The extremely good performance obtained in such packages for specific problems will make it very hard to automatically produce sparse code that is competitive with these applications. In case a package for a problem one wants to solve is already available, using a sparse compiler seems to be less attractive, although tailoring an application for one specific instance of a problem may offset performance disadvantages. In general, the main potential for sparse compilers here is to assist the development of new sparse primitives or complete packages.

The automatic conversion of a high level functional specification to an efficient implementation for sparse matrices also has been addressed. In [49], it is shown that a general realization of Gaussian elimination can be automatically converted into a tridiagonal solver by a number of development steps, where each step goes towards a lower level of abstraction. In [43, 88, 87], the observation has been made that programmers will attempt to exploit characteristics of both the *target architecture* as well as the *data being operated upon* (cf. chapter 1). Usually this implies that one particular algorithm is converted into several implementations, each of which is tailored for particular characteristics of the data and target architecture.

Hence, here the same question as explored in this dissertation arises, namely whether this conversion can be done automatically. The automatic conversion of a high level functional specification of a dense algorithm into an efficient implementation for several target architectures in which the sparsity of matrices is exploited, is presented in [88, 87]. Unlike our method, the programmer is still responsible for selecting a sparse storage scheme. On the other hand, in this approach advantage of the symmetry of matrices can also be taken. Moreover, high-level functional specifications do not suffer from the semantic gap between imperative programming and linear algebra that is inherent to our approach. Functional specifications allow for more freedom in code generation, making the transformations easier in general. Furthermore, being close to the underlying linear algebra makes it probably easier to incorporate reordering methods that permute the sparse matrices, an issue that has only been partly dealt with using awkward annotations in our sparse compiler. However, in [88] it is still reported that automatically incorporating reordering methods may be difficult.

Finally, others [125] have started to follow our approach of automatically converting a dense imperative program into sparse code by showing how a sequential implementation of Conjugate Gradient can be automatically converted into a sparse implementation of this algorithm that is suited for a distributed memory architecture.

11.4 Future Research

Future research should be focusing on improving the techniques presented in this dissertation and the development of strategies that control the different transformations of the automatic data structure selection and transformation method. Moreover, the ability to select more advanced data structures, possibly in combination with other representations of access patterns, supporting more complex access shapes and striding information should be incorporated. In particular, the sparse compiler must also be able to select sparse storage schemes for block forms, which enables the automatic generation of block algorithms.

To improve the efficiency of the generated sparse code in case some conflicts cannot be resolved, the compiler should be able to generate code that changes the storage scheme of an implicitly sparse matrix at run-time in between particular algorithms having inherently different access patterns through an implicitly sparse matrix. Generating multi-version code should also be addressed since, in practice, characteristics of the nonzero structure of implicitly sparse matrices will only be available at run-time. This may require the ability to parameterize the generated sparse code, however, since we must be ready to deal with, for example, band matrices with varying bandwidths.

Finally, separating symbolic and numerical processing, an approach frequently taken in sparse matrix computations, or automatically exploiting other properties like symmetry or the occurrence of many ones could be studied and incorporated in future implementations of sparse compilers.

Appendix A

A Brief Overview of Direct Methods

There are two different approaches to solve a system of linear equations $A\vec{x} = \vec{b}$. In **direct methods** (see e.g. [78, 102, 163, 173, 178, 216]), the system is converted into an equivalent system whose solutions are easier to determine by applying a number of elementary row or column operations (cf. section 2.2.3). A completely different approach is taken in **iterative methods** (see e.g. [53, 85, 163, 173, 210, 211, 213, 215, 216, 232]), where the number of operations required is not known in advance. In linear stationary iterative methods of the first degree, for example, a system $A\vec{x} = \vec{b}$ is solved by starting with some initial guess in $\vec{x}^{\,(0)}$, after which the next approximation of the solution is obtained as follows, where a non-singular **splitting matrix** M can be used:

$$M\vec{x}^{\,(k+1)} = (M - A)\vec{x}^{\,(k)} + \vec{b} \tag{A.1}$$

Application of this method is useful, if for each initial guess, the sequence $\vec{x}^{\,(0)}$, $\vec{x}^{\,(1)}$, $\vec{x}^{\,(2)}$ and so on converges to the real solution of the system, i.e. $\lim_{i\to\infty} \vec{x}^{\,(i)} = \vec{x}$. For $M = A$, the method is identical to a direct method. In general, however, a splitting matrix is used that can be easily inverted.

Although iterative methods play a very important role in solving sparse systems, because the storage requirements and roundoff errors can be reduced by only using elements of the original coefficient matrix, in this section we focus on direct methods. In particular, we discuss some issues related to the direct solution of dense, symmetric and sparse systems. Moreover, because using a direct method to solve a sparse system suffers from so-called fill-in, we discuss some sparsity preserving reordering methods that are used frequently in combination with direct methods.

A.1 Direct Methods for Systems of Linear Equations

In this section, we focus on direct methods to solve a system $A\vec{x} = \vec{b}$, where A is a square non-singular matrix (i.e. $\det(A) \neq 0$). These methods are based on the conversion of this system into an equivalent system with an upper triangular coefficient matrix.

A.1.1 Direct Methods for Dense Systems

First, we consider some direct methods for dense matrices.

Gaussian Elimination

A well-known direct method is **Gaussian elimination** (see e.g. [78, 102, 163, 173]), where the conversion is done in $n - 1$ successive stages.

At each stage $1 \leq k < n$, an equivalent system $A^{(k+1)}\vec{x} = \vec{b}^{(k+1)}$ is obtained from the previous system according to the following formulae, where $A^{(1)} = A$ and $\vec{b}^{(1)} = \vec{b}$. All other elements remain unaffected:

$$\begin{cases} a_{ij}^{(k+1)} = a_{ij}^{(k)} - (a_{ik}^{(k)}/a_{kk}^{(k)}) \cdot a_{kj}^{(k)} & \text{for } k < i \leq n, k \leq j \leq n \\ b_i^{(k+1)} = b_i^{(k)} - (a_{ik}^{(k)}/a_{kk}^{(k)}) \cdot b_k^{(k)} & \text{for } k < i \leq n \end{cases} \tag{A.2}$$

Here, we assume that at each stage, the **pivot** $a_{kk}^{(k)}$ is nonzero. The variable x_k is eliminated from the equations $k+1, \ldots, n$ by application of elementary row operations nullifying all elements below the diagonal in the kth column. All elements involved in these computations constitute the so-called $(n-k+1) \times (n-k+1)$ **active sub-matrix** at stage k.

In the final system $A^{(n)}\vec{x} = \vec{b}^{(n)}$, matrix $A^{(n)}$ is an upper triangular matrix U and the system $U\vec{x} = \vec{b}^{(n)}$ can be solved easily by back substitution, explained below in more detail. Back substitution is not required if A is converted into diagonal form by **Gauss-Jordan elimination**. In this case, each component of the solution vector can be computed directly. Gauss-Jordan elimination can also be used to compute A^{-1} in-place [173].

LU-Factorization

A variant is formed by **LU-factorization** (see e.g. [78, 97, 169, 173, 175, 235]). This method consists of computing a triangular factorization $A = LU$, in which U is the upper triangular matrix arising from Gaussian elimination, and L is a unit lower triangular matrix that represents all elementary row operations. In particular, each stage of Gaussian elimination can be expressed as $A^{(k+1)} = L^{(k)}A^{(k)}$, for $1 \leq k < n$, where $L^{(k)}$ represents the elimination at that stage as follows using the multipliers $l_{ik} = a_{ik}^{(k)}/a_{kk}^{(k)}$:

$$L^{(k)} = \begin{pmatrix} 1 & & & & \\ & \ddots & & & \\ & & 1 & & \\ & & -l_{k+1,k} & & \\ & & \vdots & \ddots & \\ & & -l_{n,k} & & 1 \end{pmatrix} \tag{A.3}$$

Since $U = L^{(n-1)} \ldots L^{(1)}A$, the following factorization of A is obtained:

$$A = (L^{(1)})^{-1} \ldots (L^{(n-1)})^{-1}U = LU$$

The inverse of each lower column matrix $L^{(k)}$ is simply obtained by negation of all the off-diagonal elements. Moreover, the product $(L^{(1)})^{-1} \ldots (L^{(n-1)})^{-1}$ is a lower triangular matrix in which the off-diagonal elements in the kth column are the multipliers in $(L^{(k)})^{-1}$. Consequently, the matrix L is in unit lower triangular form. The storage that is originally used to store A can be used to store both the matrices L and U, i.e. the factorization can be computed in-place, if the *unit* diagonal of L is stored implicitly. Each eliminated element $a_{ik}^{(k+1)}$ in $A^{(k+1)}$ for $k < i \leq n$ is simply replaced by the multiplier l_{ik}.

An alternative formulation of LU-factorization arises from the following factorization of a partitioned matrix A, where $H_1 = H - \frac{1}{d} \cdot \vec{v}\,\vec{u}^{\,T}$ and we assume that $d \neq 0$:

$$A = \begin{pmatrix} d & \vec{u}^{\,T} \\ \vec{v} & H \end{pmatrix} = \begin{pmatrix} 1 & 0\cdots 0 \\ \frac{1}{d}\cdot\vec{v} & I \end{pmatrix} \begin{pmatrix} d & 0\cdots 0 \\ 0 & \\ \vdots & H_1 \\ 0 & \end{pmatrix} \begin{pmatrix} 1 & \frac{1}{d}\cdot\vec{u}^{\,T} \\ 0 & \\ \vdots & I \\ 0 & \end{pmatrix} \tag{A.4}$$

A recursively obtained factorization $H_1 = L_1 D_1 U_1$ gives rise to the factorization $A = LDU$ in which *both* factors L and U are unit triangular and D is in diagonal form:

$$A = \begin{pmatrix} 1 & 0\cdots 0 \\ \frac{1}{d}\cdot\vec{v} & L_1 \end{pmatrix} \begin{pmatrix} d & 0\cdots 0 \\ 0 & \\ \vdots & D_1 \\ 0 & \end{pmatrix} \begin{pmatrix} 1 & \frac{1}{d}\cdot\vec{u}^{\,T} \\ 0 & \\ \vdots & U_1 \\ 0 & \end{pmatrix} = LDU$$

This defines either the factorization $A = (LD)U$ or $A = L(DU)$, in which only one factor is unit triangular. The latter factorization corresponds to the factorization presented above, whereas the former factorization results if before each stage an elementary row operation is applied to obtain a unit pivot. In this case, each stage can be expressed as $A^{(k+1)} = L^{(k)} D^{(k)} A^{(k)}$, for $1 \leq k \leq n$, where each $n \times n$ matrix $D^{(k)}$ represents the previous described normalization. Since $L^{(k)}$ represents the elimination at stage k, we have $L^{(n)} = I$.

Forward and Back Substitution

After the factorization $A = LU$ has been computed (where L is in lower unit triangular form), the system $LU\vec{x} = \vec{b}$ is solved in two steps. First, $L\vec{c} = \vec{b}$ is solved by **forward substitution**, where the components of $\vec{c}$ are computed for $i = 1, \ldots, n$:

$$c_i = b_i - \sum_{k=1}^{i-1} l_{ik} \cdot c_k \tag{A.5}$$

The remaining system $U\vec{x} = \vec{c}$ is solved by **back substitution**, where the components of the solution $\vec{x}$ are computed for decreasing values $i = n, \ldots, 1$:

$$x_i = (c_i - \sum_{k=i+1}^{n} u_{ik} \cdot x_k) \,/\, u_{ii} \tag{A.6}$$

The advantage of LU-factorization over other direct methods is that if we want to solve the system for multiple right-hand side vectors (i.e. solving $AX = B$), these right-hand sides do not have to be known in advance [78, 200]. All forward operations are simply recorded in the factor L *during* the factorization, so that these operations can be delayed until a new right-hand side vector becomes available. LU-factorization can also be used to compute A^{-1} by solving $AY = I$, or to determine the determinant of A. Moreover, LU-factorization has a lower operation count than Gauss-Jordan elimination for solving a system of linear equations, and performs equally well for inverting a matrix if the special form of the right-hand side in $AY = I$ is accounted for [173].

Pivoting for Stability

Row or column interchanges with a later row or column may be required during each stage of LU-factorization for two reasons. First, in some cases a row interchange is necessary to obtain a nonzero pivot (if such a pivot cannot be obtained, the matrix is singular).

The second reason for interchanging is due to the fact that representation errors for real numbers and inexact computer arithmetic may cause a loss of accuracy. Consequently, effectively a factorization $LU = A + H$ is obtained for some **perturbation matrix** H. Only if H is relatively small in comparison with A, the algorithm used to obtain the factorization is considered **stable**.

Stability can be controlled by limiting the growth of elements in the matrix during factorization. Based on the bounds of elements of H derived in [21, 177], we can limit this growth by enforcing the inequalities $|l_{ik}| \leq 1$ for all $k \leq i \leq n$ at each stage k. This can be achieved by application of **partial pivoting**, where at each stage a row interchange is performed if necessary to enforce the following inequality:

$$|a_{kk}^{(k)}| \geq \max_{k \leq i \leq n} |a_{ik}^{(k)}|$$

Alternatively, a column interchange can be used to obtain a pivot with the largest absolute value in the kth row of the upper triangular part, enforcing the inequality $|a_{kj}^{(k)}/a_{kk}^{(k)}| \leq 1$ for all $k \leq j \leq n$. Although partial pivoting yields stable factorizations in practice, the growth of elements can be further limited by the use of **complete pivoting**, where row and column interchanges may be applied to obtain a pivot with the maximum absolute value in the whole active sub-matrix.

Mathematically, there is no difference between applying all row and column interchanges to the whole compact storage containing elements of both factors *during* factorization, or permuting A into PAQ *before* the factorization, where the permutation matrices P and Q represent the accumulated effects of all row and column interchanges respectively [78, p299]. This permutation is called a **symmetric permutation** if $P = Q^T$ holds. However, because row and column interchanges correspond to rearranging the equations and relabeling the components of $\vec{x}$ respectively, these changes must be accounted for in the solution. We write the original system $A\vec{x} = \vec{b}$ as follows:

$$(PAQ)Q^T\vec{x} = (LU)Q^T\vec{x} = P\vec{b}$$

This permuted system is solved by forward substitution of $L\vec{c} = P\vec{b}$, followed by back substitution of $U\vec{y} = \vec{c}$ and permuting the resulting vector according to $\vec{x} = Q\vec{y}$.

Iterative Improvement

Even if pivoting is used by a direct method to preserve stability, inexact computer arithmetic is responsible for the fact that, in general, we still may obtain the factorization $PAQ + H = LU$ for some perturbation matrix H. Consequently, if this inaccurate factorization is used to solve a system of linear equations $A\vec{x} = \vec{b}$ as $(LU)Q^T\vec{x} = P\vec{b}$, then usually the computed vector $\vec{x}$ differs from the real solution. We can improve the accuracy of the computed solution by a method called **iterative improvement** or **iterative refinement** (see e.g. [78, 102, 173, 235]).

Starting with the computed solution in a column vector $\vec{x}^{\,(0)}$, at each step k the residual vector $\vec{r}^{\,(k)} = \vec{b} - A\vec{x}^{\,(k)}$ is determined. If the current solution differs from the real solution $\vec{x}$, then this residual vector is nonzero. In this case, we can write the real solution as $\vec{x} = \vec{x}^{\,(k)} + \vec{\delta}^{\,(k)}$ for some unknown correction vector. Moreover, because $A\vec{x} = A(\vec{x}^{\,(k)} + \vec{\delta}^{\,(k)})$ is equal to $\vec{b}$, the following equation holds:

$$A\vec{\delta}^{\,(k)} = \vec{b} - A\vec{x}^{\,(k)} = \vec{r}^{\,(k)}$$

Because a factorization of A is available, the correction vector can be computed by solving this system as $(LU)Q^T\vec{\delta}^{\,(k)} = P\vec{r}^{\,(k)}$. Subsequently, we can improve the solution as follows:

$$\vec{x}^{\,(k+1)} = \vec{x}^{\,(k)} + \vec{\delta}^{\,(k)}$$

This method can be repeated if desired until some criterion has been satisfied. During this process, it is essential to use more precision for the residual vector. Because both the original matrix A and the factors L and U must be kept in memory, the use of iterative refinement increases the storage requirements of the solution method while additional computational time is required to perform the iterations.

A.1.2 Direct Methods for Symmetric Systems

While solving a system $A\vec{x} = \vec{b}$ with a symmetric coefficient matrix, i.e. $A = A^T$, symmetry is preserved in the factorization if at any stage a suitable pivot can be chosen from the diagonal in the active sub-matrix. This form of pivoting, referred to **diagonal pivoting**, yields the factorization of a symmetric permutation $PAP^T = LDL^T$. In this manner, the storage requirements and operation count of the factorization method are reduced by only computing L and D.

Choleski Factorization

A symmetric matrix A is called **positive definite** if $\vec{x}^T A\vec{x} > 0$ holds for all $\vec{x} \neq 0$, **negative definite** if $\vec{x}^T A\vec{x} < 0$ holds for all $\vec{x} \neq 0$, and **indefinite** otherwise. For a symmetric positive definite matrix A, each diagonal element of D in the factorization $A = LDL^T$ is positive and the factorization can be written as $(LD^{\frac{1}{2}})(D^{\frac{1}{2}}L^T) = \tilde{L}\tilde{L}^T$. This **Choleski factorization** can be obtained as shown below:

$$A = \begin{pmatrix} d & \vec{v}^T \\ \vec{v} & H \end{pmatrix} = \begin{pmatrix} \sqrt{d} & 0\cdots 0 \\ \frac{1}{\sqrt{d}}\cdot\vec{v} & I \end{pmatrix} \begin{pmatrix} 1 & 0\cdots 0 \\ 0 & \\ \vdots & H_1 \\ 0 & \end{pmatrix} \begin{pmatrix} \sqrt{d} & \frac{1}{\sqrt{d}}\cdot\vec{v}^T \\ 0 & \\ \vdots & I \\ 0 & \end{pmatrix}$$

In this factorization, $H_1 = H - \frac{1}{d}(\vec{v}\cdot\vec{v}^T)$ is again a symmetric positive definite matrix [97, 169]. After the factorization $H_1 = L_1 L_1^T$ has been obtained recursively, the Choleski factorization of A is defined as follows:

$$A = \begin{pmatrix} \sqrt{d} & 0\cdots 0 \\ \frac{1}{\sqrt{d}}\cdot\vec{v} & L_1 \end{pmatrix} \begin{pmatrix} \sqrt{d} & \frac{1}{\sqrt{d}}\cdot\vec{v}^T \\ 0 & \\ \vdots & L_1^T \\ 0 & \end{pmatrix} = \tilde{L}\tilde{L}^T$$

Given the Choleski factorization $A = \tilde{L}\tilde{L}^T$, the solution of a system $A\vec{x} = \vec{b}$ is determined by subsequently solving the systems $\tilde{L}\vec{c} = \vec{b}$ and $\tilde{L}^T\vec{x} = \vec{c}$.

Note that a symmetric positive definite matrix can be factorized *without any pivoting*. Another important class of matrices for which pivoting is not required to control the stability consists of **diagonally dominant** matrices, which are matrices for which the following condition holds for all diagonal elements and at least one of the inequalities is strict:

$$|a_{ii}| \geq \sum_{j\neq i} |a_{ij}|$$

Hence, if a diagonal dominant matrix is also symmetric, the symmetry can be easily preserved in the factorization. Furthermore, because symmetric permutations preserve both properties described above (i.e. if A is a symmetric positive definite matrix or diagonal dominant matrix, then PAP^T is also a symmetric positive definite or diagonal dominant matrix), diagonal pivoting is stable.

Consequently, although pivoting is not required for stability, diagonal pivoting can be used to preserve the sparsity of symmetric positive definite and diagonal dominant matrices, as explained in more detail in following sections. For indefinite symmetric matrices, however, the use of diagonal pivoting may yield an unstable factorization method. Since the use of e.g. partial pivoting would destroy symmetry, in these cases the notion of a pivot is frequently extended to 2×2 blocks to preserve the symmetry [46, 78].

A.1.3 Direct Methods for Sparse Systems

In many scientific and engineering problems, a system $A\vec{x} = \vec{b}$ must be solved, where A is a large and sparse matrix. The storage requirements and computational time of direct methods may be reduced substantially by exploiting the sparsity of A, although additional nonzero elements may appear during the factorization.

Fill-In

Usually, LU-factorization is used to solve a sparse system $A\vec{x} = \vec{b}$, because both factors L and U in $A = LU$ remain reasonable sparse, whereas A^{-1} is rather dense in general [78, 200]. However, even during LU-factorization, some zero elements in the active sub-matrix become nonzero, i.e. **fill-in** occurs, as illustrated in figure A.1. Usually, we ignore *exact* cancellation, which occurs if subtracting two entries yields a zero, because this is only likely to occur frequently for special matrices, such as matrices with many ones.

In general, given a factorization $PAQ = LU$, we define the **filled matrix** as $L+U$ (defined as $L+L^T$ for the Choleski factorization $PAP^T = LL^T$). Obviously, $\mathit{Nonz}(PAQ) \subseteq \mathit{Nonz}(L+U)$ holds and the index set of all elements caused by fill-in is shown below:

$$\mathit{Fill}(PAQ) = \mathit{Nonz}(L+U) - \mathit{Nonz}(PAQ)$$

Because the sparse storage scheme for A is usually overwritten with the elements in the factors L and U, eventually sufficient storage must be available to store the entries of the filled matrix. Hence, usually a dynamic storage scheme must be used. However, if $\mathit{Nonz}(L+U)$ is known in advance (e.g. for (variable) band matrices without pivoting) or can be (conservatively) predicted at run-time before the data structure is initialized with **symbolic factorization** [78, 97, 131, 169, 182], then a static storage scheme can be used for the sparse matrix A. For example, in [99] symbolic factorization in case partial pivoting will be used is simply based on taking the union of the nonzero structures of all potentially target rows.

Example: In figure A.2, the nonzero structures of the test matrix $D(20, 5)$ [164, 235] and the corresponding filled matrix arising from LU-factorization without pivoting are shown.

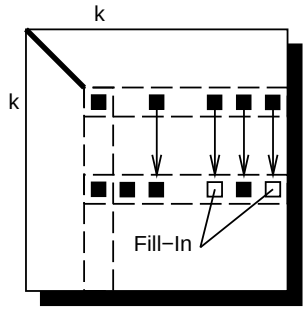


Figure A.1: $a_{ij}^{(k)} = 0$, whereas $a_{ij}^{(k+1)} \neq 0$ because $a_{ik}^{(k)} \neq 0$ and $a_{kj}^{(k)} \neq 0$

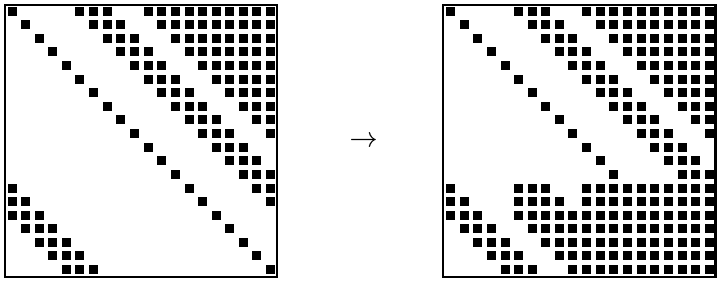


Figure A.2: Original and Filled Matrix

This example clearly illustrates that usually a dense active sub-matrix is operated on in the last stages of the factorization. Hence, to avoid the computational overhead that is inherent to sparse codes, in some codes a switch to dense storage of the active sub-matrix is made towards the end of the factorization (see e.g. [76, 78]). This switch is performed as soon as the density of the active sub-matrix exceeds a certain threshold depending on characteristics of the target machine.

Graph Representation of Nonzero Structures

The nonzero structure of a sparse $n \times n$ matrix A can be represented by a directed graph $G = (V, E)$, where $V = \{v_1, \ldots v_n\}$ is a finite set of vertices which are labeled with the integers $1, \ldots, n$. If v_i denotes the vertex in V with label i, then v_i is associated with the ith row and column, and there is an edge $(v_i, v_j) \in E$ if and only if $a_{ij} \neq 0$ holds [72, 78, 97, 108, 169]. If a matrix, or at least its nonzero structure, is symmetric, then $(v_i, v_j) \in E$ implies that we also have $(v_j, v_i) \in E$, and an undirected graph can be used to represent the nonzero structure.

Since the transversal is usually full, self-cycles caused by these elements are mostly omitted from the graph. Examples of an undirected and directed graph associated with a unsymmetric and symmetric matrix are given in figure A.3. If we ignore the numerical values, the sparse matrices precisely form the adjacency matrices of the associated graphs.

Elimination Graphs

Graphs provide an alternative view on operations on a sparse matrix that change the nonzero structure. Gaussian elimination, for instance, can be interpreted as the generation of a sequence of **elimination graphs** [78, 97, 167, 169]. We start with $G_1 = (V_1, E_1)$, representing the nonzero structure of the original matrix A. Assuming that vertex v_k is eliminated at stage k, elimination graph $G_{k+1} = (V_{k+1}, E_{k+1})$ is obtained from $G_k = (V_k, E_k)$ by removing vertex v_k and all incident edges, followed by addition of edge (v, w) for every $(v, v_k) \in E_k$ and $(v_k, w) \in E_k$ where $(v, w) \notin E_k$.

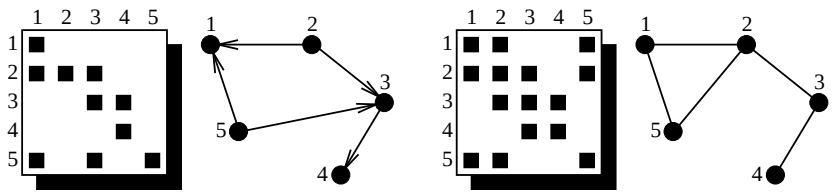


Figure A.3: Nonzero Structures with Associated Graphs

Hence, $V_{k+1} = V_k - \{v_k\}$ and $E_{k+1} = \{(v,w) \in E_k | v \in V_{k+1} \wedge w \in V_{k+1}\} \cup D_k$, where for a graph G_k and a vertex v_k, the **deficiency** D_k is defined as follows:

$$D_k = \{(v,w) | (v,v_k) \in E_k \wedge (v_k,w) \in E_k \wedge (v,w) \notin E_k \wedge v \neq w\}$$

The addition of each edge corresponds to fill-in, as illustrated in figure A.4. We assume that at the first stage, entry a_{11} is used as pivot so that rows 3 and 4 become the target rows. Due to the elimination, elements a_{32} and a_{42} become nonzero. This is represented by the addition of edges $(3,2)$ and $(4,2)$ to the graph associated with this matrix. In this manner, a sequence of elimination graphs $G_1, G_2, \ldots, G_n$ is obtained where, ignoring the possibility of exact cancellation, each graph G_k represents the nonzero structure of the $(n-k+1) \times (n-k+1)$ active sub-matrix considered at stage k. If all edges that are added during this process are also added to the graph associated with the original matrix, the **filled graph** $G_F = (V, E_F)$ is obtained, representing the nonzero structure of the filled matrix.

Modeling Gaussian elimination as a sequence of graphs transformations is simple, but has as disadvantages that dynamic data structures are required for the elimination graph with unpredictable storage requirements. Therefore, in [96, 97] an *implicit* model is discussed for the elimination graphs of symmetric matrices, as opposed to the *explicit* elimination graph model described above. This implicit model is based on the observation that if $x \in V_k$ is a vertex in an elimination graph G_k, and the set of vertices $S = \{v_1, \ldots, v_{k-1}\}$ has been eliminated, the set of vertices that are adjacent to this vertex x is described by *Reach*(x, S). The latter set is called a **reachable set** and consists of all vertices $y \notin S$ for which there is a path $(x, w_1, \ldots, w_l, y)$ in the graph G_1 associated with the *original* matrix, where all $w_i \in S$ and l may be zero. For $i < j$, both (v_i, v_j) and (v_j, v_i) are edges in the filled graph if and only if $v_j \in$ *Reach*$(v_i, \{v_1, \ldots, v_{i-1}\})$. For $i > j$, the roles of i and j must be interchanged. This implies that the nonzero structure of the filled matrix can be described in terms of the nonzero structure of the original matrix.

Because computing a reachable set can be expensive, a third model for elimination graphs is considered in [97, 96]. Here, the elimination is modeled as a sequence of **quotient graphs**, defined by partitions on the vertex set. A partition P on the vertex set of a graph $G = (V, E)$ consists of a number of subsets of V, i.e. $P = \{S_0, \ldots, S_p\}$, where $S_i \subseteq V$, such that $S_i \cap S_j = \emptyset$ if $i \neq j$ and $\bigcup_{i=0}^{p} S_i = V$. The corresponding quotient graph is obtained by collapsing the vertices in each set S_i into so-called composite vertices. There is an edge between two composite vertices S_i and S_j if and only if $(v,w) \in E$ for some $v \in S_i$ and $w \in S_j$. By collapsing adjacent vertices that have been eliminated into one composite vertex, no more storage than for the first graph is required, whereas the reachable sets can be generated more efficiently.

Finally, a method to model Gaussian elimination by successively *adding* vertices to a graph (rather than successively eliminating vertices) is presented in [131].

Permutations

A symmetric permutation of a matrix corresponds to a relabeling of vertices in the associated graph, as illustrated in figure A.5 for the first matrix of figure A.3.

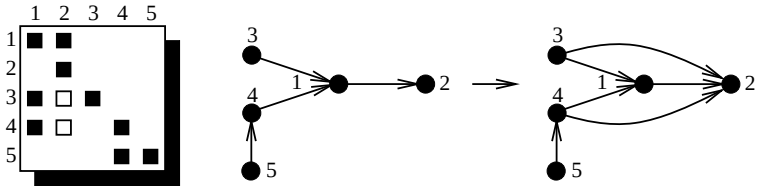


Figure A.4: Addition of Edges

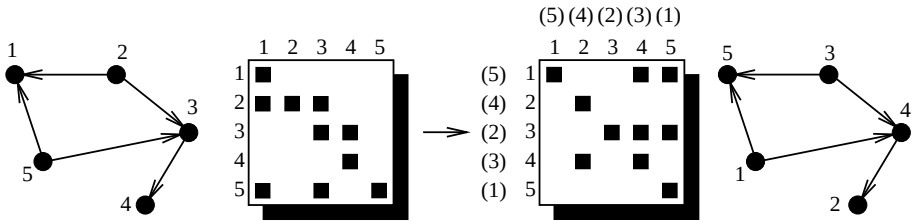


Figure A.5: Symmetric Permutation

The nonzero structure of the permuted matrix is represented by the same digraph as associated with the original matrix in which the vertices are relabeled according to the symmetric permutation. On the other hand, any relabeling of the vertices in the a graph associated with a matrix A induces a symmetric permutation PAP^T of that matrix. Therefore, the problem of finding a symmetric permutation of a sparse matrix satisfying certain requirements can usually be expressed in terms of finding a particular labeling of the vertices in the associated graph.

Unsymmetric permutations, i.e. permutations PAQ where $P \neq Q^T$, may alter the structure of the graph representing the nonzero structure of matrix A. This has motivated the use of **bipartite graphs**. For an $n \times n$ matrix, a set of $2 \cdot n$ vertices is partitioned into $R = (r_1, \ldots, r_n)$ and $C = (c_1, \ldots, c_n)$ associated with the rows and columns respectively. There is an edge from vertex $r_i \in R$ to vertex $c_j \in C$ if and only if $a_{ij} \neq 0$. The nonzero structure of the matrix resulting after any unsymmetric permutation is represented by the same bipartite graph in which the row and column vertices are relabeled accordingly. For example, in figure A.6, an unsymmetric permutation is applied to a matrix by interchanging rows 3 and 4 and columns 1 and 3. The nonzero structure of the resulting matrix is represented by the bipartite graph resulting after application of the same interchanges to the labels of the vertices representing the rows and columns respectively, as illustrated in the same figure.

A.2 Sparsity Preserving Reordering Methods

An important observation in the solution of sparse systems of linear equations is that the factorization of a permuted system may induce less fill-in than the factorization of the original system. Methods that rearrange the equations and relabel the variables in order to preserve sparsity are called **reordering methods**.

A.2.1 Reordering Methods

Although $|Nonz(A)| = |Nonz(PAQ)|$ holds for arbitrary permutation matrices P and Q, the total number of nonzero elements in the factors of $A = LU$ and $PAQ = \tilde{L}\tilde{U}$ may differ.

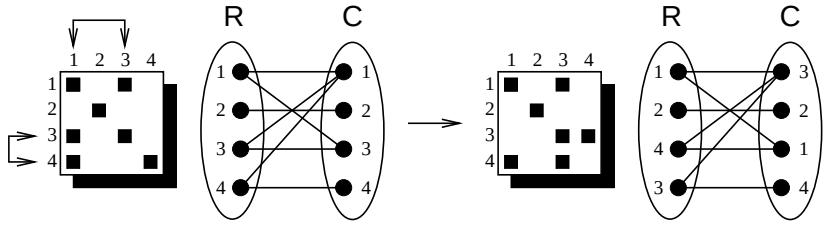


Figure A.6: Unsymmetric Permutation

Ideally, we would like to have a reordering method that determines permutation matrices for which the minimum number of nonzero elements in the filled matrix results. In this manner, we would minimize the amount of fill-in, which may be difficult to deal with, whereas preserving sparsity reduces storage requirements, computational time and may even improve the accuracy of the computed solution by diminishing the effects of accumulated errors [169, 198, 200]. However, even finding a symmetric permutation satisfying this **minimum fill-in objective** is NP-complete for both unsymmetric and symmetric matrices [182, 231]. This implies that it is computational infeasible to determine a permutation inducing the least amount of fill-in.

Therefore, in practice only heuristics are used yielding an acceptable but not necessarily optimal reduction of fill-in. Two different kind of heuristics can be distinguished [78], which are discussed in more detail in the following section:

1. Local Strategies: At each stage of the elimination, a pivot is selected minimizing some local objective related to sparsity.

2. A Priori Reordering Methods: Before the elimination, the matrix is permuted into a form in which zero elements are isolated, thereby confining fill-in to particular regions in the matrix.

Local strategies are useful in combination with methods exploiting *all* zero elements in a matrix, whereas a priori methods are frequently used to permute a matrix into a form in which fill-in is confined to particular regions. From a mathematical point of view, however, there is no difference between the two methods, because in both cases the computed factorization can eventually be expressed as $PAQ = LU$.

Because, frequently, a particular system must be solved for several right-hand side vectors and in some cases we must even solve several systems having the same nonzero structure, the solution method can roughly be divided into the phases ANALYZE/FACTORIZE/SOLVE [56, 73, 74, 81, 78, 80, 94, 164, 236], which are implemented separately.

In the first phase a good ordering is determined, followed by the actual factorization in the second phase, and computation of the solution in the final phase. If several systems having the same nonzero structure must be solved, more time can be spent in the first phase, because the costs of the analysis can be amortized over all subsequent factorizations. In case the use of diagonal pivoting yields a stable method, a symbolic factorization operating on the nonzero structure only is performed in the first phase. If, on the other hand, the ordering depends on actual numerical values, a factorization is obtained as side-effect of the first phase, called ANALYZE-FACTORIZE in this case. Now, we can still use the same ordering for subsequent factorizations, although we must monitor the stability and repeat the analysis phase in case the method is unstable. In all cases, once a factorization has been obtained, we can repetitively execute the last phase to solve the system for many right-hand sides.

A.2.2 Local Strategies

If we use a local strategy to preserve sparsity, the stability constraints arising from partial or complete pivoting are usually too restrictive with respect to the pivot selection.

A pivot which preserves the stability the best, may induce an unacceptable amount of fill-in, so that a trade-off between maintaining stability and sparsity arises [177, 202, 235]. We can increase the size of the candidate pivot set at the expense of a potential loss of accuracy by using so-called **threshold pivoting**. While factorizing a sparse matrix A, at each stage k we permute an element in the active sub-matrix satisfying the following row-wise oriented criterion for a fixed $0 < u \leq 1$ to the position of the pivot:

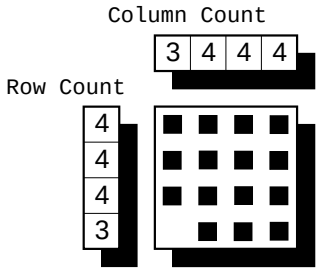


Figure A.7: Markowitz cost $m > 4$ for a matrix A

$$|a_{ij}^{(k)}| \geq u \cdot \max_{k \leq l \leq n} |a_{il}^{(k)}| \tag{A.7}$$

Alternatively, a similar column-wise oriented criterion can be used. Which of these two criteria is used depends on the storage scheme of A, since usually we can only access the entries in either a row or column efficiently. Threshold pivoting resembles complete pivoting in the sense that row *and* column interchanges are applied. It also resembles partial pivoting in the sense that stability constraints are concerned with only a column or row in the active sub-matrix. Based on experimentation, a value $u \approx 0.1$ seems appropriate, although this value may be different for specific problems.

Markowitz Strategy

If we use $r_i^{(k)}$ and $c_j^{(k)}$ to denote the number of entries appearing at stage k in row i and column j in the $(n-k+1) \times (n-k+1)$ active sub-matrix respectively, then the **Markowitz cost** of an element $a_{ij}^{(k)}$ in the active sub-matrix is defined as follows:

$$(r_i^{(k)} - 1) \cdot (c_j^{(k)} - 1) \tag{A.8}$$

Examination of the formulation (A.4) of LU-factorization reveals that, after possibly a suitable pivot has been brought into position, the next active sub-matrix is updated with $-\frac{1}{d}\vec{v}\vec{u}^{\,T}$. Because the Markowitz cost of an element is equal to the number of nonzero elements in the corresponding updating matrix, a pivot having minimum Markowitz cost modifies the least elements. Furthermore, the Markowitz cost provides an upper bound of the amount of fill-in that may occur.

These observations give rise to the **Markowitz strategy** [78, 150, 169, 215, 200], in which at each stage k, we use one of the elements in the active sub-matrix satisfying a row- or column-wise stability criterion for which the Markowitz cost is minimized as pivot. Note that, if we denote the Markowitz cost of such an element by m, it is possible that the following inequality holds, because there is no (numerically acceptable) entry in the intersection of the rows and columns having a minimum number of entries [78, 235]:

$$m \;>\; \min_{k \leq i \leq n} (r_i^{(k)} - 1) \cdot \min_{k \leq j \leq n} (c_j^{(k)} - 1)$$

In figure A.7, for example, the minimum row and column count is $r_4^{(1)} = 3$ and $c_1^{(1)} = 3$ respectively. However, because $a_{41} = 0$, an element having Markowitz cost $m = (4-1) \cdot (4-1) = 9 > 4$ will be used at the first stage.

To prevent the situation in which computational savings arising from fill-in reduction are outweighed by the costs of finding a suitable pivot, in [56, 80, 78] the following mechanism is used.

Rows and columns are ordered in increasing entry count in a data structure allowing for an efficient update at each stage. Rows and columns are scanned in increasing order of entry count, taking rows before columns. The search can be terminated as soon as a stable element is encountered with a Markowitz cost that does not exceed $(r_i - 1)^2$ while scanning a row with count r_i, or that does not exceed $(c_j - 1) \cdot c_j$ while scanning a column with count c_j. This mechanism may fail to find a suitable pivot quickly. In case $c_j^{(k)} > r_i^{(k)}$ holds for many elements satisfying the stability constraints, it is possible that a lot of these elements are searched before the termination criterion described above holds [235]. Furthermore, there may be many elements with minimum Markowitz costs that do not satisfy the stability constraints.

Therefore, the mechanism proposed in [164, 235, 236] only performs a search of a few rows in increasing number of nonzero elements. Furthermore, *all* elements in these rows are considered and the element which is the largest in absolute value of all suitable elements is used as pivot. Limiting the search for a suitable pivot yields an unacceptable increase of the amount of additional fill-in in practice, reduces the pivotal search time substantially, and even may yield more accurate solutions [235].

Minimum Degree Method

A symmetric version of the Markowitz strategy, useful for symmetric positive definite matrices for which pivoting is not required for stability and diagonal pivoting may be used to preserve sparsity (see section A.1.2), is called the **minimum degree method** [52, 78, 97, 200]. At each stage k, a symmetric permutation is applied, enforcing the following condition for the pivot $a_{kk}^{(k)}$:

$$r_k^{(k)} \leq \max_{k \leq i \leq n} r_i^{(k)}$$

Because, effectively, at each stage we use a pivot corresponding to a vertex in the elimination graph with minimum degree, the minimum degree strategy is independent of any numerical values. Therefore, the ordering can be computed symbolically beforehand once the nonzero structure of the matrix is known. The ordering can be used for several systems with the same nonzero structure. In [96, 97, 98, 144], various implementations of the minimum degree method are discussed (yielding a symbolic factorization as side-effect).

No fill-in occurs with this strategy in case the original graph is a tree, since only the leaf vertices and possibly the root vertex have minimum degree and elimination of one of these vertices yields a new tree without the introduction of additional edges. In general, if we apply a symmetric permutation to such a matrix according to a monotone labeling of the associated graph, where each child appears before its parent in the labeling, then Gaussian elimination proceeds without any fill-in [78, 167, 169].

Other Local Strategies

Simplifications of the Markowitz strategy, such as the **min.row in min.column strategy** [78, 169], where a pivot with minimum row count in the column with minimum column count is selected, usually induce too much fill-in to be practical. On the other hand, gains arising from the reduction of fill-in in the more complex strategies are usually diminished by the increase in search time. Such a complex strategy is formed by the the **minimum deficiency strategy** [57, 78, 200, 215], where a pivot is used at each stage with a minimum size of the deficiency, locally minimizing the amount of fill-in. However, even this strategy does not necessarily minimize the total amount of fill-in. Hence, because the Markowitz strategy is relatively simple to implement and yields a satisfactory reduction of fill-in, this strategy has been most successful in practice.

A.2.3 Unsymmetric A Priori Reordering Methods

We call a matrix A **bi-reducible** if there is a permuted matrix PAQ with a *non-trivial* partition into block triangular form and **reducible** in case this matrix can be expressed with $P = Q^T$ [108]. Furthermore, we will call a matrix **fully reduced** if it has a partition into block triangular form in which all diagonal blocks are irreducible.[1] An important way to confine fill-in while solving a system $A\vec{x} = \vec{b}$ is to fully reduce the matrix A initially.

A permutation achieving this goal can be found in two steps [78]. First, a permutation matrix $\tilde{P}$ is determined such that the matrix $\tilde{P}A$ has a full transversal, i.e. all elements along the diagonal are nonzero. Subsequently, a symmetric permutation $P(\tilde{P}A)P^T$ is applied to obtain a block lower triangular matrix.

A full transversal is constructed by applying one of the variants of the **algorithm of Hall** [78, 169]. In this algorithm, at each step k some row permutations are applied extending the transversal by one, while preserving the nonzero elements on the first $k-1$ diagonal positions. Eventually, for all matrices that are not **symbolically singular** [78, 169], a row permutation $\tilde{P}$ results such that all elements on the diagonal in $\tilde{P}A$ are nonzero (in contrast, even some singular matrices can have a full transversal). Alternatively, we can use the **algorithm of Hopcroft and Karp**, which operates on bipartite graphs [169]. Although this algorithm has a lower time complexity than the algorithm of Hall, the latter performs better in practice.

One way to obtain a block lower triangular form is based on the following method to permute a matrix of which the nonzero structure is represented by an *acyclic* directed graph $G = (V, E)$ into lower triangular form. First, we label all vertices $v \in V$ with a zero out-degree and eliminate all incident edges $(w, v) \in E$. Vertices of which the out-degree becomes zero are labeled next. This process is repeated until all vertices have been labeled. If we apply the symmetric permutation induced by this relabeling to the matrix, then a lower triangular matrix results (alternatively, we could relabel all vertices according to a topological sort of the graph, so that an upper triangular matrix results). For arbitrary digraphs, this method can be applied at block level to the composite vertices in the acyclic condensation of the graph [169, 234], which is the quotient graph defined by the partition of the digraph into strongly connected components. If we relabel all vertices in V according to the resulting labeling of strongly connected components, where all vertices appearing in one strongly connected component may appear in arbitrary order, then a *block* lower triangular matrix results. The **algorithm of Sargent and Westerberg** [78, 169] determines this relabeling during a depth first search of the digraph. Vertices appearing in a cycle are collapsed into one composite vertex as soon as the cycle is detected. However, because repetitive collapsing of vertices can induce substantial overhead, it is more efficient to use **Tarjan's algorithm** [197], determining all strongly connected components during a depth first search of the graph in $O(|V| + |E|)$ time.

After application of both steps, we obtain a permuted matrix $P(\tilde{P}A)P^T$ which can be partitioned into a block lower triangular form:

$$P(\tilde{P}A)P^T = \begin{pmatrix} A_{11} & & \\ \vdots & \ddots & \\ A_{p1} & \dots & A_{pp} \end{pmatrix}$$

This partitioned matrix is fully reduced, since each diagonal block corresponds to a strongly connected component and as such is irreducible [108]. Edges that are incident to a vertex in a following strongly connected component give rise to nonzero off-diagonal blocks.

[1] This partition is necessarily a minimum partition into block triangular form (see section 4.1.2). In contrast, the diagonal blocks of a minimum partition into block triangular form can still be reducible, because this partition is defined by the nonzero structure only and does not account for possible permutations.

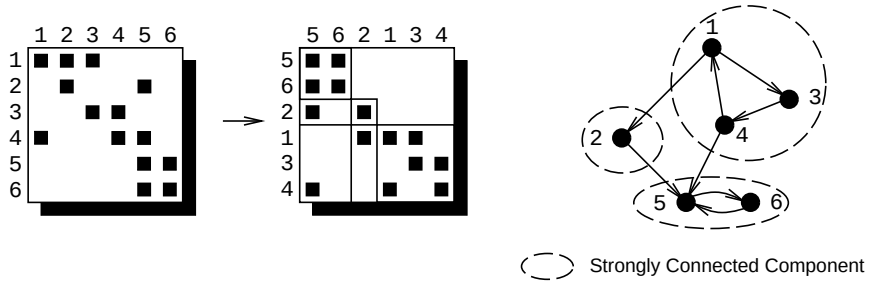


Figure A.8: Block Lower Triangular Form

An example of a symmetric permutation yielding a block lower triangular form is given in figure A.8, in which the original labeling is also used in the resulting matrix to illustrate the applied permutation.

A system $A\vec{x} = \vec{b}$ can be solved as the following sequence of smaller problems for $i = 1, \ldots, p$, where $\vec{b}' = P\tilde{P}\vec{b}$ and all vectors are partitioned according to the partition of $P(\tilde{P}A)P^T$:

$$A_{ii}\vec{y}_i = \vec{b}'_i - \sum_{j=1}^{i-1} A_{ij}\vec{y}_j$$

Once $\vec{y}$ has been computed, the solution is defined by $\vec{x} = P^T\vec{y}$. Conceptually, we apply a forward substitution at block level. Solving $A\vec{x} = \vec{b}$ as a sequence of smaller problems has as advantage that we only have to factorize the diagonal blocks A_{ii} to obtain the solution of each subproblem. Consequently, all fill-in is confined to these blocks. During the factorization of each diagonal block, pivoting for stability and preserving sparsity can be used. The off-diagonal blocks A_{ij} for $i \neq j$ only participate in multiplications. Hence, these blocks do no suffer from fill-in.

A.2.4 Symmetric A Priori Reordering Methods

Any partition of a matrix A into sub-matrices A_{ij}, where $1 \leq i \leq p$ and $1 \leq j \leq p$, gives rise to a partition of V in the graph $G = (V, E)$ associated with A. Obviously, the nonzero structure at block level of a partitioned matrix A is represented by the quotient graph defined by the corresponding partition of V. For example, the block lower triangular form considered in the previous section is represented by the acyclic condensation of the associated graph. Another example is shown in figure A.9. The four composite vertices in the quotient graph correspond to the four diagonal blocks in the partitioned matrix. Furthermore, there is an edge between two composite vertices in case the corresponding off-diagonal block is nonzero.

If we use the quotient graph associated with a block matrix to predict fill-in, a rather pessimistic approximation of the resulting nonzero structure at block level may result because we must assume that any product of two nonzero blocks yields a nonzero block.

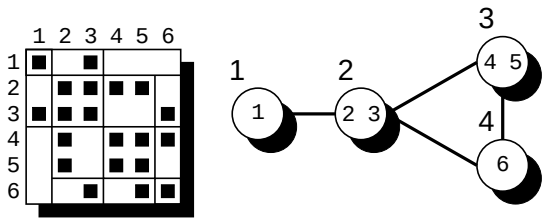


Figure A.9: Quotient Graph of a Partitioned Matrix

But even in this model, we know that for matrices having a symmetric block structure represented by a quotient graph that is a tree, performing the elimination according to a monotone labeling of the composite vertices proceeds without any fill-in at block level.

Because in such matrices at most one nonzero off-diagonal block appears below and to the right of each diagonal block, we can avoid fill-in in the off-diagonal blocks of A while solving a system $A\vec{x} = \vec{b}$ by performing the following **implicit block factorization** [78, 97]. Let A_{ij} for $1 \leq i \leq p$ and $1 \leq j \leq p$ denote the blocks in A of which the quotient graph is a tree with a monotone labeling. In that case, we first adapt the diagonal blocks as follows for $i = 1, \ldots, p$:

$$D_i = A_{ii} \; - \; \sum_{j=1}^{i-1} A_{ij} D_j^{-1} A_{ji}$$

Subsequently, the following computations are performed for $i = 1, \ldots, p$, in which all vectors are partitioned according to the partition of A:

$$\vec{c}_i = \vec{b}_i - \sum_{j=1}^{i-1} A_{ij} D_j^{-1} \vec{c}_j$$

Finally, the solution vector $\vec{x}$ is obtained as shown below for $i = p, \ldots, 1$:

$$\vec{x}_i = D_i^{-1} (\vec{c}_i - \sum_{j=i+1}^{n} A_{ij} \vec{c}_j)$$

We can avoid the explicit construction of each D_i^{-1} by performing all operations with the factorization of these diagonal blocks. Because the off-diagonal blocks remain unmodified, all fill-in is confined to the diagonal blocks. Therefore, some of the a priori ordering methods discussed in this section try to obtain a partitioned matrix for which the associated quotient graph is a tree with a monotone labeling. Because the methods are developed for (nearly) symmetric sparse matrices, all algorithms operate on undirected graphs, and symmetric permutations are used to preserve the symmetry of the original matrix.

Cuthill-McKee Method

The **Cuthill-McKee method** [52, 57, 58, 72, 78, 149, 215] method is used to reduce the bandwidth of a matrix by constructing a block tridiagonal form. Starting with a singleton $S_0 = \{v_k\}$, where v_k is a certain vertex in the associated undirected graph, each next level set S_i is constructed by taking neighbors of vertices in S_{i-1} that have not been used yet. In this manner, a **rooted level structure** [97, 169, 215] is obtained,[2] defined by the partition of the vertex set V into level sets $S_0, \ldots S_l$. The number l in this partition is called the **length**. The following formula defines the **width** of this level structure:

$$\max_{0 \leq i \leq l} |S_i|$$

Subsequently, the vertices in the level sets of this level structure are labeled consecutively. Within each level set, vertices are labeled in the order in which the neighbors of these vertices in the previous level set were inserted. Neighbors of the same vertex are labeled in the order of increasing degree. This kind of labeling can be easily obtained by a breadth-first search of the graph using a queue in which the neighbors of each next vertex are stored in order of increasing degree.

[2] Usually we assume that the associated undirected graph is connected. For disconnected graphs, the reordering methods can be applied to the different connected components

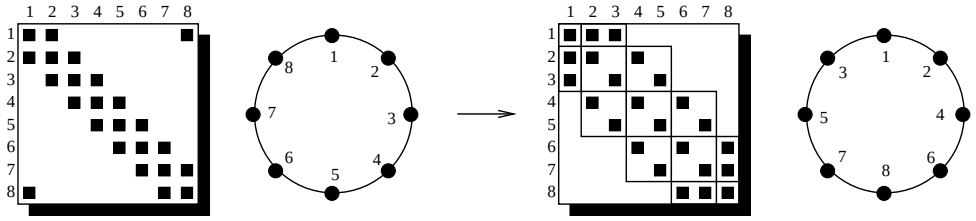


Figure A.10: Ring

The corresponding permutation gives rise to a matrix that can be partitioned into block tridiagonal form. The diagonal blocks are formed by the level sets, and off-diagonal blocks represent all interconnections:

$$PAP^T = \begin{pmatrix} A_{11} & A_{12} & & \\ A_{21} & A_{22} & \ddots & \\ & \ddots & \ddots & A_{p-1,p} \\ & & A_{p,p-1} & A_{pp} \end{pmatrix}$$

For example, we can use this method to reduce the bandwidth of the matrix shown in figure A.10, where the associated graph is a ring [149]. Starting with $S_0 = \{1\}$, we obtain the level sets $S_1 = \{2, 8\}$, $S_2 = \{3, 7\}$, $S_3 = \{4, 6\}$ and $S_4 = \{5\}$. Permuting the matrix according to a relabeling of the vertices in the order defined by the Cuthill-Mckee method yields the block tridiagonal form shown in this figure. This example also illustrates that labeling vertices in the same level set according to the order in which the neighbors of these vertices were inserted (e.g. in S_2 vertex 3 is labeled before vertex 7 if vertex 2 is inserted before vertex 8 in S_1) tends to cut off the corners of the off-diagonal blocks, which limits the resulting bandwidth.

The Cuthill-Mckee method is only effective if many small level sets result. Since the resulting level structure depends on the choice of the initial vertex, in [58] several initial vertices with low degree are considered. A good candidate for the initial vertex would be a **peripheral vertex**, which is a vertex $v \in V$ for which the largest distance to any other vertex in the graph, called the eccentricity $e(v) = \max\{d(v, w) | w \in V\}$, is equal to the largest eccentricity over all vertices, i.e. $\max\{e(v) | v \in V\}$, called the diameter of the graph. However, since obtaining such a vertex would be computationally infeasible, usually a **pseudo-peripheral vertex** is used, which is a vertex v for which $e(w) = e(v)$ holds for all w with $d(v, w) = e(v)$ [169]. Such vertices are probably just as good to start with and are easily obtained by an iterative construction of level structures [169, 215].

First, a level structure rooted at a vertex $r \in V$ of minimum degree is constructed. The length of this level structure is equal to the eccentricity $e(r)$. For each v in the last level set, $e(v) \geq e(r)$ holds. Therefore, the level structures rooted at each of these vertices are constructed in order of increasing degree. If a level structure with greater length is obtained, the process is repeated with that level structure. Otherwise, a pseudo-peripheral vertex has been obtained. Some adaptations have been proposed to improve the efficiency of this algorithm [95], although this may destroy the property that a pseudo-peripheral vertex will be found. For example, we can terminate the construction of a rooted level structures of which the width exceeds the width of the current level structure. Furthermore, we can limit the number of vertices in the last level set that are examined.

Consider, for instance, the undirected graph shown in figure A.11. Since all vertices have a degree of at least 3, we can arbitrarily start with any vertex of this degree. If vertex f is taken as root, a level structure of length 2 and width 4 results, where the last level set is $\{c, e, g, h\}$. To test whether f is a pseudo-peripheral vertex, level structures rooted at the vertices in this set are constructed.

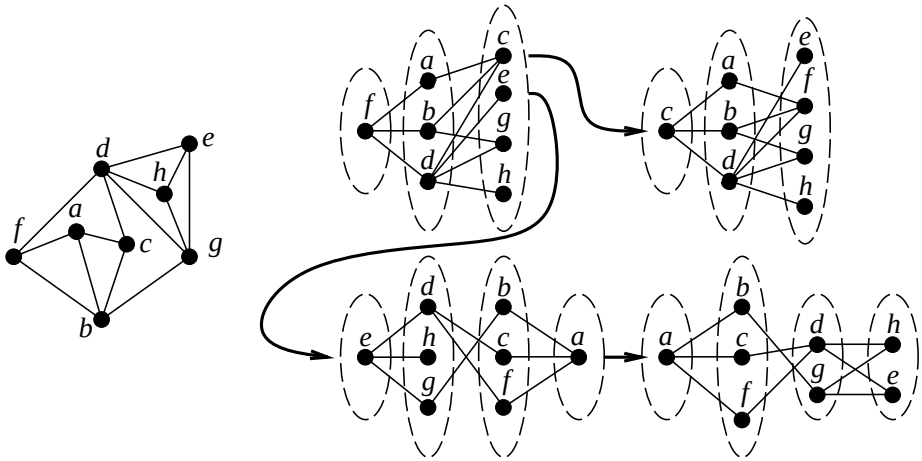


Figure A.11: Finding a Pseudo-Peripheral Vertex

Usually, these vertices are also considered in order of increasing degree, since this tends to reduce the number of level structures that must be constructed [78]. Since vertices c, e and h have minimum degree, we can arbitrarily construct the level structure rooted at c first, which is shown in figure A.11. Although no improvement in the length is obtained, we cannot conclude that f is pseudo-peripheral according to the previous given definition, because we must also consider all other vertices in the last level set of the level structure rooted at f. Construction of the level structure rooted at e, for instance, results in a level structure with length 3. Therefore, the process is restarted with the level structure rooted at e, as illustrated in figure A.11. Because the last level set consists of vertex a only, and the level structure rooted at a also has length 3, vertex e is a pseudo-peripheral vertex with eccentricity 3. In fact, because the eccentricity of this vertex is also equal to the diameter of the graph in this case, a true peripheral vertex has been found.

Profile Reduction Methods

Because a band form is preserved during Gaussian elimination without pivoting, the block tridiagonal form that is obtained by the Cuthill-Mckee method is very useful for methods that exploit all zero elements outside the band. Reversing the labeling of the graph, referred to as **reverse Cuthill-Mckee method** [57], frequently reduces the total number of elements in the profile. Therefore, this method is useful in combination with methods that exploit zero elements outside a variable band. Additionally, if the underlying graph is a tree, no fill-in is produced during LU-factorization in case this method is used. Another profile reduction method is formed by **King's algorithm** [57, 169]. Starting with a vertex of minimum degree, we select each next vertex from the vertices that are adjacent to already labeled vertices causing the least increase in the number of vertices in that latter group.

Refined Quotient Tree Algorithm

Another advantage of having a block tridiagonal matrix stems from the fact that the associated quotient graph is a simple chain. Consequently, the corresponding system can be solved with the implicit block factorization discussed at the beginning of this section. In an attempt to reduce the total amount of fill-in that occurs during this factorization, the **refined quotient tree algorithm** [97, 169] tries to convert such a quotient chain into a quotient tree by further partitioning the level sets. The method is based on the observation that each level set S_i can be partitioned according to the connected components of the level sets S_j for $i \leq j \leq l$.

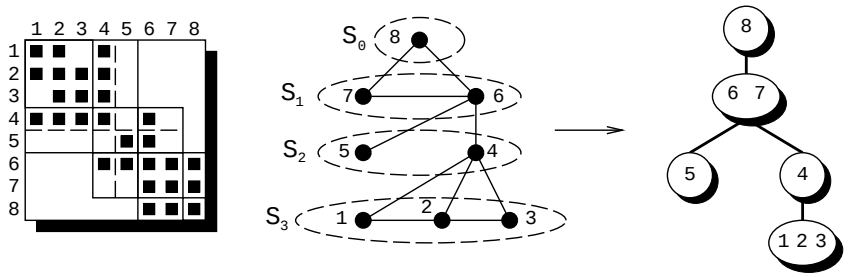


Figure A.12: Refined Quotient Tree

For example, in figure A.12, sets $\{5\}$ and $\{1, 2, 3, 4\}$ form the two connected components of the last two level sets. Hence, S_2 can be further partitioned into $\{5\}$ and $\{4\}$, resulting in the quotient tree that is also shown in this figure. Subsequently, we permute the original matrix according to a labeling of all vertices that corresponds to the monotone labeling of subsets in the refined quotient tree (i.e. all vertices in a subset are labeled consecutively while the vertices in one subset appear before all vertices in another subset if the former subset appears before the latter in the monotone labeling). In general, smaller diagonal blocks result, as illustrated with dashed lines in the figure.

Dissection Methods

Another method to obtain a partitioned matrix of which the associated quotient graph is a tree with a monotone labeling is **one-way dissection** [97]. First, we determine a number of separators of the associated graph. Obviously, if the vertices in these separators together with all incident edges are removed, the graph becomes disconnected. Hence, we label all vertices in the disconnected parts first, followed by all vertices in the separators. This relabeling induces a symmetric permutation converting the original matrix into a matrix that can be partitioned into doubly bordered block diagonal form. The diagonal blocks correspond to the disconnected parts, while the borders correspond to the vertices in the separators.

A simple example is given in figure A.13, where the separators $\{4, 5, 6\}$ and $\{10, 11, 12\}$ are used and the original labeling is shown to illustrate the resulting permutation. Because effectively a quotient tree results, again we can confine fill-in to the diagonal blocks by using the implicit block factorization presented at the beginning of this section. Moreover, because all diagonal blocks have a band structure, we can exploit zero elements outside this band during factorization of the diagonal blocks.

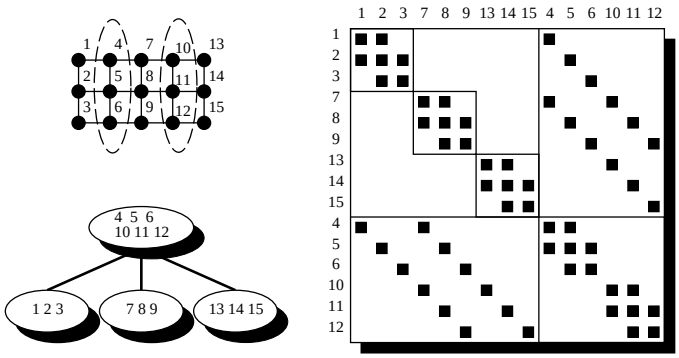


Figure A.13: One-way Dissection

Analysis of two-dimensional finite element problems (presented briefly at the end of this appendix) on regular $m \times l$ grids, where $m \leq l$, can be used to determine the number σ of vertical grid lines that must be used as separators that dissect the grid into independent blocks of comparable size for which either the storage requirements, the factorization time, or the solution time is minimized. The results of this analysis can also be used to obtain a one way dissection of irregular graphs automatically, if we use the length of the level structure as a measure for l, and the average number of elements in each level set as a measure for m. For instance, after a rooted level structure of reasonable size has been determined, using the following level sets for increasing value of i as separators, will tend to reduce the storage requirements:

$$S_{\lfloor i \cdot \delta + 0.5 \rfloor} \text{ where } \delta = \sqrt{\frac{3m+13}{2}}$$

Vertices in these sets that are not connected to vertices in the next level sets can be removed from these sets to obtain smaller separators. Vertices in each diagonal block can be relabeled according to reverse Cuthill-Mckee in order to reduce the bandwidth. In **nested dissection** [97], the level sets that is in the 'middle' of a rooted level set is used as separator and labeled last, so that the parts that become disconnected are of comparable size. Subsequently, the resulting diagonal blocks, corresponding to the separated connected components, are recursively ordered to doubly bordered block diagonal form. However, labeling *all* separators last causes all separators to appear in the border, which is more convenient with respect the required data structure. Again, the vertices in the chosen level sets that are not connected to vertices in the next set can be removed, in order to reduce the size of the separators.

Frontal Methods for Finite Element Problems

A method to find the solution of a partial differential equation on a particular region that allows for an irregular distribution of grid points is formed by the finite element method (see e.g. [7, 143, 201]). Rather than presenting the details of this method, we will focus on the algorithmic aspects of solving the corresponding systems of linear equations [169].

In the finite element method, the region of interest is discretized by subdividing this region into simple non-overlapping sub-regions, referred to as **finite elements**, where adjacent finite elements share boundaries. We define **nodes** on the boundaries, and possibly in the interior of the finite elements. In figure A.14, for example, we present a three- and six-node triangular element and a four- and eight-node quadrilateral elements which can be used in the two-dimensional case.

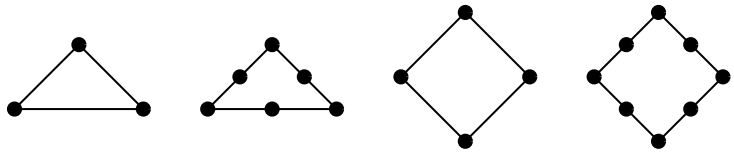


Figure A.14: Some Two-Dimensional Finite Elements

The finite elements in the domain are labeled consecutively from 1 to m. Nodes on the boundary of finite elements are shared by all finite elements to which this node belongs. Labeling all nodes from 1 to n, we can express this information in the **connectivity matrix** E [169]. This $m \times n$ boolean matrix has $e_{ij} = \mathbf{true}$ if a node with label j belongs to element with label i, and $e_{ij} = \mathbf{false}$ otherwise. In this sparse matrix, one row is associated with each element indicating the nodes that belong to this element. Likewise, one column is associated with each node indicating the elements to which this node belongs. Hence, sparse row-wise storage of E associates a list of node labels with each finite element, thereby implicitly storing the value true.

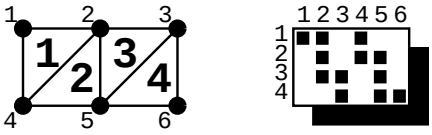


Figure A.15: Grid and Connectivity Matrix

In sparse row-wise storage of E^T (or sparse column-wise storage of E), we associate a list of labels with each node, indicating the labels of the finite elements to which this node belongs. In figure A.15, for instance, we present the connectivity matrix for a two-dimensional example is given, where a grid is formed by four finite elements and six nodes.

Several variables may be associated with each node. For instance, in the three-dimensional case, a displacement vector at a node can be decomposed into three components along the coordinate directions. For scalar problems, however, only one variable is associated with each node. In any case, the original problem can be formulated as a linear system of equations $A\vec{x} = \vec{b}$ for a **nodal assembly matrix** A, where the components of $\vec{x}$ correspond to these variables. The nodal assembly matrix A, also referred to as the stiffness matrix in structural analysis, consists of the sum of **element matrices** $A^{[k]}$ associated with each finite element. Computing this sum is referred to as **assembly**:

$$A = \sum_{k=1}^{m} A^{[k]} \tag{A.9}$$

Assuming that only one variable is associated with each node, each $n \times n$ element matrix $A^{[k]}$ has the property that $a_{ij}^{[e]} \neq 0$ can only hold if both the nodes with label i and j belong to the finite element with label k. Because only few nodes belong to one finite element, an element matrix is usually stored as a small dense matrix with indexing information. This indexing information can be thought of as a translation from local (internal) node labels to global (external) node labels. In figure A.16, the assembly of the nodal assembly matrix A belonging to the grid shown in figure A.15 is illustrated.

In this example, the graph representing the nonzero structure of the resulting matrix A is identical to the grid. However, in general, all vertices corresponding to nodes in the same finite element become connected, which may result in additional edges. For the grid shown in figure A.17, additional edges arise in the graph representing the nonzero structure of the resulting matrix [215].

Because the nodal assembly matrix is assembled according to (A.9), we know that $a_{ij} \neq 0$ can only hold if there is a finite element to which both the nodes with label i and j belong (ignoring exact cancellations that may occur during assembly). The nonzero zero structure of A can be obtained by computing the product $E^T E$, where the '**and**'-operator is used for a product and the '**or**'-operator for a sum.

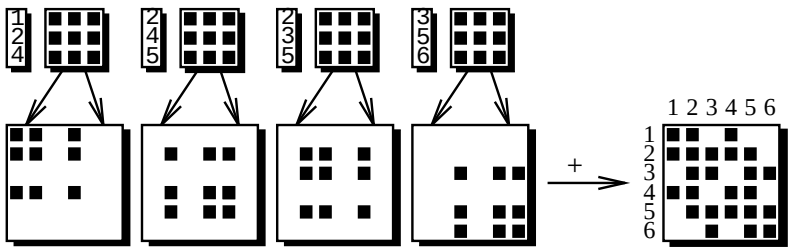


Figure A.16: Assembly

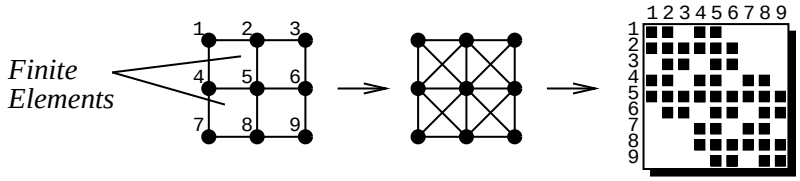


Figure A.17: Grid, Graph and Matrix

Usually, the number of nonzero elements in the resulting matrix is proportional to the number of nodes. Since this implies that only a few nonzero elements appear in each row, the resulting nodal assembly matrix can be very sparse. Therefore, a sparse method can be used to solve the system $A\vec{x} = \vec{b}$ after the assembly has been completed. Furthermore, the fact that the semi-bandwidth of A is defined as the maximum of $|i - j|$ over all nodes with labels i and j belonging to the same element justifies the use of a band method. A reordering method can be used to reduce this bandwidth even further.

An alternative technique, referred to as the **frontal method** [7, 76, 78, 81, 169], is based on the observation that eliminations can already be performed *during* assembly. A variable associated with a node becomes active as soon as the first finite element element to which this node belongs is assembled, i.e. when the corresponding element matrix $A^{[k]}$ is added to A. The variable can be eliminated after the last element in which this node occurs is assembled. Furthermore, the operations required for this elimination are confined to the sub-matrix formed by rows and columns corresponding to currently active variables. Consequently, all operations can be performed to this sub-matrix, called the **frontal matrix**, which is usually stored in a dense array that accounts for the largest possible size. Data associated with a variable is moved from secondary memory into this frontal matrix when this variable becomes active. After elimination of a variable, this data is moved back to secondary memory. Pivoting for stability can be incorporated in this technique by the additional use of some threshold criterion and unsymmetric permutations. If required, we delay some eliminations to obtain a suitable pivot, thereby only slightly increasing the size of the frontal matrix in practice [78].

The frontal method differs from other sparse techniques because, instead of reordering the matrix to reduce fill-in, we select an assembly ordering on finite elements that reduces the maximum size of the frontal matrix. The method allows efficient execution on vector processors because operating on the full frontal matrix avoids indirect addressing (see e.g. [76]), while very large problems can be solved because only the frontal method has to be kept in main memory. However, although the frontal method is an important technique, the sparse compiler presented in this dissertation provides no support for frontal methods, but simply assumes that all operations on sparse matrices are performed *after* assembly.

Bibliography

[1] S. Kamal Abdali and David S. Wise. Experiments with quadtree representation of matrices. In P. Gianni, editor, *Lecture Notes in Computer Science, No. 358*, pages 96–108. Springer-Verlag, 1988.

[2] R.C. Agarwal, F.G. Gustavson, and M. Zubair. A high performance algorithm using pre-processing for the sparse matrix-vector multiplication. In *Proceedings of the International Conference on Supercomputing*, pages 32–41, 1992.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques and Tools*. Addison-Wesley, 1986.

[4] Frances E. Allen and John Cocke. A catalogue of optimizing transformations. In Randall Rustin, editor, *Design and Optimization of Compilers*, pages 1–30. Prentice-Hall, 1971.

[5] J.R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9:491–542, 1987.

[6] J.R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Conf. Rec. ACM Sym. Principles of Programming Languages*, pages 177–189, 1983.

[7] R.J. Allwood. Matrix methods of structural analysis. In J.K. Reid, editor, *Large Sparse Sets of Linear Equations*, pages 17–24. Academic Press, 1971.

[8] F.L. Alvarado. A note on sorting sparse matrices. In *Proceedings of the IEEE*, volume 67, pages 1362–1363, 1979.

[9] F.L. Alvarado. *The Sparse Matrix Manipulation System: User and Reference Manual*. The University of Wisconsin, Madison, Wisconsin 53706, USA, 1993. SMMS93 software and documentation available from `ftp://eceserv0.ece.wisc.edu/pub/smms93`.

[10] Corinne Ancourt and Francois Irigoin. Scanning polyhedra with DO loops. In *Proceedings of Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 39–50, 1991.

[11] Edward Anderson and Youcef Saad. Solving sparse triangular linear systems on parallel computers. *International Journal of High Speed Computing*, 1(6):73–95, 1989.

[12] Laurence V. Atkinson. *Pascal Programming*. John Wiley and Sons, Chichester, 1980.

[13] E. Ayguadé, P.M.W. Knijnenburg, and J. Torres. Multi-transformations: Definition and usefulness. In *15th International Conference of the Chilean Computer Science Society*, Arica, Chile, 1995.

[14] Eduard Ayguadé and Jordi Torres. Partitioning the statement per iteration space using non-singular matrices. In *Proceedings of the International Conference on Supercomputing*, pages 407–415, 1993.

[15] Vasanth Balasundaram. *Interactive Parallelization of Numerical Scientific Programs*. PhD thesis, Department of Computer Science, Rice University, 1989.

[16] Vasanth Balasundaram. A mechanism for keeping useful internal information in parallel programming tools: The data access descriptor. *Journal of Parallel and Distributed Computing*, 9:154–170, 1990.

[17] Utpal Banerjee. *Dependence Analysis for Supercomputing*. Kluwer, Boston, 1988.

[18] Utpal Banerjee. Unimodular transformations of double loops. In *Proceedings of Third Workshop on Languages and Compilers for Parallel Computing*, 1990.

[19] Utpal Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations*. Kluwer, Boston, 1993.

[20] Utpal Banerjee. *Loop Parallelization*. Kluwer, Boston, 1994.

[21] J.L. Barlow. A note on monitoring the stability of triangular decomposition of sparse matrices. *SIAM J. Sci. Stat. Comput.*, 7:166–168, 1986.

[22] Michael Barnett and Christian Lengauer. Unimodularity considered non-essential. In *Proceedings of the Second Joint International Conference on Vector and Parallel Processing*, 1992.

[23] A.J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, 15(5):757–763, 1966.

[24] Aart J.C. Bik. A prototype restructuring compiler. Master's thesis, Utrecht University, 1992. INF/SCR-92-11.

[25] Aart J.C. Bik, Peter M.W. Knijnenburg, and Harry A.G. Wijshoff. Reshaping access patterns for generating sparse codes. In K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Lecture Notes in Computer Science, No. 892*, pages 406–422. Springer-Verlag, Berlin/New York, 1995.

[26] Aart J.C. Bik and Harry A.G. Wijshoff. Advanced compiler optimizations for sparse computations. In *Proceedings of Supercomputing*, pages 430–439, 1993.

[27] Aart J.C. Bik and Harry A.G. Wijshoff. Compilation techniques for sparse matrix computations. In *Proceedings of the International Conference on Supercomputing*, pages 416–424, 1993.

[28] Aart J.C. Bik and Harry A.G. Wijshoff. Nonzero structure analysis. In *Proceedings of the International Conference on Supercomputing*, pages 226–235, 1994.

[29] Aart J.C. Bik and Harry A.G. Wijshoff. On a completion method for unimodular matrices. Technical Report no. 94-14, Department of Computer Science, Leiden University, 1994.

[30] Aart J.C. Bik and Harry A.G. Wijshoff. On automatic data structure selection and code generation for sparse computations. In Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Lecture Notes in Computer Science, No. 768*, pages 57–75. Springer-Verlag, Berlin/New York, 1994.

[31] Aart J.C. Bik and Harry A.G. Wijshoff. Advanced compiler optimizations for sparse computations. *Journal of Parallel and Distributed Computing*, 31:14–24, 1995.

[32] Aart J.C. Bik and Harry A.G. Wijshoff. Construction of representative simple sections. In *Proceedings of the International Conference on Parallel Processing*, pages 9–18, 1995. Volume 2: Software.

[33] Aart J.C. Bik and Harry A.G. Wijshoff. Implementation of Fourier-Motzkin elimination. In *Proceedings of the first annual conference of the Advanced School for Computing and Imaging (ASCI)*, pages 377–386, 1995. Heijen, The Netherlands.

[34] Aart J.C. Bik and Harry A.G. Wijshoff. Annotations for a sparse compiler. In C.-H. Huang, P. Sadayappan, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Lecture Notes in Computer Science, No. 1033*, pages 500–514. Springer-Verlag, 1996.

[35] Aart J.C. Bik and Harry A.G. Wijshoff. Automatic data structure selection and transformation for sparse matrix computations. *IEEE Transactions on Parallel and Distributed Systems*, 7(2):109–126, 1996.

[36] Aart J.C. Bik and Harry A.G. Wijshoff. Iteration space partitioning. In H. Liddell, A. Colbrook, B. Hertzberger, and P. Sloot, editors, *Lecture Notes in Computer Science, No. 1067*, pages 475–484. Springer-Verlag, 1996.

[37] Aart J.C. Bik and Harry A.G. Wijshoff. MT1: A prototype restructuring compiler. In *Proceedings of the second annual conference of the Advanced School for Computing and Imaging (ASCI)*, pages 78–83, 1996. Lommel, Belgium.

[38] Aart J.C. Bik and Harry A.G. Wijshoff. A note on dealing with subroutines and functions in the automatic generation of sparse codes. In *Proceedings of the second annual conference of the Advanced School for Computing and Imaging (ASCI)*, pages 96–101, 1996. Lommel, Belgium.

[39] Aart J.C. Bik and Harry A.G. Wijshoff. The use of iteration space partitioning to construct representative simple sections. *Journal of Parallel and Distributed Computing*, 34:95–110, 1996.

[40] David M. Bloom. *Linear Algebra and Geometry*. Cambridge University Press, Cambridge, 1979.

[41] William Blume, Rudolf Eigenmann, Jay Hoeflinger, David Padua, Paul Petersen, Lawrence Rauchwerger, and Peng Tu. Automatic detection of parallelism: A grand challenge for high-performance computing. Technical Report no. 1348, Center for Supercomputing Research and Development, University of Illinios, 1994.

[42] K. Borsuk. *Multidimensional Analytic Geometry*. Polish Scientific Publishers, Warszawa, 1969.

[43] James M. Boyle, Maurice Clint, Stephen Fitzpatrick, and Terence J. Harmer. The construction of numerical mathematical software for the AMT DAP by program transformation. In L. Bouge, M. Cosnard, Y. Robert, and D.Trystram, editors, *Lecture Notes in Computer Science, No. 634*, pages 761–767. Springer-Verlag, 1992.

[44] W.S. Brainerd, Ch.H Goldberg, and J.C. Adams. *Fortran 90*. Academic Service, 1990.

[45] Peter J.H. Brinkhaus. Compiler analysis of procedure calls. Master's thesis, Utrecht University, 1993. INF/SCR-93-13.

[46] James R. Bunch. Partial pivoting strategies for symmetric matrices. *SIAM J. Numer. Anal.*, 11:521–528, 1974.

[47] Michael Burke and Ron Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the Symposium on Compiler Construction*, pages 162–175, 1986.

[48] David Callahan. *A Global Approach to Detection of Parallelism*. PhD thesis, Department of Computer Science, Rice University, 1987.

[49] H.H. ten Cate. Applying abstraction and formal specification in numerical software design. *Computers Math. Applic.*, 29(12):81–102, 1995.

[50] Z. Chamski. Nested loop sequences: Towards efficient loop structures in automatic parallelization. In *Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences*, pages 14–22, 1994.

[51] Alex L. Cheung and Anthony P. Reeves. Sparse data representation for dense data-parallel computation. In *Proceedings of the International Conference on Parallel Processing*, pages 106–113, 1992.

[52] Thomas F. Coleman. Large sparse numerical optimization. In G. Goos and J. Hartmanis, editors, *Lecture Notes in Computer Science, No. 165*. Springer-Verlag, 1984.

[53] L.H. Colgan. Iterative methods for solving large sparse linear systems. In J. Noye, editor, *Numerical Solutions of Partial Differential Equations*, pages 367–396. North-Holland Publishing Company, Amsterdam, 1982.

[54] Keith D. Cooper, Mary W. Hall, and Ken Kennedy. Procedure cloning. In *Proceedings of the IEEE International Conference on Computer Languages*, pages 96–105, 1992.

[55] Keith D. Cooper, Ken Kennedy, and Linda Torczon. The impact of interprocedural analysis and optimization in the R^n programming environment. *ACM Transactions on Programming Languages and Systems*, 8:491–523, 1986.

[56] A.R. Curtis and J.K. Reid. The solution of large sparse unsymmetric systems of linear equations. *Journal Inst. Maths. Applics.*, 8:344–353, 1971.

[57] Elizabeth Cuthill. Several strategies for reducing the bandwidth of matrices. In Donald J. Rose and Ralph A. Willoughby, editors, *Sparse Matrices and Their Applications*, pages 157–166. Plenum Press, New York, 1972.

[58] Elizabeth Cuthill and J. Mckee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of 24th National Conference of the ACM*, pages 157–172, 1969.

[59] Ron G. Cytron. Doacross, beyond vectorization for multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, pages 836–844, 1986.

[60] Ron G. Cytron. Limited processor scheduling of doacross loops. In *Proceedings of the International Conference on Parallel Processing*, pages 226–234, 1987.

[61] George B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, Princeton, New Jersey, 1963.

[62] George B. Dantzig and B. Curtis Eaves. Fourier-Motzkin elimination and its dual. *Journal of Combinatorial Theory*, 14:288–297, 1973.

[63] Ervan Darnell, John M. Mellor-Crummey, and Ken Kennedy. Automatic software cache coherence through vectorization. In *Proceedings of the International Conference on Supercomputing*, pages 129–138, 1992.

[64] P.F.G. Dechering, J.A. Trescher, J.P.M. de Vreught, and H.J. Sips. V-cal: a Calculus for the Compilation of Data Parallel Languages. In *Languages and Compilers for Parallel Computing*, Columbus, Ohio, August 1995.

[65] B. Dembart and K.W. Neves. Sparse triangular factorization on vector computers. In *Exploring Applications of Parallel Processing*, pages 22–25, 1977.

[66] Erik H. D'Hollander. Partitioning and labeling of index sets in DO loops with constant dependence vectors. In *Proceedings of the International Conference on Parallel Processing*, pages 139–144, 1989. Volume 2: Software.

[67] E.W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*. Academic Press, New York, 1968.

[68] David S. Dodson, Roger G. Grimes, and John G. Lewis. Algorithm 692: Model implementation and test package for the sparse linear algebra subprograms. *ACM Transactions on Mathematical Software*, 17:264–272, 1991.

[69] David S. Dodson, Roger G. Grimes, and John G. Lewis. Sparse extensions to the Fortran basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 17:253–263, 1991.

[70] Jack J. Dongarra, Iain S. Duff, Danny C. Sorensen, and Henk A. van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers*. Society for Industrial and Applied Mathematics, 1991.

[71] Michael L. Dowling. Optimal code parallelization using unimodular transformations. *Parallel Computing*, 16:157–171, 1990.

[72] Iain S. Duff. A survey of sparse matrix research. In *Proceedings of the IEEE*, pages 500–535, 1977.

[73] Iain S. Duff. Practical comparisons of codes for the solution of sparse linear systems. In Iain S. Duff and G.W. Stewart, editors, *Sparse Matrix Proceedings 1978*, pages 107–134. SIAM, Philadelphia, 1979.

[74] Iain S. Duff. MA28 – a set of Fortran subroutines for sparse unsymmetric linear equations. Technical Report AERE R-8730 (1980 revision), Computer Science and Systems Division, AERE Harwell, 1980.

[75] Iain S. Duff. A sparse future. In Iain S. Duff, editor, *Sparse Matrices and their Uses*, pages 1–29. Academic Press, London, 1981.

[76] Iain S. Duff. The solution of sparse linear equations on the CRAY-1. In J.S. Kowalik, editor, *High-Speed Computing*, pages 293–309. Springer-Verlag, Berlin, Heidelberg, New York, Tokyo, 1984. NATO ASI Series, Volume F7.

[77] Iain S. Duff. Data structures, algorithms and software for sparse matrices. In David J. Evans, editor, *Sparsity and Its Applications*, pages 1–29. Cambridge University Press, 1985.

[78] Iain S. Duff, A.M. Erisman, and J.K. Reid. *Direct Methods for Sparse Matrices*. Oxford Science Publications, Oxford, 1990.

[79] Iain S. Duff, Roger G. Grimes, and John G. Lewis. Sparse matrix test problems. *ACM Transactions on Mathematical Software*, 15:1–14, 1989.

[80] Iain S. Duff and J.K. Reid. Some design features of a sparse matrix code. *ACM Transactions on Mathematical Software*, pages 18–35, 1979.

[81] Iain S. Duff and J.K. Reid. MA27 – a set of Fortran subroutines for solving sparse symmetric sets of linear equations. Technical Report AERE R-10533, Computer Science and Systems Division, AERE Harwell, 1982.

[82] C. Eisenbeis, O. Temam, and H. Wijshoff. On efficiently characterizing solutions of linear diophantine equations and its application to data dependence analysis. In *Proceedings of the Seventh International Symposium on Computer and Information Sciences*, 1992.

[83] J. Engelfriet. Attribute grammars: Attribute evaluation methods. In B. Lorho, editor, *Methods and Tools for Compiler Construction*, pages 103–138. Cambridge University Press, 1984.

[84] Jocelyne Erhel and Bernard Philippe. Multiplication of a vector by a sparse matrix on supercomputers. In M. Cosnard, M.H. Barton, and M. Vanneschi, editors, *Parallel Processing*, pages 181–187. Elsevier Science Publishers B.V., North-Holland, 1988.

[85] D.J. Evans. Iterative sparse matrix algorithms. In D.J. Evans, editor, *Software for Numerical Mathematics*, pages 49–83. Academic Press, London and New York, 1974.

[86] Thomas Fahringer, Roman Blasko, and Hans P. Zima. Automatic performance prediction to support parallelization of Fortran programs for massively parallel systems. In *Proceedings of the International Conference on Supercomputing*, pages 347–356, 1992.

[87] Stephen Fitzpatrick, M. Clint, and P. Kilpatrick. The automatated derivation of sparse implementations of numerical algorithms through program transformation. Technical Report Apr-SF.MC.PLK, The Queen's University of Belfast, Department of Computer Science, 1995.

[88] Stephen Fitzpatrick, T.J. Harmer, and J.M. Boyle. Deriving efficient parallel implementations of algorithms operating on general sparse matrices using automatic program transformation. In Bruno Buchberger and Jens Volkert, editors, *Lecture Notes in Computer Science, No. 854*, pages 148–159. Springer-Verlag, 1994.

[89] M.J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966.

[90] George E. Forsythe and Cleve B. Moler. *Computer Solution of Linear Algebraic Equations*. Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1967.

[91] Kyle Gallivan, William Jalby, and Dennis Gannon. On the problem of optimizing data transfers for complex memory systems. In *Proceedings of the International Conference on Supercomputing*, pages 238–253, 1988.

[92] Dennis Gannon et al. SIGMA II: A tool kit for building parallelizing compilers and performance analysis systems. Department of Computer Science, Indiana University, 1992.

[93] Alan George and Joseph W.H. Liu. A note on fill for sparse matrices. *SIAM J. Numer. Anal.*, 12:452–455, 1975.

[94] Alan George and Joseph W.H. Liu. The design of a user interface for a sparse matrix package. *ACM Transactions on Mathematical Software*, 5:139–162, 1979.

[95] Alan George and Joseph W.H. Liu. An implementation of a pseudoperipheral node finder. *ACM Transactions on Mathematical Software*, 5:284–295, 1979.

[96] Alan George and Joseph W.H. Liu. A fast implementation of the minimum degree algorithm using quotient graphs. *ACM Transactions on Mathematical Software*, 6:337–358, 1980.

[97] Alan George and Joseph W.H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, Englewood Cliffs, New York, 1981.

[98] Alan George and Joseph W.H. Liu. The evolution of the minimum degree ordering algorithm. *SIAM Review*, 31:1–19, 1989.

[99] Alan George and Esmond Ng. Symbolic factorization for sparse gaussian elimination with partial pivoting. *SIAM J. Sci. Stat. Comput.*, 8:877–898, 1987.

[100] Gilbert. *Modern Algebra with Applications*. Kluwer, Boston, 1994.

[101] Patricia C. Goldberg. A comparison of certain optimization techniques. In Randall Rustin, editor, *Design and Optimization of Compilers*, pages 31–50. Prentice-Hall, 1971.

[102] Gene H. Golub and Charles F. van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, Maryland, 1983.

[103] A.J. van de Goor. *Computer Architecture*. Delftse Uitgevers Maatschappij, Delft, 1989.

[104] Branko Grünbaum. *Convex Polytopes*. Interscience Publishers, London, 1967.

[105] Fred G. Gustavson. Some basic techniques for solving sparse systems of linear equations. In Donald J. Rose and Ralph A. Willoughby, editors, *Sparse Matrices and Their Applications*, pages 41–52. Plenum Press, New York, 1972.

[106] Fred G. Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Transactions on Mathematical Software*, 4:250–269, 1978.

[107] G. Hadley. *Linear Programming*. Addison-Wesley, Reading, Massachusetts, U.S.A., 1962. University of Chicago.

[108] Frank Harary. Sparse matrices and graph theory. In J.K. Reid, editor, *Large Sparse Sets of Linear Equations*, pages 139–150. Academic Press, 1971.

[109] Paul Havlak and Ken Kennedy. Experience with interprocedural analysis of array side effects. *Supercomputing*, pages 952–961, 1990.

[110] Donald Hearn and M. Pauline Baker. *Computer Graphics*. Prentice-Hall International, 1986.

[111] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990.

[112] A. Jennings. A compact storage scheme for the solution of symmetric linear simultaneous equations. *The Computer Journal*, 9:281–285, 1966.

[113] A. Jennings and A.D. Tuff. A direct method for the solution of large sparse symmetric simultaneous equations. In J.K. Reid, editor, *Large Sparse Sets of Linear Equations*, pages 97–104. Academic Press, 1971.

[114] Stephen C. Johnson. Yacc: Yet another compiler-compiler. Technical Report 32, Bell Laboratories, Murray Hill, New Jersey 07974, 1975.

[115] Geraint Jones and Michael Goldsmith. *Programming in OCCAM 2*. Prentice Hall, New York, 1988. C.A.R. Hoare, series editor.

[116] Uwe Kastens. Lifetime analysis for attributes. *Acta Informatica*, pages 633–652, 1987.

[117] Brian W. Kernighan. *The UNIX Programming Environment*. Prentice Hall, Englewood Cliffs, New Jersey, 1984.

[118] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[119] Peter M.W. Knijnenburg. Towards unimodular transformations for non-perfectly nested loops. Technical Report no. 94-41, Department of Computer Science, Leiden University, 1994.

[120] Peter M.W. Knijnenburg and Aart J.C. Bik. On reducing overhead in loops. In *Proceedings of the Fifth International Workshop on Compilers for Parallel Computers*, pages 200–211, 1995.

[121] P.M.W. Knijnenburg, E. Ayguadé, and J. Torres. Multi-transformations: Code generation and validity. Technical Report 95-12, Department of Computer Science, Leiden University, 1995.

[122] Donald E. Knuth. *The Art of Computer Programming*. Addison Wesley, Reading, Massachusetts, 1968. Volume 1: Fundamental Algorithms.

[123] Donald E. Knuth. *An Empirical Study of Fortran Programs*. U.S. Department of Commerce, Stanford University, 1971.

[124] Donald E. Knuth. *The Art of Computer Programming*. Addison Wesley, Reading, Massachusetts, 1973. Volume 3: Sorting and Searching.

[125] Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. Automatic parallelization of the conjugate gradient algorithm. In C.-H. Huang, P. Sadayappan, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Lecture Notes in Computer Science, No. 1033*, pages 480–499. Springer-Verlag, 1996.

[126] V. Prasad Krothapalli, Thulasiraman Jeyaraman, and Mark Giesbrecht. Run-time parallelization of irregular doacross-loops. In Afonso Ferreira and José Rolim, editors, *Lecture Notes in Computer Science, No. 980*, pages 75–80. Springer-Verlag, 1995.

[127] David J. Kuck. *The Structure of Computers and Computations*. John Wiley and Sons, New York, 1978. Volume 1.

[128] David J. Kuck et al. The effects of program restructuring, algorithm change, and architecture choice on program performance. In *Proceedings of the International Conference on Parallel Processing*, pages 129–138, 1984.

[129] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Programming*. The Benjamin/Cummings Publishing Company, Redwood City, CA, 1994.

[130] Leslie Lamport. The parallel execution of DO loops. *Communications of the ACM*, pages 83–93, 1974.

[131] Kincho H. Law and Steven J. Fenves. A node-addition model for symbolic factorization. *ACM Transactions on Mathematical Software*, 12:37–50, 1986.

[132] Gyungho Lee, Clyde P. Kruskal, and David J. Kuck. An empirical study of automatic restructuring of nonnumerical programs for parallel processors. *IEEE Transactions on Computers*, pages 927–933, 1985.

[133] M.E. Lesk and E. Schmidt. Lex: A lexical analyzer generator. Technical Report 39, Bell Laboratories, Murray Hill, New Jersey 07974, 1975.

[134] Shun-Tak Leung and John Zahorjan. Improving the performance of run-time parallelization. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 83–91, 1993.

[135] John M. Levesque and Joel W. Williamson. *A Guidebook to Fortran on Supercomputers*. Academic Press, Inc., San Diego, 1991.

[136] John R. Levine, Tony Mason, and Doug Brown. *Lex and Yacc*. O'Reilly and Associates, Sebastopol, CA, 1992.

[137] John G. Lewis and Horst D. Simon. The impact of hardware gather/scatter on sparse Gaussian elimination. *SIAM J. Sci. Stat. Comput.*, Volume 9:304–311, 1988.

[138] Wei Li. Compiler cache optimizations for banded matrix problems. In *Proceedings of the International Conference on Supercomputing*, pages 21–30, 1995.

[139] Wei Li and Keshav Pingali. Access normalization: Loop restructuring for numa compilers. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 285–295, 1992.

[140] Wei Li and Keshav Pingali. A singular loop transformation framework based on non-singular matrices. In *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, 1992.

[141] Zhiyuan Li and Walid Abu-Sufah. On reducing data synchronization in multiprocessed loops. *IEEE Transactions on Computers*, C-36:105–109, 1987.

[142] Zhiyuan Li and Pen-Chung Yew. Interprocedural analysis for parallel computing. In *Proceedings of the International Conference on Parallel Processing*, pages 221–228, 1988. Volume 2: Software.

[143] H.X. Lin. *A Methodology for the Parallel Direct Solution of Finite Element Systems*. PhD thesis, Delft University of Technology, 1993.

[144] Joseph W.H. Liu. Modification of the minimum degree algorithm by multiple elimination. *ACM Transactions on Mathematical Software*, 11:141–153, 1985.

[145] Joseph W.H. Liu. A compact row storage scheme for cholesky factors using elimination trees. *ACM Transactions on Mathematical Software*, pages 127–148, 1986.

[146] Joseph W.H. Liu. A generalized envelope method for sparse factorization by rows. *ACM Transactions on Mathematical Software*, 17:112–129, 1991.

[147] David B. Loveman. Program improvement by source-to-source transformations. *Journal of the ACM*, 24:121–145, 1977.

[148] Mary E. Mace. *Memory Storage Patterns in Parallel Processing*. Kluwer, Boston, 1987.

[149] Ken J. Mann. Inversion of large sparse matrices: Direct methods. In J. Noye, editor, *Numerical Solutions of Partial Differential Equations*, pages 313–366. North-Holland Publishing Company, Amsterdam, 1982.

[150] H.M. Markowitz. The elimination form of the inverse and its applications. *Management Science*, 3:255–269, 1957.

[151] Dror E. Maydan, John L. Hennessy, and Monica S. Lam. Efficient and exact data dependence analysis. In *Proceedings ACM SIGPLAN Conference on Programming Languages Design and Implementation*, pages 1–14, 1991.

[152] Kathryn S. McKinley. *Automatic and Interactive Parallelization*. PhD thesis, Rice University, 1992.

[153] P. McMullen and G.C. Shephard. *Convex Polytopes and the Upper Bound Conjecture*. Cambridge University Press, 1971.

[154] John Michael McNamee. Algorithm 408: A sparse matrix package. *Communications of the ACM*, pages 265–273, 1971.

[155] Samuel P. Midkiff. *The Dependence Analysis and Synchronization of Parallel Programs*. PhD thesis, C.S.R.D., 1993.

[156] Samuel P. Midkiff and David A. Padua. Compiler generated synchronization for DO loops. In *Proceedings of the International Conference on Parallel Processing*, pages 544–551, 1986.

[157] Samuel P. Midkiff and David A. Padua. Compiler algorithms for synchronization. *IEEE Transactions on Computers*, C-36:1485–1495, 1987.

[158] Kenneth W. Neves. Vectorization of scientific software. In J.S. Kowalik, editor, *High-Speed Computing*, pages 277–291. Springer-Verlag, Berlin, Heidelberg, New York, Tokyo, 1984. NATO ASI Series, Volume F7.

[159] Morris Newman. *Integral Matrices*. Academic Press, New York, 1972. Pure and Applied Mathematics, Volume 45.

[160] Alexandru Nicolau. Run-time disambiguation: Coping with statically unpredictable dependences. *IEEE Transactions on Computers*, 38(5):663–678, 1989.

[161] U.S. Department of Commerce/National Bureau of Standards. *Using ANS Fortran*. U.S. Government Printing Office, Washington, 1980. Edited by Gordon Lyon.

[162] Thomas C. Oppe and David R. Kincaid. The performance of ITPACK on vector computers for solving large sparse linear systems arising in sample oil reservoir simulation problems. *Communications in Applied Numerical Methods*, 3(1):23–29, 1987.

[163] James M. Ortega and Jr. William G. Poole. *Numerical Methods for Differential Equations*. Pitman Publishing, Marshfield, Massachusetts, 1981.

[164] Ole Østerby and Zahari Zlatev. Direct methods for sparse matrices. In G. Goos and J. Hartmanis, editors, *Lecture Notes in Computer Science, No. 157*. Springer-Verlag, Berlin, 1983.

[165] David A. Padua, David J. Kuck, and Duncan H. Lawrie. High speed multiprocessors and compilation techniques. *IEEE Transactions on Computers*, C-29:763–776, 1980.

[166] David A. Padua and Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29:1184–1201, 1986.

[167] S. Parter. The use of linear graphs in Gauss elimination. *SIAM Review*, 3:119–130, 1961.

[168] Frans J. Peters. Parallelism and sparse linear equations. In David J. Evans, editor, *Sparsity and Its Applications*, pages 285–301. Cambridge University Press, 1985.

[169] Sergio Pissanetsky. *Sparse Matrix Technology*. Academic Press, London, 1984.

[170] Constantine D. Polychronopoulos. *Parallel Programming and Compilers*. Kluwer, Boston, 1988.

[171] Constantine D. Polychronopoulos, David J. Kuck, and David A. Padua. Execution of parallel loops on parallel processor systems. In *Proceedings of the International Conference on Parallel Processing*, pages 519–527, 1986.

[172] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry*. Springer-Verlag, New York, 1985.

[173] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes*. Cambridge University Press, Cambridge, 1986.

[174] William Pugh and David Wonnacott. An evaluation of exact methods for analysis of value-based array data dependences. In *Proceedings of the Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, 1993.

[175] Michael J. Quinn. *Designing Efficient Algorithms for Parallel Computers*. McGraw-Hill, New York, 1987.

[176] Michael J. Quinn. *Parallel Computing: Theory and Practice*. McGraw-Hill, New York, 1994.

[177] J.K. Reid. A note on the stability of Gaussian elimination. *Journal Inst. Maths. Applics*, 8:374–375, 1971.

[178] J.K. Reid. Direct methods for sparse matrices. In D.J. Evans, editor, *Software for Numerical Mathematics*, pages 29–47. Academic Press, London and New York, 1974.

[179] Thomas W. Reps and Tim Teitelbaum. *The Synthesizer Generator*. Springer-Verlag, New York, 1988.

[180] L.F. Romero and E.L. Zapata. Data distributions for sparse matrix vector multiplication. In *Proceedings of the Fourth International Workshop on Compilers for Parallel Computers*, pages 154–167, 1993.

[181] L.F. Romero and E.L. Zapata. Data distributions for sparse matrix vector multiplication. *Journal of Parallel Computing*, 21(4):583–605, 1995.

[182] Donald J. Rose and Robert Endre Tarjan. Algorithmic aspects of vertex elimination on directed graphs. *SIAM J. Appl. Math.*, 34:176–197, 1978.

[183] Barbara G. Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, SE-5:216–226, 1979.

[184] Youcef Saad. Krylov subspace methods on supercomputers. *SIAM J. Sci. Stat. Comput.*, 10:1200–1232, 1989.

[185] Youcef Saad. SPARSKIT: a basic tool kit for sparse matrix computations. CSRD/RIACS, 1990.

[186] Youcef Saad and Harry A.G. Wijshoff. Spark: A benchmark package for sparse computations. In *Proceedings of the International Conference on Supercomputing*, pages 239–253, 1990.

[187] Joel H. Saltz. Aggregation methods for solving sparse triangular systems on multiprocessors. *SIAM J. Sci. Stat. Comput.*, 11:123–144, 1990.

[188] Joel H. Saltz, Kathleen Crowley, Ravi Mirchandaney, and Harry Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8:303–312, 1990.

[189] Joel H. Saltz, Ravi Mirchandaney, and Kathleen Crowley. The DOConsider loop. In *Proceedings of the International Conference on Supercomputing*, pages 29–40, 1989.

[190] Joel H. Saltz, Ravi Mirchandaney, and Kathleen Crowley. Run-time parallelization and scheduling of loops. *IEEE Transactions on Computers*, 40:603–612, 1991.

[191] Hanan Samet. Connected component labeling using quadtrees. *Journal of the ACM*, 28:487–501, 1981.

[192] Zhiyu Shen, Zhiyuan Li, and Pen-Chung Yew. An empirical study on array subscripts and data dependencies. In *Proceedings of the International Conference on Parallel Processing*, pages 145–152, 1989. Volume 2: Software.

[193] Andrew H. Sherman. ALGORITHM 533 NSPIV, a Fortran subroutine for sparse gaussian elimination with partial pivoting. *ACM Transactions on Mathematical Software*, 4:391–398, 1978.

[194] Abraham Siberschatz, James L. Peterson, and Peter B. Galvin. *Operating System Concepts*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1991.

[195] Per Stenström and Lund University. A survey of cache coherence schemes for multiprocessor. *Computer*, pages 12–24, 1990.

[196] E. Su et al. Advanced compilation techniques in the PARADIGM compiler for distributed-memory multicomputers. In *Proceedings of the International Conference on Supercomputing*, pages 424–433, 1995.

[197] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Computing*, pages 146–160, 1972.

[198] Reginal P. Tewarson. Sorting and ordering sparse linear systems. In J.K. Reid, editor, *Large Sparse Sets of Linear Equations*, pages 151–167. Academic Press, 1971.

[199] Reginal P. Tewarson. *Sparse Matrices*. Academic Press, New York, 1973.

[200] William F. Tinney and John W. Walker. Direct solutions of sparse network equations by optimally ordered triangular factorization. In *Proceedings of the IEEE*, pages 1801–1809, 1967.

[201] Josef A. Tomas. The finite element method in engineering practice and education. In J. Noye, editor, *Numerical Solutions of Partial Differential Equations*, pages 227–288. North-Holland Publishing Company, Amsterdam, 1982.

[202] Ljubomir B. Tosovic. Some experiments on sparse sets of linear equations. *SIAM J. Appl. Math.*, 25:142–148, 1973.

[203] David A. Towers. *Guide To Linear Algebra*. Macmillan, 1988.

[204] M. Ujaldon, S. Sharma, J. Saltz, and E.L. Zapata. Run-time techniques for parallelizing sparse matrix problems. In Afonso Ferreira and José Rolim, editors, *Lecture Notes in Computer Science, No. 980*, pages 43–57. Springer-Verlag, Berlin, 1996.

[205] M. Ujaldon and E.L. Zapata. Development and implementation of data-parallel compilation techniques for sparse codes. In *Proceedings of the Fifth International Workshop on Compilers for Parallel Computers*, pages 78–97, 1995.

[206] M. Ujaldon and E.L. Zapata. Efficient resolution of sparse indirections in data-parallel compilers. In *Proceedings of the International Conference on Supercomputing*, pages 117–126, 1995.

[207] M. Ujaldon, E.L. Zapata, B.M. Chapman, and H.P. Zima. Data-parallel computations for sparse codes: A survey and contributions. In B.K. Szymanski and B. Sinharoy, editors, *Languages, Compilers and Run-Time Systems for Scalable Computers*, pages 253–264. Kluwer, 1995.

[208] M. Ujaldon, E.L. Zapata, B.M. Chapman, and H.P. Zima. New data-parallel language features for sparse matrix computations. In *9th IEEE International Parallel Processing Symposium*, pages 742–749, 1995.

[209] M. Ujaldon, E.L. Zapata, B.M. Chapman, and H.P. Zima. Vienna–Fortran/HPF extensions for sparse and irregular problems and their compilation. Technical Report TR 95-5, Institute for Software Technology and Parallel Systems, University of Vienna, 1995.

[210] H.A. van der Vorst. Iterative solution methods for certain sparse linear systems with a non-symmetric matrix arising from pde-problems. *J. Comp. Phys.*, 44:1–19, 1981.

[211] H.A. van der Vorst. High performance preconditioning. *SIAM J. Sci. Stat. Comput.*, 10(6):1174–1185, November 1989.

[212] J.C. van Vliet. *Software Engineering*. H.E. Stenfert Kroese B.V., Leiden, 1988.

[213] Richard S. Varga. *Matrix Iterative Analysis*. Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1962.

[214] M. Veldhorst. *An Analysis of Sparse Matrix Storage Schemes*. PhD thesis, Mathematisch Centrum, Amsterdam, 1982.

[215] R. Wait. *The Numerical Solution of Algebraic Equations*. John Wiley and Sons, Chichester, 1979.

[216] Joan Walsh. Direct and indirect methods. In J.K. Reid, editor, *Large Sparse Sets of Linear Equations*, pages 41–56. Academic Press, 1971.

[217] Debbie Whitfield and Mary Lou Soffa. An approach to ordering optimizing transformations. In *Proceedings of the second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 137–146, 1990.

[218] Harry A.G. Wijshoff. Sparse supercomputing. Research Proposal.

[219] Harry A.G. Wijshoff. Implementing sparse BLAS primitives on concurrent/vector processors: a case study. Technical Report no. 843, Center for Supercomputing Research and Development, University of Illinios, 1989.

[220] David S. Wise. Representing matrices as quadtrees for parallel processing. *Information Processing Letters*, 20:195–199, 1985.

[221] David S. Wise. Parallel decomposition of matrix inversion using quadtrees. In *Proceedings of the International Conference on Parallel Processing*, pages 92–99, 1986.

[222] David S. Wise. Undulant block elimination and integer-preserving matrix inversion. Technical Report 418, Computer Science Department, Indiana University, 1995.

[223] David S. Wise and John Franco. Costs of quadtree representation of nondense matrices. *Journal of Parallel and Distributed Computing*, 9:282–296, 1990.

[224] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proceedings ACM SIGPLAN Conference on Programming Languages Design and Implementation*, pages 30–44, 1991.

[225] Michael E. Wolf and Monica S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Algorithms*, pages 452–471, 1991.

[226] Michael J. Wolfe. Loop skewing: The wavefront method revisited. *International Journal of Parallel Programming*, 15:279–293, 1986.

[227] Michael J. Wolfe. Vector optimization vs. vectorization. *Journal of Parallel and Distributed Computing*, 5:551–567, 1988.

[228] Michael J. Wolfe. *Optimizing Supercompilers for Supercomputers*. Pitman, London, 1989.

[229] Michael J. Wolfe. *High Performance Compilers for Parallel Computers*. Addison-Wesley, Redwood City, California, 1996.

[230] Jingling Xue. Automating non-unimodular loop transformations for massive parallelism. *Parallel Computing*, 20:711–728, 1994.

[231] Mihalis Yannakakis. Computing the minimum fill-in is NP-complete. *SIAM J. Alg. Disc Meth.*, pages 77–79, 1981.

[232] David M. Young. *Iterative Solution of Large Linear Systems*. Academic Press, New York and London, 1971.

[233] Chuan-Qi Zhu and Pen-Chung Yew. A scheme to enforce data dependence on large multiprocessor systems. *IEEE Transactions on Software Engineering*, SE-13:726–739, 1987.

[234] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press, New York, 1990.

[235] Zahari Zlatev. *Computational Methods for General Sparse Matrices*. Kluwer, Dordrecht, 1991.

[236] Zahari Zlatev, Jerzy Wasniewski, and Kjeld Schaumburg. Y12M - solution of large and sparse systems of linear algebraic equations. In G. Goos and J. Hartmanis, editors, *Lecture Notes in Computer Science, No. 121*. Springer-Verlag, Berlin, 1981.

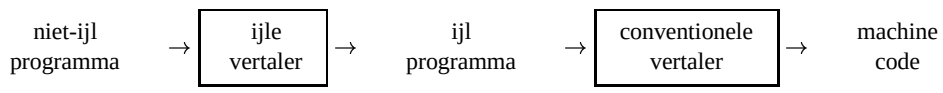
Samenvatting

Veel problemen binnen de natuurwetenschappen kunnen met behulp van matrices geformuleerd worden. Het oplossen van dergelijke problemen kan gedaan worden door bepaalde operaties op de matrices toe te passen. Als een computer gebruikt wordt voor het toepassen van deze operaties, is een representatie voor matrices nodig. De meeste imperatieve programmeertalen ondersteunen zogenaamde twee-dimensionale arrays, hetgeen de programmeur de beschikking geeft over een datastructuur die nauw verwant is aan een matrix en tevens abstraheert van het feit dat de echte representatie in het geheugen één-dimensionaal is.

In veel van bovengenoemde problemen komen echter zogenaamde **ijle matrices** voor, hetgeen matrices zijn die erg veel nul-elementen bevatten. Het moge duidelijk zijn dat, ook al is het mogelijk om een twee-dimensionale array te gebruiken als representatie voor een ijle matrix, het gebruik van een compacte representatie waarin alleen de niet-nul-elementen expliciet opgeslagen zijn het geheugengebruik van een programma drastisch kan reduceren. Bovendien kan een dergelijke representatie gebruikt worden om de totale tijd die nodig is om een programma uit te voeren aanzienlijk te reduceren door onnodige operaties op nullen niet uit te voeren. Voor grote ijle matrices kan het gebruik van een compacte representatie de enige mogelijkheid zijn om een probleem binnen redelijke tijd op te lossen.

Omdat dergelijke compacte representaties niet direct ondersteund worden in imperatieve programmeertalen, moet de programmeur datastructuren die wel ondersteund worden gebruiken om een compacte representatie expliciet te implementeren. Programma's waarin dit gebeurt zijn bijzonder moeilijk te ontwerpen en te onderhouden. Vertalers hebben tevens moeite om dergelijke programma's te optimaliseren. Omdat huidige herstructurerende vertalers in staat zijn bepaalde karakteristieken van een computer zeer succesvol te benutten, ligt het voor de hand te onderzoeken of een herstructurerende vertaler ook karakteristieken van de invoergegevens kan benutten om het geheugengebruik en executietijd van een programma te reduceren. In tegenstelling tot conventionele herstructurerende vertalers, die voornamelijk gericht zijn op het toepassen van *programma-transformaties*, moeten in deze aanpak ook *datastructuur-transformaties* toegepast worden.

In dit proefschrift wordt specifiek gekeken naar de mogelijkheid om een programma waarin simpelweg een twee-dimensionale array gebruikt wordt als representatie voor elke ijle matrix (een niet-ijl programma) automatisch te converteren naar een programma waarin compacte representaties gebruikt worden (een ijl programma). Om een efficiënt programma te verkrijgen, moet de ijle vertaler tijdens deze conversie rekening houden met de operaties die uitgevoerd worden op de ijle matrices, patronen waarin niet-nul-elementen in elke ijle matrix voorkomen en de karakteristieken van de computer waarop uiteindelijk het programma uitgevoerd zal worden. Een herstructurerende vertaler die een dergelijke conversie uit kan voeren wordt een **ijle vertaler** genoemd. Het automatisch gegenereerde ijle programma wordt vervolgens door een conventionele vertaler vertaald naar machine code voor een bepaalde computer, zoals hieronder is geïllustreerd:



Het automatisch genereren van een ijl programma geeft minder aanleiding tot het maken van programmafouten dan het expliciet ontwerpen van een ijl programma. Bovendien verlicht een ijle vertaler de taak van de programmeur. De ijle vertaler kan aan de hand van karakteristieken van de computer, de patronen waarin de niet-nul-elementen voorkomen in de ijle matrices en de operaties die op die matrices uitgevoerd worden, geschikte datastructuren kiezen en een efficiënt ijl programma genereren. Tenslotte, omdat een ijle vertaler de functionaliteit van een niet-ijl programma gemakkelijker kan doorzien dan de functionaliteit van een ijl programma, is het gegenereerde programma ook beter te optimaliseren.

De technieken die besproken worden in dit proefschrift zijn daadwerkelijk geïmplementeerd in een prototype ijle vertaler. Diverse simpele experimenten met deze prototype ijle vertaler zijn tevens opgenomen in dit proefschrift.

Curriculum Vitae

Aart J.C. Bik is geboren op 31 mei 1969 te Gouda. Op 2 juni 1987 behaalde hij het gymnasium β diploma, waarna hij informatica ging studeren aan de Rijksuniversiteit Utrecht. Op 29 augustus 1988 behaalde hij het propaedeutisch examen (cum-laude) en op 25 mei 1992 slaagde hij voor het doctoraal examen (cum-laude). Zijn afstudeerproject bestond uit het implementeren van een herstructurerende vertaler die ook gebruikt is tijdens zijn promotie onderzoek. Op 16 mei 1992 trad hij in dienst van de Nederlandse Organisatie voor Wetenschappelijk Onderzoek (NWO) als onderzoeker in opleiding (OIO) met als tijdelijke standplaats de Rijksuniversiteit Utrecht. Kort daarop werd zijn definitieve standplaats de Rijksuniversiteit Leiden. Gedurende vier jaar verrichtte hij promotie onderzoek op het gebied van herstructurerende vertalers en berekeningen op ijle matrices onder begeleiding van zijn promotor prof. dr. H.A.G. Wijshoff, hetgeen afgerond werd met dit proefschrift in mei 1996.