Universiteit Leiden

Master Computer Science

Countering the computation vs communication trade-off with compiler techniques

Name:	Dennis Buurman
Student ID:	s2027100
Date:	[12/03/2025]
Specialisation:	Advanced Computing and Systems
1st supervisor:	Kristian Rietveld
2nd supervisor:	Rob van Nieuwpoort

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

Parallel code inherently has a trade-off between computation and communication. Manually balancing this trade-off is complex and time-consuming. Many efforts have been made to automate the generation of efficient parallel code. However, these efforts often still require developers to work with directives for data layout and locality, limiting the compiler's options for optimizing computation and communication. In prior work, the tUPL parallel programming paradigm was used to express and optimize k-means clustering; resulting in multiple parallel implementations, differing in characteristics for communication and computation. Additionally, noticeable performance differences in computation and communication were measured between implementations. However, the presented derivation process was incomplete, as it did not contain transformations for the derivation and insertion of communication code. In this thesis, this derivation process is extended with a set of communication transformations that allow the derivation and insertion of different communication codes. Besides leading to a complete derivation, this also resulted in six new parallel implementation variants which showed a minor performance increase compared to prior work.

Contents

1 Introduction

In High Performance Computing (HPC) applications, communication between processes is a major concern for performance optimization [1], [2]. Message Passing Interface (MPI) [3] and Open Multi-Processing (OpenMP) [4] remain the standard practice for parallel programming [5], [6]. To achieve optimal performance in HPC applications, all processing components must be utilized and balanced as much as possible. This is especially important considering the cost of computation vs communication; where the latter can cost significantly more per operation [7]. Communication can be reduced by adding computation [8], [9]. However, balancing this trade-off between communication and computation is quite a complex and time-consuming effort to perform manually. In addition, the communication and computation trade-off is target-platform dependent, meaning that different platforms might require different balancing. Numerous research efforts have been made to automate parallelization, including the generation of necessary communication [10]–[12]. Despite this, communication code for parallel applications is typically still implemented and tuned by hand. Could this complex and time-consuming work be automated and still achieve on par or better performance compared to hand-written implementations? High Performance Fortran (HPF) [10], [13] is a notable example of an attempt at this. However, HPF was constrained to arrays. It failed mainly due to gaps in compiler technology, inconsistency of implementations, and missing features [14]. As a consequence, HPF code was commonly tuned by hand for specific machines and compilers in order to achieve reasonable performance.

The tUPL parallel programming paradigm [15] is a unifying optimization platform for programs expressed as tuple-based computations. The atomic nature of the tuple-based operations make the framework inherently parallel. tUPL does not allow explicit data structures to be specified. Originally developed for database query optimization, tUPL's use cases proved to be more versatile [16]–[18]. Compared to HPF, the tUPL compiler has much more freedom in the application of transformations to the initial specification. It is not constrained by failures in data dependency analysis as was the case with HPF. The inherent parallelism and freedom in transformation application allow tUPL to better manage the balance between communication and computation; resulting in more options to create optimized parallel implementations.

Hommelberg et al. [19] used tUPL to express and optimize k-means clustering [20] and PageRank [21]. This work resulted in optimized parallel `C/C++` MPI implementations that improved upon standard `C/C++` MPI implementations as well as outperform the state-of-the-art Hadoop[1] implementations. To achieve this, the algorithms were first expressed as tUPL specifications. Then, using a systematic approach, these tUPL-specifications were transformed into different `C/C++` MPI implementations. However, the transformation set used in this derivation process did not include transformations for communication code, resulting in an incomplete derivation process. Can we define transformations that enable the expression of communication code in tUPL?

In this thesis, a set of rigid transformations is introduced to derive communication code from a tUPL specification. A syntax extension was required to accommodate for the new transformations. The new transformations add the missing link in the derivation from an initial tUPL specification to distributed memory `C/C++` MPI implementations. Subsequently, we show that with the addition of these new transformations, the k-means derivation is now complete and that these transformations give rise to other k-means variants with a different balance between communication and computation. After, a case is made for the application of the derivation process to other algorithms. Furthermore, the performance of different derivations from the same initial tUPL k-means specification is investigated on different hardware and MPI rank configurations. Performance evaluations show that the best performing variant can differ between different hardware platforms. As a consequence, there is no "one size fits all" implementation. Using compiler transformations, we can generate different, optimized implementations for different target platforms, instead of hand-tuning on each platform.

The remainder of this thesis is structured as follows. Section 2 describes current dominant parallel programming models, previous efforts of automation, and the tUPL programming paradigm. Section 3 covers a

[1] `https://hadoop.apache.org/`

reproducibility study on the work on k-means from Hommelberg et al. [19]; the basis of this work. Section 4 provides a description of new tUPL communication transformations. Section 5 applies the new transformation to k-means, showcases new k-means implementation variants derived, and demonstrates the general applicability of the new transformations by applying them to a sorting algorithm. The performance of the new k-means variants is evaluated in Section 6. Section 7 contains a brief discussion. Section 8 ends this thesis by providing conclusions and directions for future work.

2 Background and Related Work

This section briefly describes concepts related to HPC, discusses previous efforts to automate parallel programming, and provides a short description of the tUPL programming paradigm.

2.1 Message Passing Interface (MPI)

MPI is a portable message-passing library interface specification designed for the message-passing parallel programming model [3]. It describes operations that handle data transfers between address spaces of different processes. With MPI, communication between processes is achieved using messages. MPI is a specification, not a language. Its operations can be expressed in the C/C++ and Fortran programming languages. Other languages, such as Python or Java, can use MPI through a library interface. A variety of MPI implementations exist. Open MPI [22] is a popular open source implementation that supports a wide range of machines and interconnects while achieving very high performance. To this day, MPI remains the standard for distributed-memory HPC applications [5], [6].

Kanor [23] is an extension for C++ with declarative semantics. It is designed to simplify programming communication explicitly as compared to hand-tuned MPI. In Kanor, developers can declaratively specify explicit communication patterns of Bulk Synchronous Parallel (BSP) programs.

Kokkos [24] is a C++ library that enables performance portable computation kernels for manycore architectures. It unifies abstractions for fine-grained data parallelism and memory access patterns, allowing implementations of a kernel to run efficiently on multiple types of hardware architectures. Kokkos supports popular backend models such as CUDA, HIP, SYCL, HPX, OpenMP and C++. As Kokkos and MPI are often used together, Suggs et al. [25] integrated Kokkos within an MPI implementation to increase usability.

2.2 Open Multi-Processing (OpenMP)

OpenMP is an application programming interface (API) for shared-memory parallel programming in C/C++ and Fortran [4]. It was designed to provide a portable shared-memory alternative to the distributed-memory MPI. OpenMP makes use of threads to perform tasks in parallel. OpenMP programs switch between sequential and parallel code segments using the fork-join model [26]. Programs run sequentially, but fork into parallel threads that execute their designated tasks. When all tasks are completed, the threads join and continue the execution sequentially until another fork is encountered. As its distributed-memory counterpart, OpenMP remains dominant in shared-memory parallel programming [27].

2.3 High Performance Fortran (HPF)

With its constructs that allow parallel computation, HPF [10] was the first effort to a standardized high-level data-parallel programming system based on the Fortran programming language [28]. The goal of HPF was to present an accessible machine-independent programming model for scalable parallel computing systems that produced code with performance comparable to hand-written MPI code [14]. The model was fully centered around arrays. In HPF, the developer set distribution directives, which specified the data distribution of the arrays. Implicit parallelism was provided by the “owner-computes” rule. This rule stated that calculations on distributed arrays are assigned such that each calculation is carried out by processors that own the data. Communication was generated implicitly when a calculation involved elements owned by different processes.

tUPL provides more flexibility and control compared to HPF. In tUPL, data structures are generated during compilation, leading to more flexibility in data distribution. This results in more options for optimization. The added flexibility also provides more control over code generation. These differences provide tUPL with better tools for balancing the trade-off between communication and computation. That is, we have the possibility to reduce the total cost of communication by substitution with calculation, as is done with the recalculation communication scheme from Hommelberg et al. [19]. This is advantageous considering that the gap between communication and computation cost is widening [7]. Scalability of HPC applications is limited by communication costs. Increasing concurrency results in a decrease in locality [1]. At some point, the increase in communication cost is higher than the decrease in cost of computation. This results in increased execution time instead of the expected decrease. Therefore, it is beneficial to optimize and balance the computation vs communication trade-off.

2.4 Partitioned Global Address Space (PGAS)

Ultimately, HPF did not succeed in its goal to be a standard data-parallel programming system. After the fall of HPF, DARPA started the HPCS language project [29]. DARPA funded the creation of new languages focused on parallelism. These include Cray's Chapel [11], [30] and IBM's X10 [12]. Chapel and X10 aimed to provide a high-level Object-Oriented (OO) language with extensive support for parallelism. Both languages adopt the Partitioned Global Address Space (PGAS) model [31]. In this model, multiple processes execute an algorithm in parallel; communicating through a simulated shared memory. The shared memory is actually an interconnected network of memories used to decrease data access latency. Abstractions are added to PGAS languages in order to utilize both local and remote data access. PGAS is a middle ground between strict message-passing (MPI) and shared-memory (OpenMP [4]) models.

Chapel was designed to tighten the gap between sequential and parallel programming by incorporating well-known sequential programming language features. Next to this, Chapel tried to simplify parallel programming by adding a multi-threaded execution model. Here, parallelism is denoted as independent computation threads. High-level abstractions of parallelism are implemented by the compiler. This model moves the burden of thread management from the developer to the compiler.

X10 focuses on increasing software productivity for Non-Uniform Cluster Computing (NUCC) systems without sacrificing much in terms of performance. The design of the X10 language tries to balance safety, analyzability, scalability, and flexibility. The popular OO programming language Java [2] was used as a basis. Instead of the single uniform heap used by Java, X10 adopted the PGAS model to allow scalable parallelism. X10 also introduces asynchronous activities to handle thread parallelism and asynchronous data transfer.

In practice, PGAS languages like Chapel and X10 are rarely used [6]. MPI and OpenMP, regardless of their complexity, still remain the standard for HPC parallel programming.

With HPF, Chapel, and X10, the developer still needs to specify the data layout and locality, limiting compiler options. tUPL specifications do not contain such data layout directives. The abstract approach to tUPL specifications allows the tUPL compiler to apply more aggressive transformations to the original specification.

2.5 Charm++

`Charm++` [32] is an OO `C++`-based portable programming language developed at the University of Illinois. In short, `Charm++` is `C++` without global variables, with extensions to enable parallel computations. `Charm++` uses *chares* that represent parallel processes which communicate using messages. Additional abstractions are provided to allow data sharing in different modes. There is a clear separation between sequential and parallel objects. This provides the developer with control over the choice of local or remote execution of operations. With this model, it is important for the developer to understand the costs and benefits of local vs remote execution. In other words, the developer is mainly responsible for balancing computation and communication.

[2] `https://www.java.com/`

2.6 Linda

Linda [33], which first appeared in 1986, is a coordination language that enables distributed shared-memory (DSM) parallelism through a tuple space [34]. Its operators are meant to be added to other programming languages, such as C [3]. The Linda operators use tuple-based operations that allow cooperation between processes. Although both Linda and tUPL utilize tuple spaces, the two languages are fundamentally different. Linda is aimed at providing a general set of primitives that can be implemented in multiple languages, while tUPL provides specifications that can be transformed into parallel implementations of different languages.

2.7 tUPL

tUPL [15], [35], previously known as the Forelem framework, features atomic, tuple-based operations where data is accessed through a tuple space. In tUPL, loops iterate over a set of tuples from a so-called tuple reservoir. Tuple reservoirs are conceptual, meaning that tuples within a reservoir do not have a physical or virtual address. There is no strict execution order in loop-based computations. A tuple `t` contains data fields or index fields. Index fields reference other tuples or address data stored in shared spaces. Shared spaces are also conceptual. A shared space `A` has an affine address function `F_A`. The address function maps tuple indices to a unique (conceptual) location in this shared space. This location can be used to store data. `A[F_A[t]]` denotes a unique address in the shared space `A`. In the remainder of this thesis, we will refer to `A[F_A[t]]` using array-notation `A[t]=x`, where x is stored at the address $A[F_A[t]]$. The abstract notation does not allow explicit data structures to be specified.

2.7.1 Loop constructs

tUPL is centered around the `forelem` and `whilelem` loop constructs. The `forelem` loop executes its loop body exactly once for each tuple in the given reservoir. Tuples are sampled in arbitrary order. For a reservoir `A` with tuples containing field `f1`, we can sum `f1` for each point using the loop:

```
forelem (t ∈ A)
    sum += t.f1
```

The `whilelem` loop continues to sample tuples arbitrarily until all tuples in its reservoir result in 'no-op' operations. As a result, a `whilelem` loop can execute a single tuple multiple times in a row. A loop body can contain one or more if-statements guarding operations in the loop body. The if-statements can be viewed as stopping conditions. Exchanging the `forelem` with a `whilelem` loop in the previous example would result in an infinite sum due to the properties of the `whilelem` loop. The `whilelem` construct is best employed for conditional sequences of operations, such as sorting. Consider a reservoir `B` consisting of tuples with fields $f1, f2$ containing addresses of adjacent data values accessed through a reservoir `C`. A simple sorting on the values of `C` would then be accomplished with:

```
whilelem (⟨f1, f2⟩ ∈ B)
    if (C[f1] > C[f2])
        swap(C[f1], C[f2])
```

This loop would continue swapping adjacent pairs until all pairs are correctly ordered. The values are then sorted.

Sampling of tuples is arbitrary, but relies on a Just Scheduling method [36]. Just scheduling assures that, on infinite scale, continuously enabled transitions will be taken infinitely many times. That means, each 'enabled', not a no-op, tuple will be sampled at some point.

[3] https://www.comp.nus.edu.sg/~wongwf/linda.html

The out-of-order loop execution and lack of explicit data structures has big implications for the compiler. It provides more freedom for compiler transformations, but the transformation conditions need to be carefully evaluated.

2.7.2 Transformations

Transformations relevant to this thesis are orthogonalization, reservoir splitting, localization, loop blocking, materialization, and concretization. **Orthogonalization** introduces data access order by partitioning or grouping on one or more tuple fields of a loop. For each of the selected partitioning fields, an outer loop is added. Each loop iterates over the different values of their corresponding field. The inner-most loop then loops over tuples where the selected fields equal all sampled values in the outer loops. Take a tuple reservoir `A` with fields `f1, f2, fn` and the following loop:

```
forelem (t ∈ A)
    ... t ...
```

Orthogonalization on `f1` would then result in

```
forelem (ii ∈ A.f1)
    forelem (t ∈ A.f1[ii])
        ... t ...
```

effectively ordering the traversal of tuple data fields. The tuples are now visited in groups that have the same value for `f1`. Orthagonalization prepares the code for materialization.

Reservoir splitting simply splits a tuple reservoir into `i` subreservoirs that can be processed in parallel. Any partitioning split `S` of reservoir `A` is valid, as long as it adheres to $\bigcup_i \texttt{S(A)}_i\texttt{=A}$. Reservoir splitting enables a distribution of shared spaces over different processes.

Localization stores the data from shared spaces directly in the tuples instead of separately. In other words, the data from shared spaces is added to tuples in the form of new data fields. For example, when applying localization to k-means, each tuple contains both the coordinates and current cluster membership variables instead of separately. Localization results in different data structures to be generated during a later stage in compilation.

Loop blocking partitions a reservoir into x blocks, and introduces a loop that iterates over these blocks. Effectively, this results in the creation of data partitions. This affects the generation of data structures. The partitionings created by loop blocking are not fixed.

Materialization fixes the execution order of a loop. Tuples in the loop's reservoir receive a unique index and are now accessed through an array with integer subscripts. This transforms an unordered reservoir `t∈A` into an array `A` with indexes `i ∈ [0, n-1]`, where `n` is the amount of tuples in the reservoir. The order in which tuples receive an index is not fixed. A loop in materialized form can be transformed further to optimize data access and storage.

In the **concretization** transformation, loops from a materialized specification are transformed into regular, `C`-like, `for` loops. The iteration order of the loops is chosen (usually `0` to `n`) and the data structures selected by the optimization process are generated.

Different transformation permutations yield different implementations with different data distributions, communication schemes, and more. The transformations, except for concretization, can be applied multiple times, leading to multiple possible final implementations. The order in which transformations are applied is not set, except for concretization. The concretization step is always the final transformation before transitioning to code generation. During code generation, code for communicating (partial) results is inserted. The code generation step produces the final result; a `C/C++` MPI implementation.

2.7.3 Application to the k-means algorithm

In [19], the above transformations were applied to a tUPL k-means specification, resulting in two different tUPL specifications: non-localized and localized, as showcased in Algorithm 1 and Algorithm 2 respectively. Both algorithms are in materialized form. In these algorithms, PM[i] is the cluster membership for each point i, k the amount of clusters, PC (short for PCOORDS) the reservoir of point coordinates, PM_C[m] (short for PM_COORDS) the cluster center coordinates of cluster m, and PM_S[m] (short for PM_SIZE) the amount of point members of cluster m. PT[i] is the result of applying localization to the cluster membership and coordinates. Both data values are added to each tuple in the materialized reservoir PT.

Algorithm 1 k-means tUPL specification after applying orthagonalization, reservoir splitting, and materialization

```
**whilelem** ($i \in [0, |\text{PM}| - 1]$)
  **forelem** ($m \in [0, k - 1]$)
    **if** ($\text{PM}[i] \mathrel{!=} m \;\&\&\; \text{dist}(\text{PC}[i], \text{PM\_C}[m]) < \text{dist}(\text{PC}[i], \text{PM\_C}[\text{PM}[i]])$) {
      $\text{PM\_C}[\text{PM}[i]] = (\text{PM\_C}[\text{PM}[i]] * \text{PM\_S}[\text{PM}[i]] - \text{PC}[i])/(\text{PM\_S}[\text{PM}[i]] - 1)$
      $\text{PM\_S}[\text{PM}[i]] -= 1$
      $\text{PM\_C}[m] = (\text{PM\_C}[m] * \text{PM\_S}[m] + \text{PC}[i])/(\text{PM\_S}[m] + 1)$
      $\text{PM\_S}[m] += 1$
      $\text{PM}[i] = m$
    }
```

Algorithm 2 k-means tUPL specification after applying orthagonalization, reservoir splitting, localization, and materialization

```
**whilelem** ($i \in [0, |\text{PT}| - 1]$)
  **forelem** ($m \in [0, k - 1]$)
    **if** ($\text{PT}[i].c\_x \mathrel{!=} m \;\&\&\; \text{dist}(\text{PT}[i].x, \text{PM\_C}[m]) < \text{dist}(\text{PT}[i].x, \text{PM\_C}[\text{PT}[i].c\_x])$) {
      $\text{PM\_C}[\text{PT}[i].c\_x] = (\text{PM\_C}[\text{PT}[i].c\_x] * \text{PM\_S}[\text{PT}[i].c\_x] - \text{PT}[i].x)/(\text{PM\_S}[\text{PT}[i].c\_x] - 1)$
      $\text{PM\_S}[\text{PT}[i].c\_x] -= 1$
      $\text{PM\_C}[m] = (\text{PM\_C}[m] * \text{PM\_S}[m] + \text{PT}[i].x)/(\text{PM\_S}[m] + 1)$
      $\text{PM\_S}[m] += 1$
      $\text{PT}[i].c\_x = m$
    }
```

From both specifications, a translation was made to `C/C++` MPI code using two different communication schemes. This resulted in four different final implementations. As can be seen in the Algorithms 1, 2, communication code is not explicitly specified. This code is added during code generation. The applied communication schemes are called recalculation and derived.

In the recalculation scheme, after each algorithm iteration, a full recalculation of results is performed. For k-means, this refers to a full recalculation of cluster means and sizes after each full point reassignment cycle. In [19], the following recalculation scheme fragment was provided:

```
forelem (⟨x⟩ ∈ X)
  SIZE[M[x]] += 1
  forelem (⟨n,y,i⟩ ∈ T_e.⟨n,y⟩[⟨M[x],x⟩])
    VAL[n,i] += DATA[y,i]

forelem (⟨m,i⟩ ∈ M)
  VAL[m,i] = VAL[m,i] / SIZE[m]
```

Here `X` is the data point tuple reservoir, $\texttt{T}_e$ an extended tuple reservoir consisting of all tuples $\langle \texttt{n,y,i} \rangle$ where $0 \leq n < k$ is one of the clusters, $0 \leq y < N$ one of the N data points, and $0 \leq i < d$ an index with d the dimension of the data points. `DATA[x]` denotes the coordinates of data point `x`, `VAL[m]` the cluster center of cluster `m`, and `SIZE[m]` the size of cluster `m`.

The above code fragment recalculates the cluster means based on a (local) partition of points $\texttt{X}_p$. Cluster centers are recalculated using only the values of `M` and `DATA` stored locally. By adding reduction code, the ranks can communicate the sums and local sizes of the each cluster.

However, the problem here is that there is no transformation defined that can transform a tUPL specification to include a communication scheme. Therefore, it is not specified how communication, such as the above recalculation code, is automatically derived and inserted during code generation.

The derived scheme communicates local differences of each process in order to iteratively update to new global values. With k-means, after each cycle, the difference between current local and previous global versions of the mean values sum and mean sizes is communicated. As a result, only the difference is communicated, which can be used to update global cluster means and sizes.

Unfortunately, there is no reduction code provided for the derived communication scheme applied to k-means. As is the case with the recalculation scheme, the derived scheme suffers from the lack of defined communication transformations that allow it to be added to a specification. In Section 4, we will introduce transformations to deduce reduction code and generate communication code.

3 Reproducibility Study

To establish a performance baseline before refining the transformation process, experiments 1 and 2 from [19] were reproduced on updated hardware. Experiments 1 and 2 evaluate performance by varying input sizes and thread counts respectively. Originally, these experiments ran on the DAS4 compute cluster[4]. To get a more representative performance baseline, the experiments were repeated on the more recent DAS5[5] and DAS6[6] clusters. Table 1 shows the hardware available for the DAS systems. Note that DAS6 at Leiden was used before the hardware upgrades that took place in October 2024.

Cluster	Nodes	CPU	Cores	Threads	Max Threads
DAS4/Leiden	16	2x Intel E5620	4	8	256
DAS5/Leiden	24	2x Intel E5-2630v3	8	16	768
DAS6/Leiden	18	1x AMD EPYC-2 (Rome)	24	48	864

Table 1: Hardware of the DAS systems. Cores and threads are denoted per CPU.

Datasets used during the experiments were generated by the provided random data generator. As input, this generator takes the total number of points, the point dimension, and the amount of clusters. Intended cluster centers are generated using a uniform distribution on the interval [0, 10]. Points are first assigned an intended cluster membership from a uniform distribution. After, coordinates are assigned to the points using a normal distribution with the assigned cluster center as a mean. Each dataset used in the experiments contained **4 clusters** with data points of **dimension 4**. For comparative reasons, a stopping condition of 0.0001 was added to the implementations. Input size is denoted as 2^x data points, where x is the denoted input size. Threads are configured to maximize Core/CPU/Node utilization. For example, on DAS4, 8 threads will be run on 1 node. The execution time is defined as a sum of the read time, calculation time, and write time. The calculation time is established by taking the worst calculation time for each run of the experiment. In this thesis, the focus lies on the optimization of calculation times. This is because repeated runs re-use data read into memory. Experiments 1 and 2 compare the worst calculation times of 4 different k-means implementations. The k-means implementations are named as follows:

- Implementation 1: derivation from Algorithm 1 using the recalculation communication scheme.
- Implementation 2: derivation from Algorithm 1 using the derived communication scheme.
- Implementation 3: derivation from Algorithm 2 using the recalculation scheme.
- Implementation 4: derivation from Algorithm 2 using the derived communication scheme.

The original DAS4 results from [19] can be seen in Figures 1 and 2. While all implementations scale nicely, the calculation time between implementations shows a growing performance difference between the best and worst performing implementation variants.

3.1 Performance Variability

DAS5 and DAS6 results of experiments 1 and 2 showed unexpectedly high performance variability. Initial runs showed significantly different performance; not only between the benchmark and original experiments, but also between the benchmark samples themselves. Identical configurations yielded significantly different results.

Figure 3 shows the results of experiment 1, run on DAS5. The results are averaged over 30 runs instead of the original 10. Across the board, performance does not show as clear a trend as on DAS4. Implementation

[4] https://www.cs.vu.nl/das4/
[5] https://www.cs.vu.nl/das5/
[6] https://www.cs.vu.nl/das6/

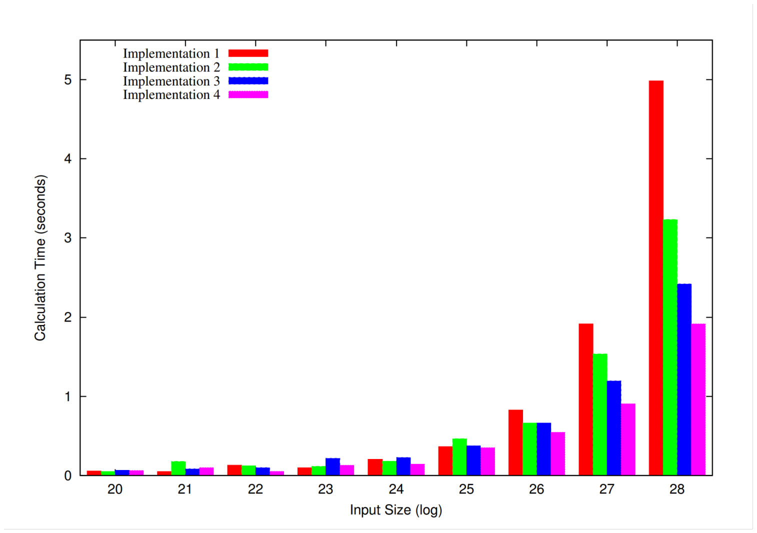


Figure 1: The calculation time of Implementations 1 to 4. This experiment ran using 64 threads: 8 nodes with each 8 threads. Input sizes 20 to 28 were used. A convergence delta of 0.0001 was used. Experiment ran on DAS4. Results were averaged over 10 runs. Graph sourced from Hommelberg et al. [19].

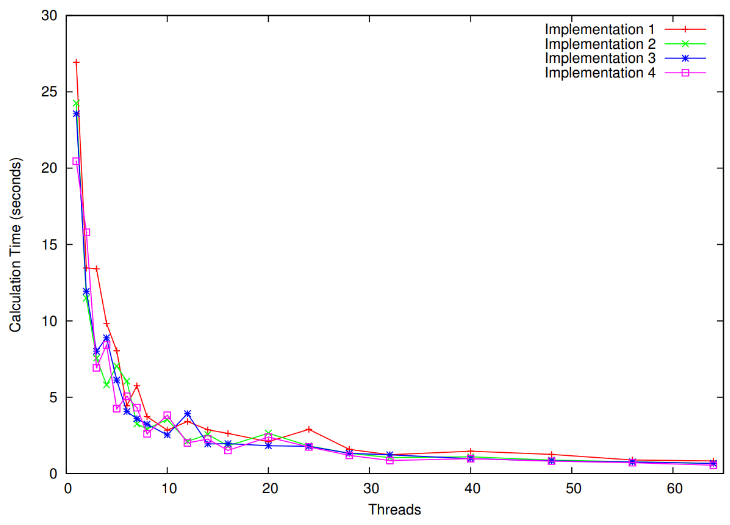


Figure 2: The calculation time of Implementations 1 to 4, using a varying numbers of threads, on a dataset of input size 26. A convergence delta of 0.0001 was used. Experiment ran on DAS4. Results were averaged over 10 runs. Graph sourced from Hommelberg et al. [19].

3 performs best on size 27, while it is second to last place on size 28. The statistical soundness of these results was measured using a boxplot and 95% Confidence Interval (CI) plot, as shown in Figures 4 and 5 respectively. From these figures, it is immediately evident that there is high variability in performance. There are many outliers and the CI plot does not converge. Even when increasing the runs to 100, the performance variability persists, and no accurate performance numbers can be presented. This is a major problem for this research. Because if the results have such high variability, there is no way to present accurate performance of different, maybe newly added, implementation variants.

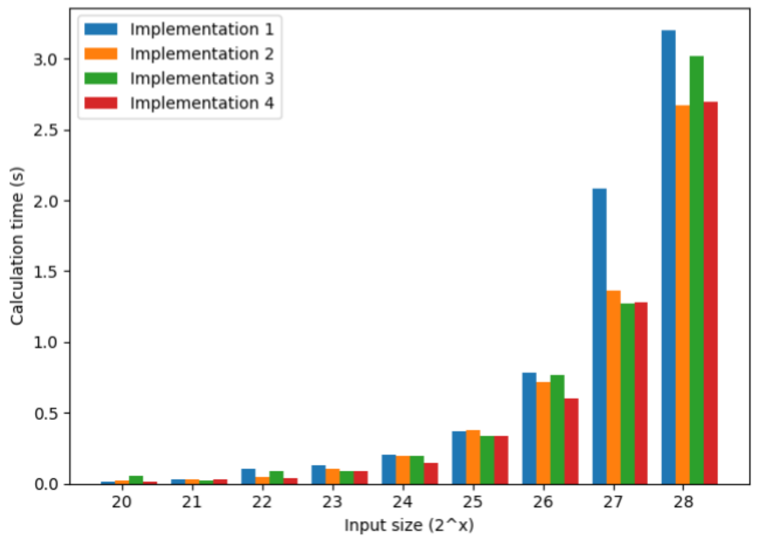


Figure 3: The calculation time of Implementations 1 to 4. This experiment ran using 64 threads: 8 nodes with each 8 threads. Input sizes 20 to 28 were used. A convergence delta of 0.0001 was used. Experiment ran on DAS5 cluster. Results were averaged over 30 runs.

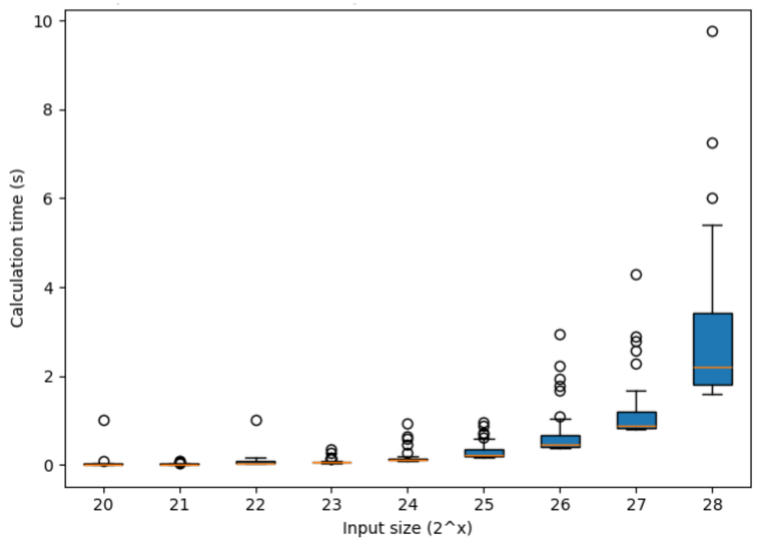


Figure 4: Box plots comparing calculation times per run, per input size on Implementation 3. Results show the performance distribution over 30 runs. This experiment ran using 64 threads: 8 nodes with each 8 threads. A convergence delta of 0.0001 is used. Experiment ran on DAS5 cluster.

3.1.1 Removing random initialization

Comparing two runs with identical configurations revealed that each run within the same experiment was initialized using a different random seed. This randomized seed was used to generate initial cluster centers. However, the performance of k-means highly depends on the selected initial cluster centers [37]. Different initial clusters will results in different point reassignments, which has a large effect on the number of iterations to convergence. Therefore, using random initialization in the algorithm results in many outliers, as well as the inability to create a CI plot that converges at a reasonable rate. As a consequence of this random

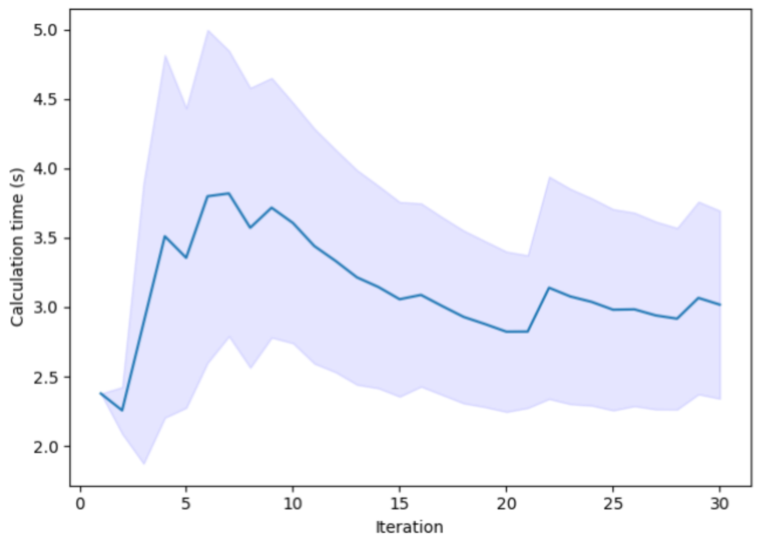


Figure 5: 95% CI plot over 30 runs on Implementation 1 with input size 28. The blue line represents the mean calculation time, while the area around it represents the CI. Experiment ran using 64 threads: 8 nodes with each 8 threads. A convergence delta of 0.0001 is used. Experiment ran on DAS5 cluster.

initialization, the number of iterations k-means required to converge ranged from around 5 to well above 50. The presented DAS4 results used randomized initialization. Therefore, the DAS4 results do not present an accurate evaluation of implementation variant performances; meaning they should not be used to compare (relative) performance between different DAS clusters. Additionally, the seeds used to acquire the DAS4 results are not reported, making it impossible to reproduce these results. Using a fixed initialization seed for each experiment removed the iteration difference between different runs in the same experiment.

3.1.2 Dataset consistency

However, after using a set initialization seed, the number of iterations between different input sizes were still far from consistent. As can be seen in Figure 6, results between input sizes do not scale proportionally for Implementation 3. This occurs because of a wide range between iterations till convergence. For example, Implementation 3 needed 9 iterations to convergence for input size 27, while it took 32 iterations for size 28. As a result, the worst calculation time for size 28 in Implementation 3 seems peculiarly high. Other configurations yielded different anomalies. Multiple factors can cause these anomalies to occur. The most obvious factor is initial cluster center selection. However, this should not be the case as initial center selection used the now fixed initialization seed. The dataset and its generation procedure is another factor that highly impacts k-means performance. A different layout of data points results in different point reassignments. Therefore, it is important to keep a consistent data pattern between the different datasets (input sizes). The experimental results from Figures 1,2,3,6 all used the same dataset generator. Parameters for this generator included a seed, the number of data points to generate, the amount of clusters to generate, and the dimension of each data point. This generator was run for each different input size (2^x points). All datasets were generated with the same seed. A property of this method is that increasing x by 1 doubles the amount of points that need to be sampled from the generator. Assuming the same generator seed is used, the first 2^x points after an input size increase by 1 will be the same. However, the remaining $2^{x+1} - 2^x$ points will be different. This changes the data pattern between generated datasets of different sizes, affecting the behaviour of k-means. To solve this problem, the data pattern between differently sized datasets needs to be more consistent.

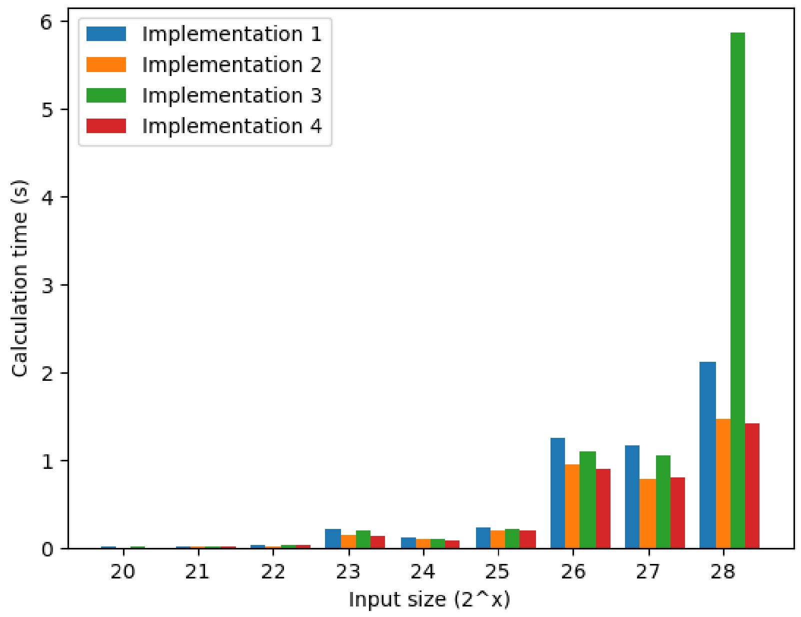


Figure 6: Input size variation on implementations 1 to 4. Results are averaged over 10 runs. Experiment ran using 64 threads: 8 nodes with each 8 threads. A convergence delta of 0.0001 was used. Experiment ran on DAS5. Initialization seed is set to 2363385789.

One option to improve consistency is to apply point duplication. Point duplication is based around the idea that the amount of points doubles after the input size is increased by 1. This method starts by creating a dataset of a default size. Then, for each incremental size increase, a copy is created for each point vector. This approach comes with one downside. Namely, reducing the dataset size below the chosen default size. Reducing below this size would cause the data layout to change as points would need to be removed. An easy workaround is to set the default size to the smallest requested size. However, when is a dataset too small? Take Figure 1 for example. With sizes [20-28] and points of dimension 4, the amount of points would be increased from $2^{20} * 4 \approx 4$ million to $2^{28} * 4 \approx 1$ billion. The points are saved as doubles, taking 8 bytes each. This means that a size 20 dataset requires $2^{20} * 4 * 8 \approx 33$ million bytes, or 33MB. DAS5 has 20MB cache on each CPU (40MB per node) [7]. In other words, the smallest dataset fits in the cache memory of 1 node. Therefore, the input size range was adapted to [24-28], with size 24 the default dataset size. Size 24 requires $2^{24} * 4 * 8 \approx 536$ million bytes (536MB), corresponding to 14 DAS5 nodes, to fit the dataset in cache.

3.1.3 Generating initial means

Although the initial means can now be consistently generated at run-time by providing the same seed, a better option is to include initial means in the datasets. This not only improves reproducibility, but also allows a fair comparison with other implementations that can use the same provided initial means. Therefore, initial mean sets were generated alongside the datasets. The method used for selecting initial means is based on k-means++ [38]. In k-means++, points are iteratively chosen as centroids, the initial means. The first centroid is randomly selected from all points. The other centroids are iteratively chosen from remaining points. The next centroids are selected from the remaining points by sampling a probability distribution; where the

[7] https://ark.intel.com/content/www/us/en/ark/products/83356/intel-xeon-processor-e5-2630-v3-20m-cache-2-40-ghz.html

probability of each point is based on its distance to all previously selected centroids. Larger distances receive a higher probability. By forcing distance between the centroids there is a higher probability of initial centroids to belong to different clusters. The applied method follows k-means++, except it changes the probability distribution sampling with a max distance. The initial centroid is sampled using a uniform distribution. This distribution is initialized using the same seed provided for the point sampling generator. This ensures that each execution of the dataset generator with identical parameters results in identical datasets. A benefit of the point duplication upscaling method is that initial means can be reused between different sizes. Doubling coordinates does not affect the cluster means. As a result, both the data pattern and initial means are more consistent between differently sized datasets. A downside of k-means++ is that each iteration needs to loop through all data points, making this method inefficient for larger datasets. However, considering the generated cluster means will be saved and reused, this is not a problem. On demand, other initialization methods can easily be added to the dataset generator.

Adding mean initialization through a file required a small workaround in order to maintain the same control flow as before. The k-means algorithms initialized the means by randomly assigning points to a cluster. When initial means are provided, this is not needed. The recalculation scheme works without problems, as it fully reassigns all points and updates the means accordingly. However, the derived scheme updates using old values, which are not present when means are provided. To solve this, the derived scheme variants employ one recalculation cycle at the start. Another solution would be to include an initial point assignment based on the provided means.

3.2 Performance Baseline

With the dataset and algorithm changes, we can create a more reliable performance baseline. Because of the dataset and algorithm changes, the experimental setup slightly differs from DAS4 experiments 1 and 2. All datasets used were created using seed 971 with 10 sets of initial means, and points generated in 4 dimensions and 4 clusters. Each experiment run is initialized with the seed 340632450. The stopping condition, or convergence delta, is kept at 0.0001. These experiments were run using initial means sets 3 and 7, see Appendix A.

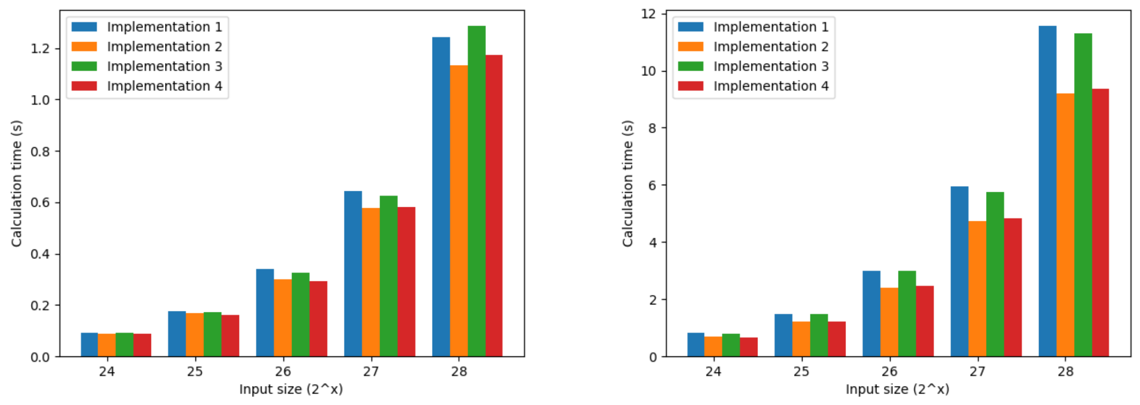
3.2.1 Experiment 1: performance for increasing problem size

Because of point duplication, the input sizes used in experiment 1 change from [20-28] to [24-28]. Experiment 1 is still run with 8 nodes, 8 threads each. The DAS5 results are shown in Figure 7. The best performing initial means set generated was set 3, which required 5 iteration to converge. These results show that the derived variant improves performance, while localization has only a small impact. On DAS5, localization may even decrease performance slightly, as was the case with implementation 4, the localized variant of implementation 2.

Initial means set 7, on the other hand, is the worst performing set. Using mean set 7, iteration counts sometimes slightly differed between different input sizes. However, the iteration count for a given configuration was deterministic, and thus reproducible. The required iterations ranged from 68 to 71. Different input sizes may be split differently, leading to more or less favorable distributions. Still, the iterations to convergence are more consistent than before.

Compared to mean set 3, mean set 7 only contains one different initial point, namely point 0. This shows k-means' sensitivity to initial mean selection. The notable increase in iterations to convergence logically results in a higher execution time. However, for mean set 7, the relative performance differences between implementations are increased compared to mean sets 3.

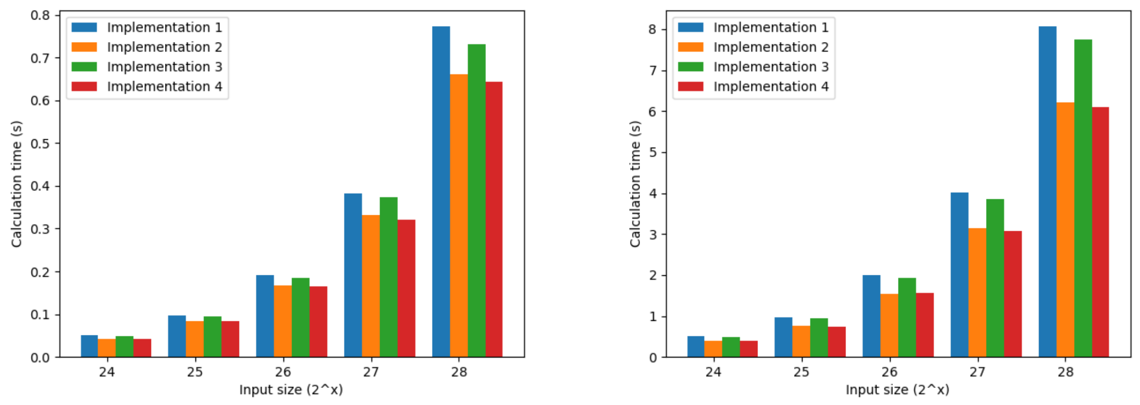
Results from experiment 1 on DAS6 are shown in Figure 8. As expected, the iteration counts of the runs on DAS5 and DAS6 are identical. As is the case with DAS5, the derived scheme outperforms the recalculation scheme. However, the localized Implementations 2 and 4 perform significantly better on DAS6 compared to



(a) (b)

Figure 7: Input size variation on implementations 1 to 4. Results are averaged over 10 runs. Experiment ran using 64 threads: 8 nodes with each 8 threads. A convergence delta of 0.0001 was used. Experiment ran on DAS5. Initialization seed is set to 340632450. Initialized using mean set 3 (a) and mean set 7 (b).

DAS5. On DAS6, localization improves performance across the board. The only instance where localization improves performance on DAS5 is found in the mean set 7 results. Here, Implementation 3 performs slightly better than Implementation 1. As a result, the best performing implementation differed between the two clusters. The implementations achieving the highest performance are Implementation 2 on DAS5 and Implementation 4 on DAS6. The differences in relative performance show the importance of optimizing implementations for specific compute clusters.

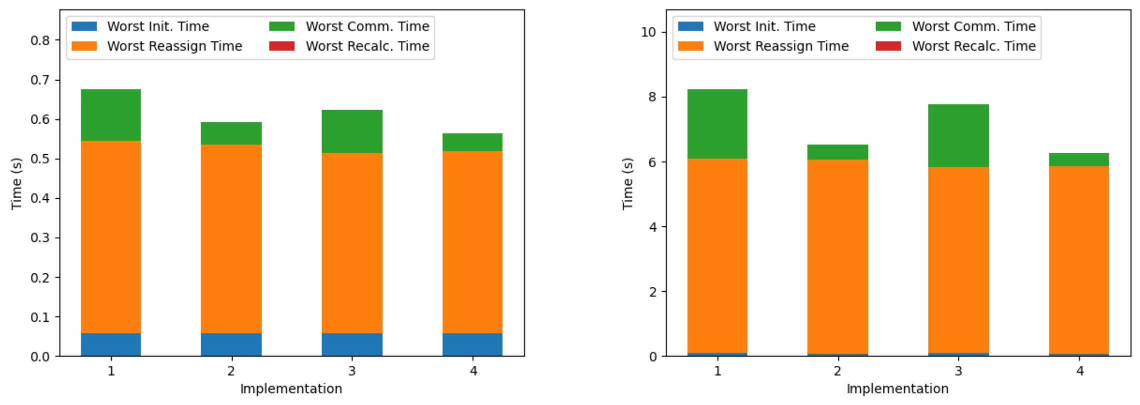


(a) (b)

Figure 8: Input size variation on implementations 1 to 4. Results are averaged over 10 runs. Experiment ran using 64 threads: 8 nodes with each 8 threads. A convergence delta of 0.0001 was used. Experiment ran on DAS6. Initialization seed is set to 340632450. Initialized using mean set 3 (a) and mean set 7 (b).

Figure 9 shows the time spent per calculation component from input size 28 of DAS6 benchmark experiment 1. Note that the mean recalculation time is not clearly visible because it is dwarfed by other components.

Mean recalculation and initialization are constant for a given dataset. The rest of the components scale with iteration count. As a result, the init. time results in a minor relative performance difference between the two experiment runs. However, the majority of calculation time is spent in the reassign and communication components. Comparing time growth per component, see Tables 5, 6, see Appendix C, shows that, while the reassign time grows similarly between implementations, the communication time grows almost twice as fast with implementation 1 compared to implementation 2. In other words, there is a growing gap between the communication time of implementation 1 and 2 when increasing the iteration count. Moreover, the communication time of implementation 1 grew at a slightly higher rate than the iteration increase rate itself. As a result, the relative performance between the derived and recalculation implementations increases with the iteration count.



(a) (b)

Figure 9: Stacked bar plot of input size variation experiment. Bars show component timing per implementation from input size 28 from mean set 3 (a) and mean set 7 (b). Results are from runs presented in Figure 8.

Figures 10, 11 respectively show the box plots and size 28 CI plot of the DAS5 benchmark experiment 1, see Figure 7 (a). This time, outliers only occasionally occurred and were not as severe. In both graphs, the deviation from the mean is minimal. The CI converged nicely within 10 runs, as expected. Note that the y-axis on the CI plot is zoomed in to milliseconds. DAS6 shows similar results, see Appendix C. These statistics show that the presented benchmark results are indeed an accurate representation of performance.

3.2.2 Experiment 2: strong scaling

For experiment 2, performance scaling is tested using various thread count configurations. On DAS6, the thread count ranges from [2-768]. On DAS5, the range is smaller, namely [2-384]. This is a significant step up compared to the 64 thread maximum used in DAS4. The increased thread count allow us to see if the trend in scalability continues when increasing the thread counts well over 64. The input size is increased from 26 to 28. Results for DAS5 and DAS6 are shown in Figure 12. The DAS5 results show the same scaling trend compared to DAS4 for the first 64 threads. This trend continues past the 64 thread limit from DAS4. However, the speedup factor steadily decreases above 64 threads. For example, with DAS6 Implementation 1, increasing the thread count from 32 to 64 results in a speedup factor of around 1.99, almost halving the calculation time. This level of scaling is not sustainable with higher thread counts. When increasing thread counts from 384 to 768, the speedup factor is reduced to 1.33. Implementation 4 only reached a speedup of 1.14 in the range [384-768]. The decreasing speedup concurs with the statement that, at some point, the benefits of increasing concurrency are negated by the increase in communication cost added.

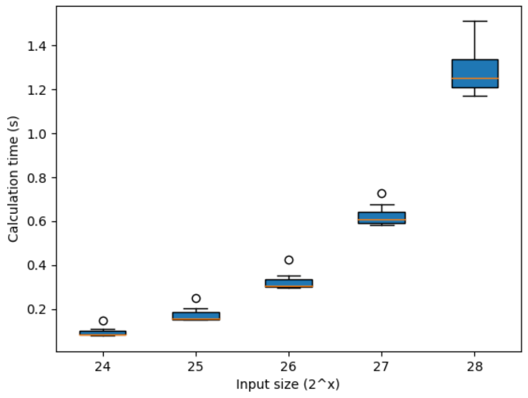


Figure 10: Box plots comparing calculation times per run, per input size on Implementation 3. Runs are taken from DAS5 benchmark experiment 1 were used, see Figure 7.

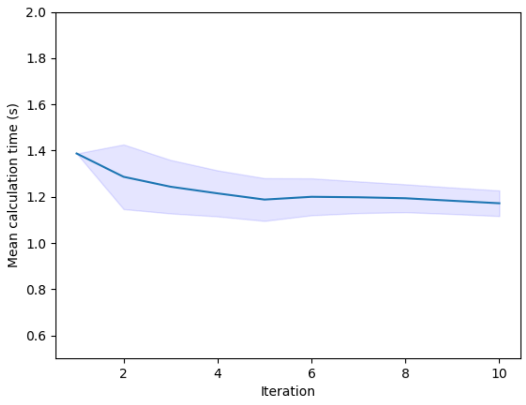


Figure 11: 95% CI plot over the 10 runs on Implementation 4 from DAS5 benchmark experiment 1, see Figure 7. The blue line represents the mean, while the area around it represents the CI.

On both DAS5 and DAS6, relative performance between variants starts to fluctuate between variants with thread counts above 64. The standard deviation between runs is similar for CI plots of different thread counts; around 0.2s to 0.4s. On lower thread counts, this deviation does not have a large impact on performance. For example, using 8 threads, implementation 4 has a calculation time of around 10s. However, when increasing thread counts to 384, the calculation time is reduced to around 0.6s, while having a similar standard deviation between runs; still 0.2s to 0.4s. The same standard deviation has a much higher impact on lower runtimes. As a result, the runtimes on higher thread counts fluctuate, causing unstable performance where different implementations look best on different thread counts.

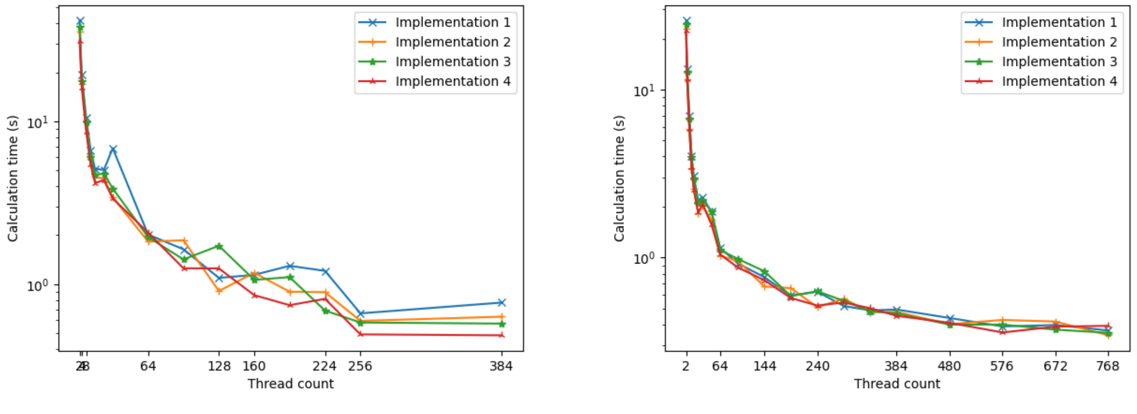


Figure 12: Thread count variation on implementations 1 to 4. Results are averaged over 10 runs. The y-axis is in logarithmic scale. Dataset was of size 28. A convergence delta of 0.0001 was used. Experiment ran on DAS5 (a) and DAS6 (b). Initialization seed is set to 340632450. Initialized using mean set 3.

4 Defining tUPL communication transformations

In prior work, the derivations from Algorithms 1, 2 to Implementations 1 to 4 were based on many assumptions. Most importantly, it is not specified where shared spaces are stored. For example, is `PC[]` stored as a local copy on each process? Or is it read from a global shared space? This problem extends to partitions. As a consequence, the derivation of the communication scheme cannot be specified explicitly. Is the communication scheme strongly, eventually, or even weakly consistent? Currently, it looks like the scheme is supposed to be strongly consistent, updating the global `PM_C[]`, `PM_S[]` atomically after each write. However, the current derivations are not strongly consistent and read data from local copies of shared spaces. Another assumption is the convergence condition. When does a process terminate? This is currently not present in the specifications. As a result of these assumptions, it is unclear how the `MPI C++` implementations are derived from the provided final specifications.

To fill this gap in the derivation process, new syntax and transformations are added to tUPL. The goal of these extensions is to transform a materialized tUPL-specification into a specification that enables the tUPL-compiler to generate communication code. Extending the syntax was required in order to express inter-process communication in tUPL-specifications. New transformations enable tUPL-specifications to be transformed into a specification such that tUPL can generate communication code. The new transformations introduce and operate on the aforementioned syntax extension. In the remainder of this section, three new transformations are described:

- **shared space localization**, which introduces local and global shared space versions,
- **synchronization delay**, which delays the synchronization of intermediate results,
- and **convergence**, which adds a stopping condition.

Additionally, the new transformations are applied to the materialized k-means tUPL-specifications presented in Algorithms 1, 2, resulting in many new implementation variants.

4.1 Shared space localization

Before any communication code can be inserted, it must be clear whether shared spaces are local or global, and if they are a partition of a shared space. **Shared space localization** transforms a shared space into a local and global copy. Changes are made to the local version, while reads are based on the global version. Global and local are synchronized after each write to local. The interpretation of the local and global versions is still flexible. For example, the global version can be a local copy of a globally synced space or be synced in shared memory. Keeping this flexibility provides future transformations with more options. This transformation requires the tUPL-specification to be in materialized form before it can be applied. Shared space localization can be applied to one or more shared spaces. The shared spaces to which shared space localization is applied are considered to be included in the transformation. Depending on read and/or write accesses, the new local copies either replace their previous global version or must be synchronized to global with all involved processes. Accesses to a shared space can be determined through static analysis of the tUPL-specification. Table 2 shows the access types and corresponding set of shared spaces present after the shared space localization transformation is applied. In general, a global copy is only required when a shared space is written to.

	Local Copy	**Global Copy**
Read-only	Yes	No
Write-only	Yes	Yes
Read-write	Yes	Yes

Table 2: Shared space localization access table. Here, for each access type, it is stated whether it needs a local copy, global copy, or both.

Table 2 holds in most cases. However, there is an edge case with partitions. When a shared space is divided into a set of fully disjoint partitions, and each process operates only on their assigned (disjointed) partition, there is no need to include a global copy. In this case, there is no possibility of overlap between reads and/or writes from different processes to the same data. When it is clear the shared space is divided into a set of fully disjoint partitions, a choice can be made to only include a local copy in the tUPL-specification. Not including a global version of a shared space removes the need for synchronization, thus reducing overall communication cost.

A syntactical extension is required to denote whether a shared space is local or global. To this end, the syntax is extended with subscripts for shared spaces. The global version of a shared space receives the '$_g$' subscript, while the local copy receives the '$_l$' subscript. Shared space localization adds a subscript to the shared spaces it is applied to. Table 3 shows the result of applying shared space localization to a shared space A.

	Local Copy	**Global Copy**	**Synchronization**
Read-only	A_l		
Write-only	A_l	A_g	$Sync(A_l, A_g)$
Read-write	A_l	A_g	$Sync(A_l, A_g)$

Table 3: Shared space localization transformation table. Result of applying shared space localization on shared space A for different access types to A.

To illustrate, applying shared space localization to `PM_S` in the assignment `PM_S[PM[`i`]] -= 1` would result in either

```
PM_S$_l$[PM[$i$]] -= 1
Sync(PM_S$_l$, PM_S$_g$)
```

or, if there is no global variant,

```
PM_S$_l$[PM[$i$]] -= 1.
```

Shared space localization prepares the specification for the synchronization delay transformations described in the next subsection. The `sync` function is a placeholder and is invalid for final tUPL-specifications. At least one synchronization delay transformation, which includes the localized shared spaces, is required after applying shared space localization.

After the application of shared space localization, it is assumed that each untreated shared space is global. Each read/write is from/to a globally shared memory or from/to a strongly consistent local copy of a globally synchronized memory. In other words, the default notation denotes atomic operations that are synchronized after each write.

4.2 Synchronization delay

Delaying the synchronization of intermediate results is the next next step in the process. To this end, the **synchronization delay** (sync. delay) transformation was designed. The goal of this transformation is to buffer updates made to the targeted shared spaces. The communication code to achieve this consists of two `forelem` loops, namely a buffering and synchronization loop. In the buffering loop, all local changes made to targeted shared spaces are buffered. Then, in the synchronization loop, the buffered values are reduced to new global values. A mathematical deduction is required to receive the reduction code needed for buffering and synchronization.

Sync. delay can only be applied to shared spaces to which shared space localization has been applied. Recall, a tUPL-specification must be materialized before shared space localization can be applied. Sync. delay consists of three steps:

- **selection**, where one or more shared spaces are selected,
- **communication code deduction**, where the core operations of the tUPL-specification are used to mathematically deduce communication code,
- and **communication code insertion**, where the mathematical communication code is transformed to tUPL and inserted in the specification.

Sync. delay uses symbolic mathematics to deduce communication code. In its current state, the deduction of communication code is manually guided. In Section 5.2.4, a proof-of-concept is provided for automating the deduction.

4.2.1 Selection

In the selection step, one or more shared spaces are selected as targets to buffer updates for. The selection of shared spaces is based on a main target, and may include its write-dependencies as additional targets. The main target determines which shared space must be communicated. The inclusion of additional targets in the communication code is optional. Any shared space with a global copy can be selected as main target. This excludes read-only shared spaces. An example heuristic for main target selection is to select the available shared space with the most dependencies in its write operations. Including more dependencies results in more delayed communication, as more intermediate results are included in the delay. Write-dependencies refer to shared spaces with write-access that are used in assignment operations to the main shared space. For example, in `PM_C[m] = (PM_C[m]*PM_S[m]+PC[i])/(PM_S[m]+1); PM_S[m] += 1`, if we select `PM_C` as main target, `PM_S` is a write-dependency. `PC` is read-only, so this is ignored. `PM_S` could be included in the transformation. Note that a write-dependency shared space can only be included if shared space localization has been applied to it. Not including write-dependencies may produce invalid, or incorrect results. However, this depends on the operations affected by the sync. delay, as well as the nature of the algorithm. With k-means, ignoring the write-dependencies still produces correct results due to the convergence-based nature of the algorithm.

4.2.2 Communication code deduction

The next step is to create communication code. In this step, the 'core' operations of the tUPL-specification are used to deduce communication code. This step produces reduction code that is necessary for creating buffers and synchronization operations; as done in the insertion step described in Section 4.2.3. The deduction starts by exporting the core operations to Sympy [39], a symbolic mathematics package for Python. Sympy offers manipulation of expressions. This includes parsing, simplification, substitution, and solving of different expression types. Extra functionality can be added with regular functions and/or classes. In Sympy, the core operations are transformed into one-sided Sympy expressions.

To derive the communication code, the expressions are rewritten and solved for the targeted shared spaces. The resulting equations are simplified and substituted in other expressions and/or equations. Exhaustively performing these actions creates multiple paths which can lead to different communication code options. In case of multiple options, a choice must be made. A simple heuristic is the communication code expected to perform the least amount of operations. This could be determined through static and/or dynamic analysis. However, when shared space localization is not applied to all required shared spaces, some options become unavailable/invalid. This method results in a reduction equation that equates the main target to an expression. The expression determines how the main target will be reduced/communicated.

This step may alter shared spaces to partitions in order to accommodate for the equation requirements. For example, if the reduction equations consists of operations that only operate on a disjoint set of points for each process, the points would be altered to a local partition. Local shared space partitions are denoted by the '$_p$' subscript, which can be concatenated with the local or global subscripts to create local or global partitions of shared spaces.

4.2.3 Inserting communication code

In this step, the Sympy equation from the deduction step is transformed into a buffering `forelem` loop and a synchronization `forelem` loop. Buffers are created based on the right-hand side of the Sympy equation. For example, if we communicate A, where the equation $A = \frac{N}{D}$ represents the reduction, buffers are created for the operands N and D. Buffers created by sync. delay receive the suffix '`_B`'. When a process writes to a buffer, it writes to its local version of the buffer. When the buffer is read, the buffer is synchronized right before the read. In other words, before every read of the buffer, the buffer is synchronized (globally). Buffers are initialized with zeros, and are reset to zero after each cycle. N and D are expressions, which can take different forms depending on the results from the deduction step. N and D are considered the buffered operations. The buffering loop iterates over all values used to address the buffered operations, filling the buffer as it goes. Then, the synchronization loop uses the filled buffers to perform the reduction equation $A = \frac{N}{D}$, where A is the global version, or a local copy of global, and N and D are the buffers. The synchronization loop iterates over all values used to address the destination of the reduction operation. In other words, the address range of the synchronization loop is equivalent to the length of the buffers.

A first sync. delay inserts the communication code in the same scope as the core loop used to deduce communication. In this scope, the communication code is inserted after its corresponding core loop. The synchronization loop is placed after the buffering loop. Sync. delay can be applied multiple times. Only the first sync. delay performs the selection and deduction steps. Each subsequent application of sync. delay to the same targets moves the communication code one scope up; where the scope of the function containing the communication code serves as an upper limit.

While applying one or more sync. delays, loop blocking can be used at any level to create chunks. These chunks, combined with sync. delay, provide control over communication moments and sizes. Combining loop blocking and sync. delay creates opportunities to optimize the specification for the target cluster computer. For example, loop blocking can break a node's partition into p blocks, where p is the amount of processes used in said node. The p blocks can then communicate within the node, which is faster than communication between nodes. The same holds on CPU-thread level. This form of data control could even be used to add shared-memory parallelism (OpenMP) on node and/or CPU level. However, such hybrid implementations would require additional transformations and identifiers, such as a node-partitioning and CPU/thread partitions.

When applying sync. delay multiple times, it is possible that the communication code would be placed after the core loop of the specification. This produces a specification that only synchronizes after the core loop has already terminated. To avoid this, loop blocking can be applied before escaping the core loop with another sync. delay. For this purpose, loop blocking would encapsulate the core and communication code in a single block ($p = 1$ blocks).

The insertion of the buffering loop can introduce a termination problem when inserted within a `whilelem` loop. The operations of the buffering loop are unguarded. As a consequence, the buffering loop will always perform its operations, which causes the `whilelem` loop to never result in no-ops. Recall that a `whilelem` loop only terminates in case no tuples are left for which the encapsulating if-statement holds true. To allow the specification to terminate, a convergence transformation is required.

4.3 Convergence

To solve the termination problem introduced by sync. delay, communication should only be enabled when changes are made. The **convergence** transformation adds a local 'change' shared space and a change buffer that track changes made to a shared space. Both shared spaces have the same size as the buffers they track. Equal to buffers, the 'change' shared space is initially filled with zeros and reset at the start of each cycle. Whenever a tracked value is changed, a counter is increased in the local change space. A change loop is inserted right before the corresponding buffering loop. The change loop writes local changes to the change buffers. The if-statement'`if CHANGE_B[<index>] > 0`' is added to the buffering loop, only buffering data when changes are made. Recall that buffers are synchronized before they are read. The same condition is applied to the synchronization loop. As a result, only changed indexes are communicated, and the process terminates when no changes are made. This transformation is only required if the application of sync. delay results in a termination problem. In other cases, it is unnecessary.

5 Applying the communication transformations

In this section, the transformations described in Section 4 will be applied to k-means. Additionally, the general applicability of the transformations will be demonstrated by applying them to a tUPL sorting specification.

5.1 Shared Space Localization

In Algorithm 3, shared space localization is applied to the k-means specification from Algorithm 1. The transformation is applied to `PM_C` and `PM_S`. The application of shared space localization results in minor changes. Both `PM_C` and `PM_S` are split into a global and local version. Creating a distinction between a local and global version serves as a preparatory step for synchronization delay.

Algorithm 3 Applying shared space localization to PM_C and PM_S

```
whilelem $(i \in [0, |\text{PM}| - 1])$
  forelem $(m \in [0, k - 1])$
    if $(\text{PM}[i] \mathrel{!}= m \;\&\&\; \text{dist}(\text{PC}[i], \text{PM\_C}_g[m]) < \text{dist}(\text{PC}[i], \text{PM\_C}_g[\text{PM}[i]]))$ {
      $\text{PM\_C}_l[\text{PM}[i]] = (\text{PM\_C}_l[\text{PM}[i]] * \text{PM\_S}_l[\text{PM}[i]] - \text{PC}[i])/(\text{PM\_S}_l[\text{PM}[i]] - 1)$
      $\text{sync}(\text{PM\_C}_l, \text{PM\_C}_g)$
      $\text{PM\_S}_l[\text{PM}[i]] -= 1$
      $\text{sync}(\text{PM\_S}_l, \text{PM\_S}_g)$
      $\text{PM\_C}_l[m] = (\text{PM\_C}_l[m] * \text{PM\_S}_l[m] + \text{PC}[i])/(\text{PM\_S}_l[m] + 1)$
      $\text{sync}(\text{PM\_C}_l, \text{PM\_C}_g)$
      $\text{PM\_S}_l[m] += 1$
      $\text{sync}(\text{PM\_S}_l, \text{PM\_S}_g)$
      $\text{PM}[i] = m$
    }
```

5.2 Synchronization Delay

Before showcasing the result of applying sync. delay to Algorithm 3, the details of each individual step are described. Then, the specification from the application of sync. delay is shown. Additionally, sync. delay will be applied a second time. This subsection ends with a proof-of-concept for the automation of sync. delay.

5.2.1 Selection

First, a main target must be selected from the available shared spaces: `PM_C` and `PM_S`. From these options, `PM_C` has the most complex write-operations. Therefore, `PM_C` is selected as the main target. The only write-dependency `PM_S` is included in the transformation process. The inclusion influences the validity of certain options later in the deduction step.

5.2.2 Deduction

The goal of this step is to rewrite the core operations of k-means in such a way that the targets, PM_C and PM_S, can be recalculated (globally) by reduction. K-means has the following 'core' operations:

$$\begin{aligned}
\text{PM\_C}[\text{PM}[i]] &= \frac{\text{PM\_C}[\text{PM}[i]] * \text{PM\_S}[m] - \text{PC}[i]}{\text{PM\_S}[m] - 1} \\
\text{PM\_S}[\text{PM}[i]] &= \text{PM\_S}[\text{PM}[i]] - 1 \\
\text{PM\_C}[m] &= \frac{\text{PM\_C}[m] * \text{PM\_S}[m] + \text{PC}[i]}{\text{PM\_S}[m] + 1} \\
\text{PM\_S}[m] &= \text{PM\_S}[m] + 1 \\
\text{PM}[i] &= m
\end{aligned}$$

These operations are transported to Python and transformed to Sympy notation. This results in the following equations, denoted as expressions:

```
# Indexed arrays
C = IndexedBase('C') # PM_C
S = IndexedBase('S') # PM_S
P = IndexedBase('P') # PC

# Indices
n, k, i = symbols('n k i', cls=Idx)

# Indexed Equations
i_equations: Dict = {
    1 : - C[n+1] + (C[n] * S[n] - P[n]) / (S[n] - 1),
    2 : - S[n+1] + S[n] - 1,
    3 : - C[n+1] + (C[n] * S[n] + P[n]) / (S[n] + 1),
    4 : - S[n+1] + S[n] + 1
} # Indexed equations
```

Here, each shared space is denoted by an IndexedBase array variable. This array is indexed using an Idx, or index, variable. With these equations, a chain of solve, simplification, and substitution actions is used to deduce communication code. As mentioned in Section 4.2, this process is currently guided manually. The deduction process is based on a mathematical deduction, see Appendix B. The first step is to substitute equation 4 into 3. In Sympy, this is denoted as follows:

```
f = i_equations[4]
x = S[n+1]
res = solve(f, x)[0] # returns a list
pprint(Eq(x, res))

f = i_equations[3].subs(res, x)
x = P[n]
res = solve(f, x)[0]
pprint(Eq(x, res))
```

Here, the function `solve` tries to solve expression `f` for `x`. To output an expression as an equation, `Eq()` is used. The above snippet results in:

```
S[n + 1] = S[n] + 1

C[n + 1]*S[n + 1] = C[n]*S[n] + P[n]
```

In the following, to simplify the process, `C[n]*S[n]` is substituted with `A[n]`. The next step is deducing the sum-notation. This is done through generation. Simply said, we fill in `n` for some values `k`. After filling in `n` for values `k = [start, start+2], start = 0` we receive:

```
[(A[1], A[0] + P[0]), (A[2], A[1] + P[1]), (A[3], A[2] + P[2])]
```

Next, a substitution chain from left to right is created. This results in:

```
[(A[1], A[0] + P[0]), (A[2], A[0] + P[0] + P[1]), (A[3], A[0] + P[0] + P[1] + P[2])]
```

Each expression is then parsed to look for summations. Detected summations are changed to sum-notation. Sympy automatically simplifies expressions if able. Therefore, only advanced simplifications need to be programmed separately. Note that the current code for the Sympy deduction is created specifically for this case. Creating a working 'compiler' that performs all transformations is outside the scope of this thesis. However, it is within scope to prove that it is possible to do so. After parsing, the 'chain' is transformed to:

```
[(A[1], A[0] + P[0]), (A[2], A[0] + Sum(P[k], (k, 0, 1))),
                      (A[3], A[0] + Sum(P[k], (k, 0, 2)))]
```

Now, using the starting index and step size, the symbolic indices are placed back into the expressions. The last expression in the chain is then returned as the transformed expression:

$$\mathtt{A[n + 1] = A[-i + n] +} \sum_{k=-i+n}^{n} \mathtt{P}[k]$$

The result should still be checked with, for example, an inductive proof package[8]. If we state that `A[0]=0` and fill in `n=i=0`, we receive `A[1] = P[0]`, which we can use to substitute `i` with `0`:

$$\mathtt{A[n + 1] =} \sum_{k=-i+n}^{n} \mathtt{P}[k]$$

Then, if we decrease `n` by one and revert the substitution `A[]=C[]*S[]` we get:

$$\mathtt{C[n]*S[n] =} \sum_{k=0}^{n-1} \mathtt{P}[k]$$

We can now use this equation in the original equation 3:

$$\mathtt{C[n+1] =} \frac{P[n]+\sum_{k=0}^{n-1} P[k]}{S[n+1]}$$

`P[n]` can be added into the sum. The last step is to use the recurrent solver (`rsolve()`) from Sympy on equation 4. This results in `S[n+1]=n+1`, which is equivalent to a sum +1 from 0 to `n`. Performing this substitution gives the final result:

$$\mathtt{C[n+1] =} \frac{\sum_{k=0}^{n} P[k]}{n+1}$$

The final result denotes the recalculation scheme from [19]. This variant will be called the summation variant of the recalculation scheme. However, another variant is possible. The summation of points can be substituted by a multiplication using the equation $\mathtt{A[n + 1] =} \sum_{k=-i+n}^{n} \mathtt{P}[k]$, where `A[n+1]=C[n+1]*S[n+1]`. This results in a new multiplication variant of the recalculation scheme.

With `PM_C` as main target, the summation variant is always applicable. This is because there is no write-dependency in the global reduction expression. Only a set of local data points (`PC`) is required. The multiplication variant contains the write-dependency `PM_S` in its reduction expression. Therefore, the multiplication variant is only available if both `PM_C` and `PM_S` are localized and included in the targeted set. Sync. delay may alter included shared spaces in order to accommodate for the buffering operations. For example, the multiplication variant requires not only a set of local points, but also local partitions of means `PM_C` and sizes `PM_S`. This is only possible due to the flexible interpretation of the local and global versions of the shared spaces.

[8] https://github.com/kunal-mandalia/proof-by-induction

The final choice depends on which expression is expected to perform the least amount of operations. This presents an opportunity to optimize communication and computation. For example, with high n, communicating PM_C*PM_S saves computational costs compared to communicating $\sum_{i=0}^{n-1}$PC[i]. Therefore, during the compilation process, the multiplication variant would be chosen.

The derived scheme performs incremental updates to the targeted shared spaces. Each communication cycle, the derived scheme compares the values of the targeted shared spaces from the previous communication cycle to the current values. This way, incremental updates can be applied. This is a more generic scheme and therefore does not require a lot of deduction steps. Applying incremental updates to the targets is sufficient. For k-means, this comes down to a simple fraction $a = \frac{b}{c}$, where $a =$PM_C, $b =$PM_C*PM_S+PC, $c =$PM_S. Substituting $b = a * c$ into the fraction creates PM_C=(PM_C*PM_S)/PM_S, to which the derived scheme can easily be applied. Like the recalculation scheme, the derivation of this equation can be performed by Sympy. A proof for the derived scheme is provided in [40].

5.2.3 Insertion

The last step is to transform the deduced equation to tUPL-notation. Assume the multiplication variant was chosen in the deduction step. Then, the means are communicated through the fraction PM_C $= \frac{N}{D}$. Here, the reduction to PM_C is described as a division of the buffered operations $N =$ PM_C*PM_S and $D = \sum_{i=0}^{n}$PC[i]. A buffer is created for each buffering operation. The buffering forelem loop consists of two operations; one for each buffering operation. In Algorithm 3, the variable $m \in [0, k-1]$ is used to address both PM_C and PM_S. Therefore, the buffering loop also iterates over $m \in [0, k-1]$. Note that, if the summation variant was chosen, the buffering loop would have iterated over all points $i \in [0, |PM|-1]$. The synchronization forelem loop contains the reduction operation; in this case PM_C $= \frac{N}{D}$. Here, a new global value is calculated by dividing the filled buffers N and D. The address range of the sync. loop is equivalent to the length of the buffers.

The first sync. delay inserts the communication code in the same scope as the core loop used to deduce it. Algorithm 4 shows the application of sync. delay to the PM_C and PM_S shared spaces. Here, the communication code is placed after the inner forelem loop. The specification is altered to accommodate for the chosen buffering/communication method. The local versions of PM_C and PM_S are transformed into local partitions, as required by the multiplication variant. However, point reassignment is still based on the global version. In this specification, the global version is updated at the end of each iteration of the whilelem loop. In other words, there is a communication moment after each point reassignment.

To avoid clutter in the specifications, the initialization and reset of the buffers are omitted. Recall that buffers are initially filled with zeros and reset after each communication cycle. The following code fragment shows the buffer initialization of Algorithm 4:

```
forelem ($m \in [0, k-1]$)
    PM_S_B[m] = 0.0
    PM_C_B[m] = 0.0
```

This loop would be placed right before the buffering loop to ensure it is set and reset to zeros before each buffering cycle.

Algorithm 4 Applying synchronization delay to Algorithm 3

```
whilelem ($i \in [0, |\text{PM}| - 1]$) {
  forelem ($m \in [0, k - 1]$)
    if ($\text{PM}[i] \; != m$ && $\text{dist}(\text{PC}[i], \text{PM\_C}_g[m]) < \text{dist}(\text{PC}[i], \text{PM\_C}_g[\text{PM}[i]])$) {
      $\text{PM\_C}_{lp}[\text{PM}[i]] = (\text{PM\_C}_{lp}[\text{PM}[i]] * \text{PM\_S}_{lp}[\text{PM}[i]] - \text{PC}[i])/(\text{PM\_S}_{lp}[\text{PM}[i]] - 1)$
      $\text{PM\_S}_{lp}[\text{PM}[i]] -= 1$
      $\text{PM\_C}_{lp}[m] = (\text{PM\_C}_{lp}[m] * \text{PM\_S}_{lp}[m] + \text{PC}[i])/(\text{PM\_S}_{lp}[m] + 1)$
      $\text{PM\_S}_{lp}[m] += 1$
      $\text{PM}[i] = m$
    }
  forelem ($m \in [0, k - 1]$) { // buffering loop
    $\text{PM\_S\_B}[m] += \text{PM\_S}_{lp}[m]$
    $\text{PM\_C\_B}[m] += \text{PM\_C}_{lp}[m] * \text{PM\_S}_{lp}[m]$
  }
  forelem ($m \in [0, k - 1]$) // sync. loop
    $\text{PM\_C}_g[m] = \text{PM\_C\_B}[m]/\text{PM\_S\_B}[m]$
}
```

Applying a second sync. delay to Algorithm 4 would place the communication code after the outer `whilelem` loop. However, this produces a specification that only synchronizes after the core loop has terminated (see Appendix 15). To avoid this, loop blocking is applied before escaping the core loop with another sync. delay. Algorithm 5 shows the application of loop blocking, followed by a second sync. delay. The amount of blocks, x, is set to 1 to encapsulate the core loop without actually partitioning the shared space. The resulting specification synchronizes after each iteration, instead of after each point. Note that, when $x > 1$, synchronization occurs after each part.

Note that Algorithm 5 uses the summation communication code instead of the multiplication code from Algorithm 4. This is a choice. The multiplication code is also applicable. The summation code does not require partitions, as local changes are applied to the local copy of global means and sizes. Each communication moment, the global means and sizes are completely recalculated (globally) and saved in the local copies. In Algorithm 4, all point reassignments are based on the global mean values calculated at the previous communication moment. However, in the multiplication variant from Algorithm 5, the local copy of `PM_C` is updated with local point reassignments. As a result, the point reassignments between the two versions are slightly different. Not updating the global means during an iteration affects point reassignment and may alter the convergence rate in terms of iterations required. This difference becomes more negligible the more processes are involved. With more processes, each process receives a smaller part of the points. As a result, the local copy of global means is updated using a smaller part of the points. In practice, both versions produce correct results. Implementations 1-4 all update a local copy of global means and sizes during point reassignment.

5.2.4 Automating Synchronization Delay

In Section 5.2.2, it is shown that it is possible to use Sympy to deduce the communication operations from the core set of operations of a tUPL-specification. Automating the application of expression transformations in the deduction step is the final step in fully automating the synchronization delay transformation. Because there are many possible deductions, this can be implemented using a tree-like mapping structure that contains expressions as nodes and transformations as branches. The root nodes would be the original indexed expressions. Each original expression in the set has a separate root node. For each node, its expression is solved for all potential components, preferably without expanding the expression. This results in equations that can be used as substitutions across nodes. Other expression transformations, such as rewrite to sum-notation, can also be applied to nodes. When a transformation is applied to a node, a new

Algorithm 5 Applying loop blocking and two synchronization delays to Algorithm 3

```
whilelem ($ii \in [0, (|\text{PM}|/x) - 1]$) {
  forelem ($i \in [ii * x, (ii + 1) * x - 1]$)
    forelem ($m \in [0, k - 1]$)
      if ($\text{PM}[i]\ != m\ \&\&\ \text{dist}(\text{PC}[i], \text{PM\_C}_l[m]) < \text{dist}(\text{PC}[i], \text{PM\_C}_l[\text{PM}[i]])$) {
        $\text{PM\_C}_l[\text{PM}[i]] = (\text{PM\_C}_l[\text{PM}[i]] * \text{PM\_S}_l[\text{PM}[i]] - \text{PC}[i])/(\text{PM\_S}_l[\text{PM}[i]] - 1)$
        $\text{PM\_S}_l[\text{PM}[i]] -= 1$
        $\text{PM\_C}_l[m] = (\text{PM\_C}_l[m] * \text{PM\_S}_l[m] + \text{PC}[i])/(\text{PM\_S}_l[m] + 1)$
        $\text{PM\_S}_l[m] += 1$
        $\text{PM}[i] = m$
      }
  forelem ($ii \in [0, (|\text{PM}|/x) - 1]$)
    forelem ($i \in [ii * x, (ii + 1) * x - 1]$) {
      $\text{PM\_S\_B}[\text{PM}[i]] += 1$
      $\text{PM\_C\_B}[\text{PM}[i]] += \text{PC}[i]$
    }
  forelem ($m \in [0, k - 1]$)
    $\text{PM\_C}_l[m] = \text{PM\_C\_B}[m]/\text{PM\_S\_B}[m]$
}
```

node is created and the transformation from source to destination is saved. To avoid duplication, multiple parent nodes can lead to the same child node. This could be implemented using a map, or dictionary of the form `Dict[Node, Dict[Tuple[expr, expr], Node]]`, where `expr, expr` represents the transformation from (sub)expression to (sub)expression. To help guide this process, rules and conditions can be added. Most notably, which shared spaces are communicated? And, what type of shared spaces are in the expression (local/global/partitions)? By imposing rules and conditions, candidate nodes can more easily be selected. Each node that satisfies the imposed rules and conditions is considered a candidate. After the tree has been generated, the best-fitting candidate is selected. Here, the 'best-fit' depends on either a heuristic or the best performance achieved on the setup in use.

A template of the proposed Node and Tree classes is as follows:

```
class Node:
    expression: any                 # Node expression
    components: List[any]           # list of (sub)expressions from expression
    equations: List[Tuple[any, any]] # list of equations from solving
                                    # expression for each component

class Tree:
    conditions: List[any]                         # list of conditions
    nodes: List[Node]                             # all Nodes in the tree
    lineage: Dict[Node, List[Node]]               # track lineage of nodes
    tree: Dict[Node, Dict[Tuple[any, any], Node]] # {src: {(expr, expr): dest}}
```

Here, each Node contains a Sympy expression, a list of (sub)expressions obtainable from the main expression, and a list of equations that solve the main expression for each component in the components list. The Tree contains a list of nodes, initialized with the core expressions from a tUPL-specification. The tree can be extended by applying transformations to its nodes. For example, a new node can be created by substituting equations from one node into the expression of another node. The tree then maps the source node to the new node through the performed substitution. This substitution is denoted as a tuple of source and substitution, both denoted as an expression. The 'lineage' of a node can be saved by mapping a node to its predecessor nodes. A node can have more than one predecessor, as multiple paths may lead to it. Exhaustively performing all available transformations will generate new nodes and paths that can contain eligible candidate expressions. This structure can be used to reproduce the substitutions from Section 5.2.2.

The Tree and Node classes should use Sympy solvers and transformators in order to apply substitutions and other transformations to expressions. Sympy contains many solvers, but some special functions need to be programmed manually. Note that the pool of available solve and transformation methods can easily be extended. New methods can be added when ready. Adding a new method potentially results in a fundamentally different tree, and thus may lead to different end candidates.

5.3 Convergence

For k-means, the application of convergence after sync. delay is required in order to obtain a terminating tUPL-specification. This is because the buffering loop is placed within the `whilelem` core loop. In Algorithm 6, the convergence transformation is applied to Algorithm 5. This algorithm presents the final specification that can be used to derive implementation 1. Changes are counted on both ends; both cluster `PM[`i`]` and cluster m. Only tracking one may lead to incorrect results. For instance, if only `PM[`i`]` is tracked, updates to a specific m are lost if $m \neq$`PM[`i`]` holds an entire cycle for that m.

Algorithm 6 Applying the convergence transformation to Algorithm 5

```
whilelem ($ii \in [0, (|\text{PM}|/x) - 1]$) {
  forelem ($i \in [ii * x, (ii + 1) * x - 1]$)
    forelem ($m \in [0, k - 1]$)
      if ($\text{PM}[i] \mathrel{!}= m$ && $\text{dist}(\text{PC}[i], \text{PM\_C}_l[m]) < \text{dist}(\text{PC}[i], \text{PM\_C}_l[\text{PM}[i]])$) {
        $\text{CHANGE}_l[\text{PM}[i]] += 1$
        $\text{CHANGE}_l[m] += 1$
        $\text{PM\_C}_l[\text{PM}[i]] = (\text{PM\_C}_l[\text{PM}[i]] * \text{PM\_S}_l[\text{PM}[i]] - \text{PC}[i]) / (\text{PM\_S}_l[\text{PM}[i]] - 1)$
        $\text{PM\_S}_l[\text{PM}[i]] -= 1$
        $\text{PM\_C}_l[m] = (\text{PM\_C}_l[m] * \text{PM\_S}_l[m] + \text{PC}[i]) / (\text{PM\_S}_l[m] + 1)$
        $\text{PM\_S}_l[m] += 1$
        $\text{PM}[i] = m$
      }
  forelem ($m \in [0, k - 1]$)
    if ($\text{CHANGE}_l[m] > 0$)
      $\text{CHANGE\_B}[m] += \text{CHANGE}_l[m]$
  forelem ($ii \in [0, (|\text{PM}|/x) - 1]$)
    forelem ($i \in [ii * x, (ii + 1) * x - 1]$)
      if ($\text{CHANGE\_B}[\text{PM}[i]] > 0$) {
        $\text{PM\_S\_B}[\text{PM}[i]] += 1$
        $\text{PM\_C\_B}[\text{PM}[i]] += \text{PC}[i]$
      }
  forelem ($m \in [0, k - 1]$)
    if ($\text{CHANGE\_B}[m] > 0$)
      $\text{PM\_C}_l[m] = \text{PM\_C\_B}[m] / \text{PM\_S\_B}[m]$
}
```

Communicating changes also prevents early process termination. If only local changes are considered, a process will terminate directly when no local changes are made. As global changes affect the local state, this is undesirable. Alternatively, it is possible to count global changes and add a threshold. This would be done by guarding the whole communication loops instead of their operations. Selecting which stopping condition is used is not set; it can be changed later in the compilation process.

5.4 Updating implementation specifications

Algorithm 6 shows the updated specification that can be used to derive implementation 1, which uses the recalculation scheme with the summation variant. To obtain a specification for the derived scheme, some minor changes are needed. The derived scheme is based on incrementally updating the target shared spaces. Again, PM_C is set as target, and its dependency PM_S is included. Each process writes to its own copies of PM_C and PM_S. These are full local copies. The values are updated by applying incremental updates based on the changes made by each process. As mentioned in Section 5.2.2, updating to new global values can be done with the equation PM_C=(PM_C*PM_S)/PM_S. This comes down to PM_C= $\frac{N}{D}$, where $N =$(PM_C*PM_S*p)-((p-1)*PM_C$_{-1}$*PM_S$_{-1}$) and $D =$(PM_S*p)-((p-1)*PM_S$_{-1}$. Here, each process p adds its (altered) copy of PM_C and/or PM_S and subtracts $p - 1$ times the previous version of this calculation. A proof of this method is shown in [40]. Apart from the update operations, the only change needed is the addition of a shared space that saves the previous values of PM_C and PM_S. This is added in the insertion step. Algorithm 7 shows the final specification used to derive implementation 2. Here, PM_C_OLD is used to store the previous value of the updated PM_C. Unlike the recalculation scheme, the update operations for the derived scheme do not have to be fully deduced. Obtaining an equation containing all targets is enough. With the recalculation scheme, the recalculation of communicated shared spaces had to be deduced. Recalculation depends on the operations that are recalculated. The derived scheme update is more generic, as it uses the changes made by each process to update. When the derived scheme is chosen, sync. delay tries to produce an update equation to which iterative updates can be applied. This can be any equation that contains only the targets as variables. It then also adds shared spaces that store the previous version of the updated shared spaces.

Applying the new transformations to Algorithm 2, where shared space values are localized in tuples, is identical compared to Algorithm 1. The changes from Algorithm 2 carry over to the updated specifications. As a result, updated specifications for implementation 3 and 4 are created by repeating the transformation processes applied to implementations 1 and 2 on Algorithm 2, respectively.

Algorithm 7 Implementation 2 final tUPL-specification

```
**whilelem** $(ii \in [0, (|\text{PM}|/x) - 1])$ {
  **forelem** $(i \in [ii * x, (ii + 1) * x - 1])$
    **forelem** $(m \in [0, k - 1])$
      **if** $(\text{PM}[i] \; != m \;\&\&\; \text{dist}(\text{PC}[i], \text{PM\_C}_l[m]) < \text{dist}(\text{PC}[i], \text{PM\_C}_l[\text{PM}[i]]))$ {
        $\text{CHANGE}_l[\text{PM}[i]] += 1$
        $\text{CHANGE}_l[m] += 1$
        $\text{PM\_C}_l[\text{PM}[i]] = (\text{PM\_C}_l[\text{PM}[i]] * \text{PM\_S}_l[\text{PM}[i]] - \text{PC}[i])/(\text{PM\_S}_l[\text{PM}[i]] - 1)$
        $\text{PM\_S}_l[\text{PM}[i]] -= 1$
        $\text{PM\_C}_l[m] = (\text{PM\_C}_l[m] * \text{PM\_S}_l[m] + \text{PC}[i])/(\text{PM\_S}_l[m] + 1)$
        $\text{PM\_S}_l[m] += 1$
        $\text{PM}[i] = m$
      }
  **forelem** $(m \in [0, k - 1])$
    **if** $(\text{CHANGE}_l[m] > 0)$
      $\text{CHANGE\_B}[m] += \text{CHANGE}_l[m]$
  **forelem** $(m \in [0, k - 1])$
    **if** $(\text{CHANGE\_B}[m] > 0)$ {
      $\text{PM\_S\_B}[m] += \text{PM\_S}_l[m]$
      $\text{PM\_C\_B}[m] += \text{PM\_C}_l[m] * \text{PM\_S}_l[m]$
    }
  **forelem** $(m \in [0, k - 1])$
    **if** $(\text{CHANGE\_B}[m] > 0)$ {
      $\text{PM\_C}_l[m] = \text{PM\_C\_B}[m] - ((p - 1) * \text{PM\_C\_OLD}[m])$
      $\text{PM\_S}_l[m] = \text{PM\_S\_B}[m] - ((p - 1) * \text{PM\_S\_OLD}[m])$
      $\text{PM\_C\_OLD}[m] = \text{PM\_C}_l[m]$
      $\text{PM\_S\_OLD}[m] = \text{PM\_S}_l[m]$
    }
  **forelem** $(m \in [0, k - 1])$
    **if** $(\text{CHANGE\_B}[m] > 0)$
      $\text{PM\_C}_l[m] = \text{PM\_C}_l[m]/\text{PM\_S}_l[m]$
}
```

5.5 New k-means variants

With the options provided by the sync. delay transformation, 6 new implementations emerged; 3 for both Algorithm 1 and Algorithm 2. All are based on the recalculation scheme, but differ from Algorithm 6.

Implementation 1 uses the summation recalculation option. However, as mentioned in Section 4.2, the means can also be recalculated using multiplication. This version is shown in Algorithm 4. Continuing the transformation process with multiplication recalculation results in Algorithm 8, which will be referred to as Implementation 5. This version tracks local changes made to its local partitions of `PM_C` and `PM_S`. As mentioned in Section 4.2.3, changing to local partitions affects point reassignment. Aside from this, the only practical difference is the buffering method. The summation method performs $i * d$ additions and i increments to fill `PM_C_B` and `PM_S_B`, respectively; where i is the amount of local data points, and d the dimension of points. The multiplication method only performs $m * d$ multiplications; where m refers to the amount of means. With k-means, in general, the amount of points (i) is magnitudes higher than the amount of clusters (m). Therefore, the multiplication variant is expected to perform significantly faster during communication. More specifically, in preparing the buffers for communication. Applying the exact same process from Algorithm 8 to Algorithm 2 results in Implementation 6.

Algorithm 8 Implementation 5 tUPL-specification

```
whilelem $(ii \in [0, (|\text{PM}|/x) - 1])$ {
  forelem $(i \in [ii * x, (ii + 1) * x - 1])$
    forelem $(m \in [0, k - 1])$
      if $(\text{PM}[i] \mathrel{!}= m \;\&\&\; \text{dist}(\text{PC}[i], \text{PM\_C}_g[m]) < \text{dist}(\text{PC}[i], \text{PM\_C}_g[\text{PM}[i]]))$ {
        $\text{CHANGE}_l[\text{PM}[i]] += 1$
        $\text{CHANGE}_l[m] += 1$
        $\text{PM\_C}_{lp}[\text{PM}[i]] = (\text{PM\_C}_{lp}[\text{PM}[i]] * \text{PM\_S}_{lp}[\text{PM}[i]] - \text{PC}[i])/(\text{PM\_S}_{lp}[\text{PM}[i]] - 1)$
        $\text{PM\_S}_{lp}[\text{PM}[i]] -= 1$
        $\text{PM\_C}_{lp}[m] = (\text{PM\_C}_{lp}[m] * \text{PM\_S}_{lp}[m] + \text{PC}[i])/(\text{PM\_S}_{lp}[m] + 1)$
        $\text{PM\_S}_{lp}[m] += 1$
        $\text{PM}[i] = m$
      }
  forelem $(m \in [0, k - 1])$
    if $(\text{CHANGE}_l[m] > 0)$
      $\text{CHANGE\_B}[m] += \text{CHANGE}_l[m]$
  forelem $(m \in [0, k - 1])$
    if $(\text{CHANGE\_B}[m] > 0)$ {
      $\text{PM\_S\_B}[m] += \text{PM\_S}_{lp}[m]$
      $\text{PM\_C\_B}[m] += \text{PM\_C}_{lp}[m] * \text{PM\_S}_{lp}[m]$
      $\text{PM\_C}_g[m] = \text{PM\_C\_B}[m]/\text{PM\_S\_B}[m]$
    }
}
```

With full recalculation (summation), updates made to local copies of means and sizes during each iteration are overwritten at communication. In other words, the local updates made to $\texttt{PM\_C}_l$ and $\texttt{PM\_S}_l$ in the core loop are overwritten by the reduction in the synchronization loop. Simply stated, tracking local changes with full recalculation is unnecessary. Removing the updates made to local copies saves computations. This presents an optimization for the sync. delay transformation. Algorithm 9 shows Implementation 7, where the updates to the local copies of `PM_C` and `PM_S` are removed. Note that changes still need to be tracked. Otherwise, the algorithm would not converge. Implementation 8 is created by applying the same process to Algorithm 2.

Algorithm 9 Implementation 7 tUPL-specification

```
whilelem ($ii \in [0, (|\text{PM}|/x) - 1]$) {
  forelem ($i \in [ii * x, (ii + 1) * x - 1]$)
    forelem ($m \in [0, k - 1]$)
      if ($\text{PM}[i]$ != $m$ && dist($\text{PC}[i], \text{PM\_C}_l[m]$) < dist($\text{PC}[i], \text{PM\_C}_l[\text{PM}[i]]$)) {
        $\text{CHANGE}_l[\text{PM}[i]] += 1$
        $\text{CHANGE}_l[m] += 1$
        $\text{PM}[i] = m$
      }
  forelem ($m \in [0, k - 1]$)
    if ($\text{CHANGE}_l[m] > 0$)
      $\text{CHANGE\_B}[m] += \text{CHANGE}_l[m]$
  forelem ($ii \in [0, (|\text{PM}|/x) - 1]$)
    forelem ($i \in [ii * x, (ii + 1) * x - 1]$)
      if ($\text{CHANGE\_B}[\text{PM}[i]] > 0$) {
        $\text{PM\_S\_B}[\text{PM}[i]] += 1$
        $\text{PM\_C\_B}[\text{PM}[i]] += \text{PC}[i]$
      }
  forelem ($m \in [0, k - 1]$)
    if ($\text{CHANGE\_B}[m] > 0$)
      $\text{PM\_C}_l[m] = \text{PM\_C\_B}[m]/\text{PM\_S\_B}[m]$
}
```

Implementations 1-8 all perform sync. delay on both the means (`PM_C`) and mean sizes (`PM_S`). However, it is also possible to ignore the `PM_S` write-dependency from `PM_C`. If shared space localization and sync. delay are only applied to `PM_C`, we receive a very different implementation. Here, `PM_S` is not a target and remains strongly consistent. As a result, the communication method needs to be adjusted. Only `PM_C` is buffered and communicated. This corresponds to Equation 19 from Appendix B. Applying this equation during sync. delay results in Algorithm 10. This implementation will be referred to as Implementation 9. Note that Equation 19 requires each process to operate on a local partition of `PM_C`. This partition is initialized by summing all local data point coordinates in their corresponding mean. Implementation 10 is the result of applying the same process to Algorithm 2.

Ignoring dependencies may still produce incorrect results depending on the underlying algorithm. K-means, a clustering algorithm, is not very strict, or precise, in results. As long as the points end up in the correct cluster, the results are deemed correct. Still, it is worthwhile to try different implementation variants in order to optimize performance for a target platform.

Algorithm 10 Implementation 9 tUPL-specification

```
whilelem ($i \in [0, |\text{PM}| - 1]$) {
  forelem ($m \in [0, k-1]$)
    if ($\text{PM}[i] \mathrel{!}= m$ && $\text{dist}(\text{PC}[i], \text{PM\_C}_g[m]) < \text{dist}(\text{PC}[i], \text{PM\_C}_g[\text{PM}[i]])$) {
      $\text{PM\_C}_{lp}[\text{PM}[i]] -= \text{PC}[i]$
      $\text{PM\_S}[\text{PM}[i]] -= 1$
      $\text{PM\_C}_{lp}[m] += \text{PC}[i]$
      $\text{PM\_S}[m] += 1$
      $\text{PM}[i] = m$
    }
  forelem ($m \in [0, k-1]$)
    $\text{PM\_C\_B}[m] += \text{PM\_C}_{lp}[m]$
  forelem ($m \in [0, k-1]$)
    $\text{PM\_C}_g[m] = \text{PM\_C\_B}[m]/\text{PM\_S}[m]$
}
```

All implementations perform two sync. delays. While this is generally a good idea, it is also possible to apply the convergence transformation after one sync. delay. Algorithm 11 shows the result of applying convergence after one sync. delay. Application of convergence is equivalent, but now synchronization happens after each data point. The same could be done to the summation variant. Omitting the second sync. delay results in significantly more communication moments, resulting in higher overall calculation times. However, the purpose of this variant is to demonstrate the flexibility of the transformations. Adding new transformation options could lead to new transformation permutations, resulting in implementations with different performance characteristics.

Algorithm 11 Applying the convergence transformation to Algorithm 4

```
whilelem ($i \in [0, |\text{PM}| - 1]$) {
  forelem ($m \in [0, k-1]$)
    if ($\text{PM}[i] \mathrel{!}= m$ && $\text{dist}(\text{PC}[i], \text{PM\_C}_g[m]) < \text{dist}(\text{PC}[i], \text{PM\_C}_g[\text{PM}[i]])$) {
      $\text{CHANGE}_{lp}[\text{PM}[i]] += 1$
      $\text{CHANGE}_{lp}[m] += 1$
      $\text{PM\_C}_{lp}[\text{PM}[i]] = (\text{PM\_C}_{lp}[\text{PM}[i]] * \text{PM\_S}_l[\text{PM}[i]] - \text{PC}[i])/(\text{PM\_S}_{lp}[\text{PM}[i]] - 1)$
      $\text{PM\_S}_{lp}[\text{PM}[i]] -= 1$
      $\text{PM\_C}_{lp}[m] = (\text{PM\_C}_{lp}[m] * \text{PM\_S}_{lp}[m] + \text{PC}[i])/(\text{PM\_S}_{lp}[m] + 1)$
      $\text{PM\_S}_{lp}[m] += 1$
      $\text{PM}[i] = m$
    }
  forelem ($m \in [0, k-1]$)
    if ($\text{CHANGE}_l[m] > 0$)
      $\text{CHANGE\_B}[m] += \text{CHANGE}_l[m]$
  forelem ($m \in [0, k-1]$)
    if ($\text{CHANGE\_B}[m] > 0$) {
      $\text{PM\_S\_B}[m] += \text{PM\_S}_{lp}[m]$
      $\text{PM\_C\_B}[m] += \text{PM\_C}_{lp}[m] * \text{PM\_S}_{lp}[m]$
      $\text{PM\_C}_g[m] = \text{PM\_C\_B}[m]/\text{PM\_S\_B}[m]$
    }
}
```

5.6 Parallel sorting in tUPL

To demonstrate the general applicability of the proposed transformations, we discuss how they can be applied to another problem, namely sorting. Sorting is a core problem in computer science. It is a fundamental part of many different algorithms. A plethora of sequential and parallel sorting implementations exists. Especially in parallel implementations, algorithm performance depends on hardware architecture [41]–[43]. Changing hardware architectures results in relative performance differences between the same set of sorting algorithms [43]. Each algorithm has distinct communication and computation characteristics; interacting differently with various architectures. For example, inter-process communication is different depending on the data partitioning method. Take parallel quicksort and bitonic sort. With parallel quicksort, the input data is partitioned into p parts using $p-1$ pivots, where the p parts are merged after being locally sorted. Here, the partitions and the merge operations need to be communicated. With bitonic sort, input data of length n is divided into bitonic sequences of size m, where each bitonic sequence is a concatenation of an ascending and a descending sequence of numbers [43]. Then, in multiple stages, the bitonic sequences are merged into bitonic sequences of size $2m, 4m, 8m, ..., n$, where each step requires communication for swapping elements. Parallel quicksort and bitonic sort differ in both communication size and the amount of communication moments, resulting in different communication characteristics. With different communication and computation characteristics, generating the best implementation for a specific hardware configuration is difficult. This presents a case for tUPL. An abstract, basic tUPL sorting specification can be transformed into different sorting algorithms [44]. The remainder of this section describes how the proposed transformations can be applied to tUPL sorting algorithms.

Algorithm 12 shows the basic tUPL sorting specification. Here, T is the tuple reservoir containing tuples $\langle i, j \rangle$ where $i < j$, and X the shared space containing the keys (referenced by $F_X[i]$ or $F_X[j]$).

Algorithm 12 Basic sorting tUPL specification

```
T = $\{\langle i, j \rangle | 0 \leq i < j < |X|\}$
**whilelem** ($t \in T$)
  **if** ($X[t.i] > X[t.j]$)
    swap($X[t.i], X[t.j]$);
```

In [44], this basic tUPL sorting specification is transformed into a divide-and-conquer sorting specification that first partitions the input and then sorts the partitions. Extending this partitioning transformation with more partitioning variants allows the derivation of multiple sorting algorithms and variants; each having different communication and computation characteristics. Subsequently, derivations will deliver different performance depending on the hardware architecture.

Sample sort [45] is a well-known divide-and-conquer sorting algorithm. It is similar to quicksort. However, instead of partitioning with a single pivot, sample sort partitions using a set of different samples/pivots. This results in multiple buckets. These buckets can be distributed among processes. The processes then sort their assigned buckets in parallel using some other sorting algorithm. The end result is received by concatenating the buckets in order. The partitioning transformation from [44] needs to be extended with bucketing partitioning before the basic sorting specification can be transformed into a sample sort specification. Algorithm 13 shows the application of a concept bucketing partitioning transformation to the basic tUPL sorting specification, resulting in tUPL pseudocode for sample sort. Note that the sorting is still in unmaterialized form. This is to show that different sorting algorithms can still be derived for the sorting parts. Applying other partitioning transformations will result in different sorting algorithms.

The communication transformations proposed in this thesis can transform Algorithm 13 to a specification that expresses inter-process communication. Algorithm 14 demonstrates the result of applying shared space localization and sync. delay to Algorithm 13. Here, convergence is unnecessary as the buffering and synchronization loops are not within a `whilelem` loop. Shared space localization has been applied to all shared spaces, as can be seen by the 'l' and 'g' subscripts. Sync. delay has been applied to both the sampling

($Samples$) and bucketing ($Buckets$) shared spaces. With $Samples$, sync. delay creates a buffering loop that gathers all samples. The synchronization loop is simplified to an assignment, as the samples from all processes are collected. For $Buckets$, sync. delay results in a buffering loop that buffers all buckets and a synchronization loop that concatenates all sorted buckets. In Algorithm 14, the partitioning extracts k samples per thread $rank$, with p total threads. The $p * k$ total samples are gathered globally and sorted afterwards. Local keys are placed in buckets depending on splitters selected from the global samples. Using a bucket buffer, each $rank$ receives a partition containing all keys from each bucket $rank$. Finally, the end-result is received by concatenating the sorted partitions. The sorting of partitions is again in unmaterialized form. As is the case with k-means, the `C++` implementation of a global shared space can either be a local synchronized copy or shared memory. This particular partitioning method imposes a strict ordering of buckets to threads. If this order is ignored, the buckets could be concatenated in the wrong order. Therefore, this method imposes an exact order before concretization. Note that this order only applies to the bucket-concatenation loop. The unmaterialized sorting parts offer another optimization opportunity. Each sorting part can be transformed into a different sorting algorithm. This allows tUPL to optimize the computation.

Algorithm 13 Sample sort tUPL pseudocode

```
// Extract $p * k$ samples, with $p$ the amount of splitters, $k$ the oversampling factor
forelem ($i \in [0, 1, ..., (p-1)k]$)
    $Samples[i] = sample(A)$ // randomly sample a key

// Sort set of samples
whilelem ($t \in T = \{\langle i, j\rangle | 0 \leq i < j < |Samples|\}$)
    if ($Samples[i] > Samples[j]$)
        $swap(Samples[i], Samples[j])$

// Select splitters from sorted samples
forelem ($i \in [0, 1, ..., p]$)
    $Splitters[i] = Samples[i * k]$ // take every $k$'th sample

// Add keys from data points to correct buckets
forelem ($i \in [0, 1, ..., |A|]$)
    forelem ($j \in [0, 1, ..., |Splitters|]$)
        if ($A[i] > Splitters[j-1] \&\& A[i] \leq Splitters[j]$)
            $Buckets[j].append(A[i])$ // add key $i$ to bucket $j$

// Sort buckets (sequentially or in parallel)
forelem ($x \in [0, 1, ..., |Buckets|]$)
    $Partition = Buckets[x]$ // sort bucket $x$
    whilelem ($t \in T = \{\langle i, j\rangle | 0 \leq i < j < |Partition|\}$)
        if ($Partition[i] > Partition[j]$)
            $swap(Partition[i], Partition[j])$

// Concatenate buckets
forelem ($i \in [0, 1, ..., |Buckets|]$)
    $Result.append(Buckets[i])$
```

Algorithm 14 Sample sort tUPL pseudocode with communication transformations applied.

```
// Extract $k$ local samples
**forelem** ($i \in [0, 1, ..., k]$)
    $Samples_l[i] = sample(A_l)$ // randomly sample a local key

// Gather all local samples (inserted by sync. delay)
**forelem** ($i \in [0, 1, ..., |Samples_l|)$
    $Samples\_B.append(Samples_l[i])$
$Samples_g = Samples\_B$

// Sort the global set of samples
**whilelem** ($t \in T = \{\langle i, j\rangle | 0 \leq i < j < |Samples_g|\}$)
    **if** ($Samples_g[i] > Samples_g[j]$)
        $swap(Samples_g[i], Samples_g[j])$

// Select local splitters from sorted global samples
**forelem** ($i \in [k, 2k, ..., (p-1)k]$)
    $Splitters_l[i/k] = Samples_g[i]$ // take every $k$'th sample

// Add keys from local data points to correct buckets
**forelem** ($i \in [0, 1, ..., |A_l|]$)
    **forelem** ($j \in [0, 1, ..., |Splitters_l|]$)
        **if** ($A_l[i] > Splitters_l[j-1] \&\& A_l[i] \leq Splitters[j]$)
            $Buckets_l[j].append(A_l[i])$ // add key $i$ to bucket $j$

// Fill bucket buffers and communicate local partitions (inserted by sync. delay)
**forelem** ($j \in [0, 1, ..., |Buckets_l|]$)
    $Buckets\_B[j].append(Buckets_l[j])$ // add local buckets to buffer
$Partition_l = Buckets\_B[rank]$

// Sort local partition
**whilelem** ($t \in T = \{\langle i, j\rangle | 0 \leq i < j < |Partition_l|\}$)
    **if** ($Partition_l[i] > Partition_l[j]$)
        $swap(Partition_l[i], Partition_l[j])$

// Communicate results
$Buckets\_B[rank] = Partition_l$
**if** ($rank == 0$)
    **forelem** ($i \in [0, 1, ..., |Buckets\_B|]$)
        $Result.append(Buckets_B[i])$
```

6 Experiments

In this section, the performance of the new k-means variants is compared to the variants presented in [19]. This comparison was performed on two different hardware architectures to compare the communication vs computation trade-off. The parameter configuration of the experimental setup of Section 3 was used. Experiments were run on DAS5 and/or DAS6; see Table 1 for the hardware specifications. To reiterate, all datasets were generated using seed 971 with points generated in 4 dimensions and 4 clusters. Dataset size x refers to a dataset with 2^x points. Each experiment run was initialized with seed '340632450' and mean set 3 or 7, see Appendix A. The convergence delta was set to 0.0001. Presented results are averaged over 10 runs. Table 4 shows a summary of all implementation variants.

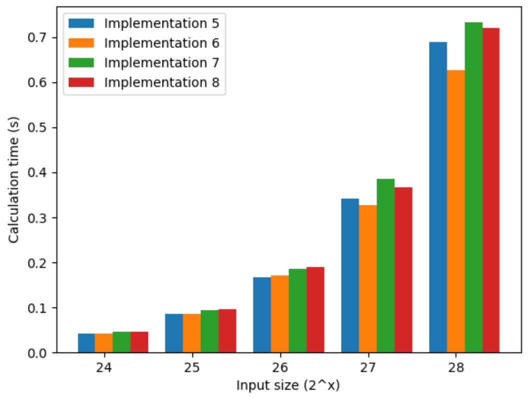
Impl.	Comm. Scheme	Localization	Buffering Method	Program Name	Ref
1	Recalculation	no	Summation	own	6
2	Derived	no	Derived	own_inc	7
3	Recalculation	yes	Summation	own_loc	-
4	Derived	yes	Derived	own_inc_loc	-
5	Recalculation	no	Multiplication	own_m	8
6	Recalculation	yes	Multiplication	own_m_loc	-
7	Recalculation	no	Summation	own_im	9
8	Recalculation	yes	Summation	own_im_loc	-
9	Recalculation	no	Multiplication (only PM_C)	own_values_only	10
10	Recalculation	yes	Multiplication (only PM_C)	own_values_only_loc	-

Table 4: Implementation reference table containing available implementations and their key characteristics. For localized implementations, refer to non-localized variants and Algorithm 2 algorithm reference.

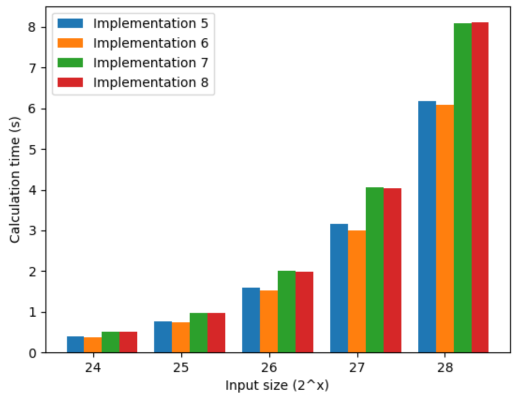
6.1 Performance of new variants

In a first experiment, the performance of the newly derived implementations 5 to 8 is evaluated. The input size range was set to [24-28]. Each run utilized 8 nodes with 8 threads each. This experiment was run using mean set 3 or 7, see Appendix A. Iterations to convergence were identical to the experiment 1 baseline from Section 3; 5 iterations for mean set 3 and 68-71 iterations for mean set 7. Consequently, the runtimes for mean set 7 runs are significantly longer compared to mean set 3 runs. The results for DAS6 are shown on Figure 13. DAS5 results are similar; see Figure 14. Confidence interval plots showed that performance was consistent on both clusters, see Appendix C. Implementation 5 and 6 achieve better performance compared to implementations 7 and 8. The localized implementations 6 and 8 achieve better performance compared to their non-localized implementations 5 and 7. However, performance between the localized and non-localized implementations is similar on DAS5 with mean set 7. Localization has less effect when running with mean set 7 compared to mean set 3. As also observed in Section 3, the effect of localization is less pronounced on DAS5 compared to DAS6; resulting in a minor relative performance difference between DAS5 and DAS6. DAS5 has 2x20MB cache per node, while DAS6 has 128MB per node. Localization allows tuples to contain more data, benefiting hardware architectures with higher cache.

Implementations 5 to 8 all use the recalculation scheme. However, implementations 5 and 6 both utilize the multiplication buffering method enabled by the use of local partitions of `PM_C` and `PM_S`. On the other hand, implementations 7 and 8 utilize the default summation buffering method. Here, only the updates to the local copies of `PM_C` and `PM_S` are removed. Clearly, the reduced operation count of the multiplication variant results in better performance. Comparing the performance graphs of mean sets 3 and 7 from Figure 13 reveals that the relative performance difference between the implementations increases when increasing the iteration count. Figure 15 shows the time spent per component on input size 28. The communication overhead of the multiplication method is significantly lower when compared to the summation method. This difference comes from the difference in buffering between the methods. The actual data communicated, the means, is the same. In the multiplication method, initializing the local partitions increases initialization time. Increasing the iteration count on the same dataset does not increase initialization time. Therefore, the reduced overhead results in a much higher relative performance difference with increased iteration counts. Changing from local copies to local partitions has little effect on computation time. Point reassignment, the biggest computational part of the algorithm, takes a similar amount of time in all implementations.

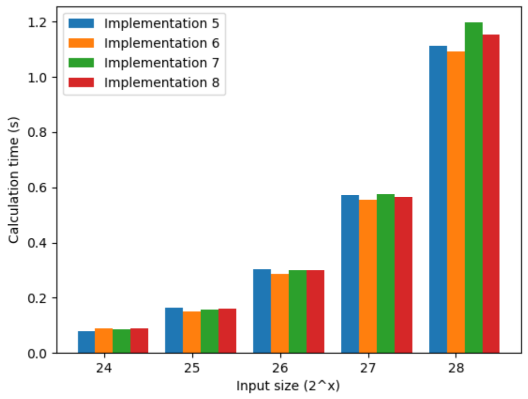


(a)

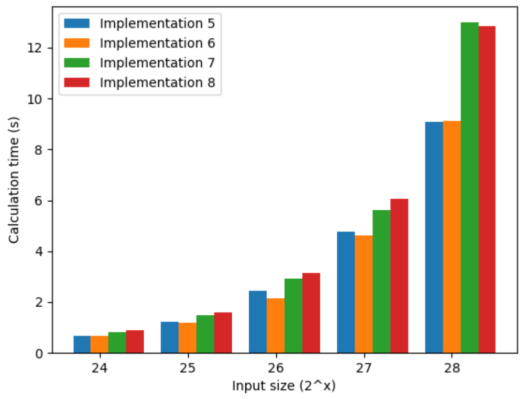


(b)

Figure 13: Input size variation on implementations 5 to 8. Experiment ran using 64 threads: 8 nodes with each 8 threads. Experiment ran on DAS6. Initialized using mean set 3 (a) and mean set 7 (b).

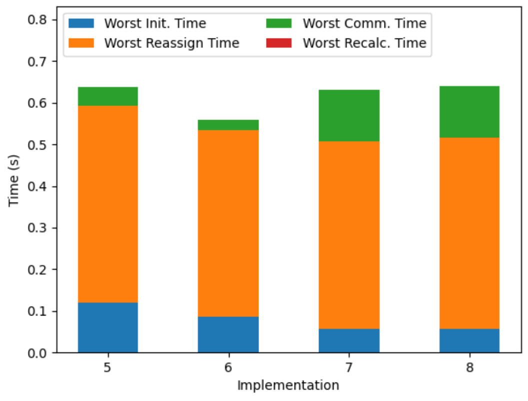


(a)

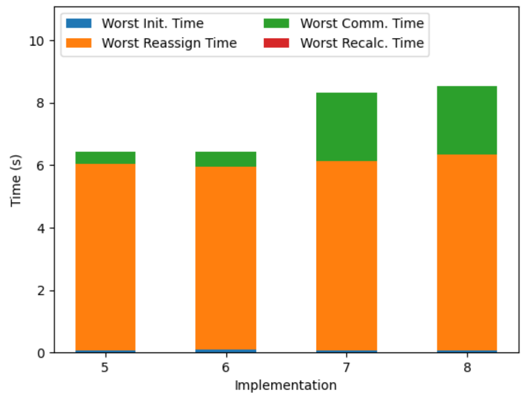


(b)

Figure 14: Input size variation on implementations 5 to 8. Experiment ran using 64 threads: 8 nodes with each 8 threads. Experiment ran on DAS5. Initialized using mean set 3 (a) and mean set 7 (b).



(a)



(b)

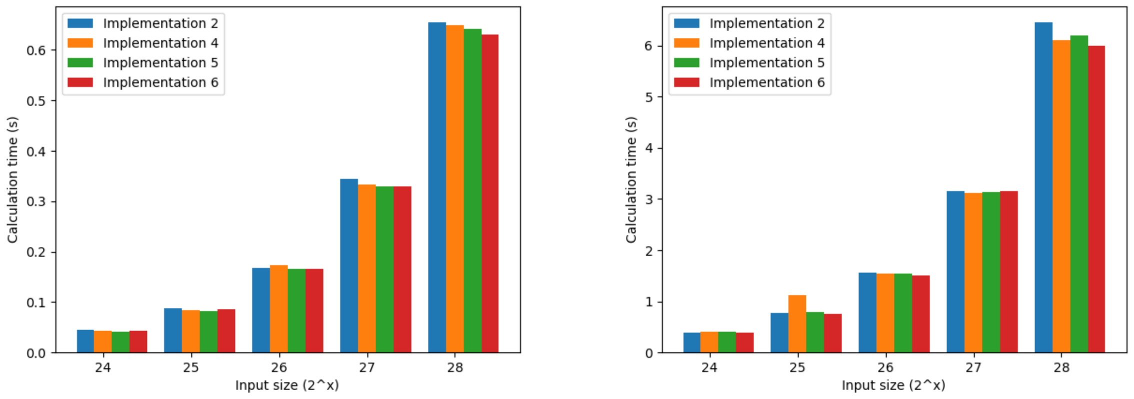
Figure 15: Stacked bar plot of DAS6 experiment 1. Bars show component timing per implementation from input size 28 from mean set 3 (a) and mean set 7 (b). Results are from runs presented in Figure 13.

6.2 Comparison to previously best variants

The previous best implementations were 2 and 4. Both used the derived scheme. They performed significantly better than their recalculation counterparts. Since implementations 5 and 6 improved the performance of the recalculation scheme, it makes sense to compare them to implementations 2 and 4. The DAS6 results of this comparison are shown in Figure 16. DAS5 results are shown in Figure 17. This time, performance is comparable between all implementations. Localization slightly improved performance on DAS6, while it mostly decreased performance on DAS5. As a result, there are slight performance differences between DAS5 and DAS6. However, these performance differences are small; performance between the presented variants is similar. With both mean sets, implementation 5 and 6 achieved slightly better performance; both on DAS5 and DAS6. The comparable performance between the mean sets is explained by the similarities between the buffering methods. All implementations use a variant of multiplication buffering with similar complexity; resulting in similar communication overhead. Additionally, point reassignment time remains similar. Logically, comparable computation and communication times result in comparable performance.

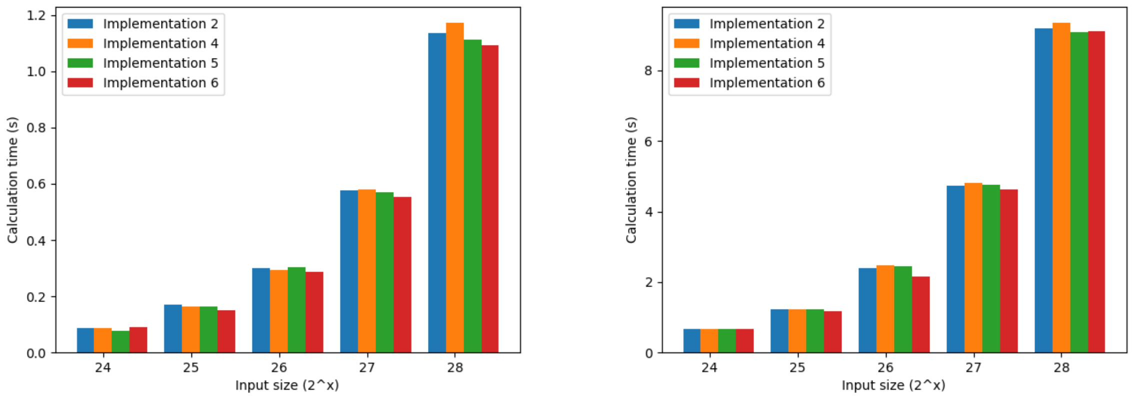
6.3 The impact of ignoring the k-means write-dependency

This experiment shows the difference between including and ignoring the PM_S write-dependency in the transformation process. In this experiment, implementation 1, including PM_S, is compared to implementation 9, ignoring PM_S, using dataset sizes [24-28]. Runs used 8 nodes with 8 threads each. The experiment was run on DAS6 with mean set 3, see Appendix A. The results are shown on Figure 18. Implementation 1 uses the recalculation scheme and has shared space localization and sync. delay applied to both PM_C and PM_S. Implementation 9 also uses the recalculation scheme, but PM_S is excluded from the communication transformations. As a result, PM_S is synchronized after each write. As expected, implementation 9 performs significantly worse. The added communication by updating PM_S after each point reassignment result in an enormous increase in communication. Implementation 9 spends $> 90\%$ of its execution time communicating. This experiment shows the importance of reducing communication as much as possible. It also shows that we need to specify algorithms differently, such as ‘convergence-like’, which is possible with tUPL.



(a) (b)

Figure 16: Input size variation on implementations 2, 4-6. Experiment ran using 64 threads: 8 nodes with each 8 threads. Experiment ran on DAS6. Initialized using mean set 3 (a) and mean set 7 (b).



(a) (b)

Figure 17: Input size variation on implementations 2, 4-6. Experiment ran using 64 threads: 8 nodes with each 8 threads. Experiment ran on DAS5. Initialized using mean set 3 (a) and mean set 7 (b).

6.4 Comparing communication and computation scaling

This experiment focuses on the effect of increasing the thread count on the computation and communication times. First, the scaling factors of computation and communication time were compared when adding threads. A distinction was made between adding intra-node or inter-node threads. With intra-node, all threads ran on a single node. With inter-node, each thread ran on a different node. All implementations used in this experiment where implemented using MPI. The thread count range was set to [2-16] with a step size of 1. The lower bound was set to 2 in order to receive a non-zero baseline for communication between threads. The upper bound was limited by the amount of standard nodes available. The scaling factors were calculated by dividing the baseline value of the corresponding variable (lower bound) by the value measured for that variable at each thread count. The lower bound receives factor 1. For example, to receive the scaling factor of

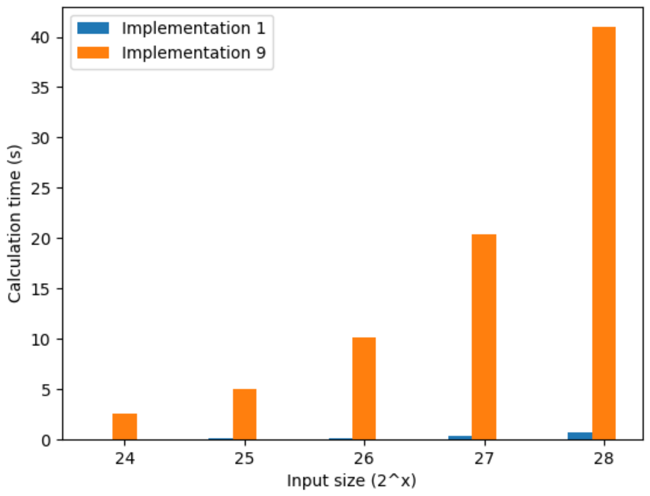


Figure 18: Input size variation on implementations 1 and 9. Experiment ran using 64 threads: 8 nodes with each 8 threads. Experiment ran on DAS6. Initialized using mean set 3.

the intra-node computation time at 8 threads, the intra-node computation time at 8 threads is divided by the intra-node computation time at 2 threads (lower bound). This experiment ran using mean set 3. The computation and communication times were averaged over 10 runs. All implementations follow similar trends in the measured components. Therefore, all results were averaged over runs from implementations 2, 4, 5, and 6. For example, the intra-node communication time presented is the average of the intra-node communication times of implementations 2, 4, 5, and 6. This experiment was run using the InfiniBand network.

Results for DAS6 are shown on Figure 19. These results clearly depict a gap between computation and communication scaling. Computation time scales nicely, doubling the scaling factor for every doubling of the thread count. This is expected, as more threads reduce the partition size per thread. However, this does not translate to communication. Communication time scales very poorly. As a consequence, with increased thread counts, the overall scaling will be limited by communication. This is evident even at this small scale. There is no significant difference between intra and inter-node times. This is expected for the computation part, as all nodes use identical hardware. However, intra-node communication was expected to outperform inter-node communication. However, at this scale, the results show no significant difference between communication times. The comparable communication time could be explained by DAS6's 100 Gb/s InfiniBand (IB) network[9]. Inter-node communication through IB is shown to have little impact on latency compared to intra-node communication [46].

Further increasing the thread counts provides a better view of the relation between computation and communication times. To this end, the average computation and communication times per thread were measured and compared for the thread count range [2-768]. This experiment ran on DAS6 and used mean set 3. The computation and communication times were averaged over 10 runs from implementations 2, 4, 5, and 6. The results are shown on Figure 20. When increasing the thread count, computation time keeps decreasing, while communication time stays roughly consistent; it even increases slightly. This results in a performance bottleneck, which becomes more noticeable after the turning point at around 300 threads. These observations concur with the statement that scalability is limited by communication [1]. The effect is similar when more iterations are required to converge; see the mean set 7 results on Figure 31.

[9] `https://www.cs.vu.nl/das6/network.shtml`

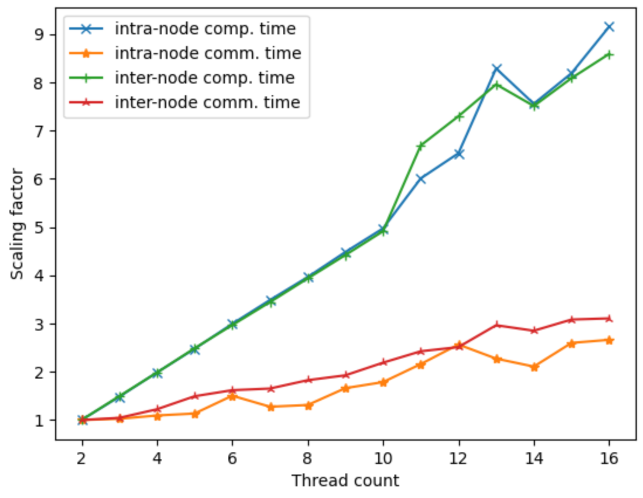


Figure 19: Comparison between scaling factors of intra and inter-node computation and communication times. Results averaged over 10 runs from implementations 2, 4, 5, and 6. Each line presents the average result from implementations 2, 4, 5, and 6 for that specific component. Experiment ran on DAS6 using mean set 3.

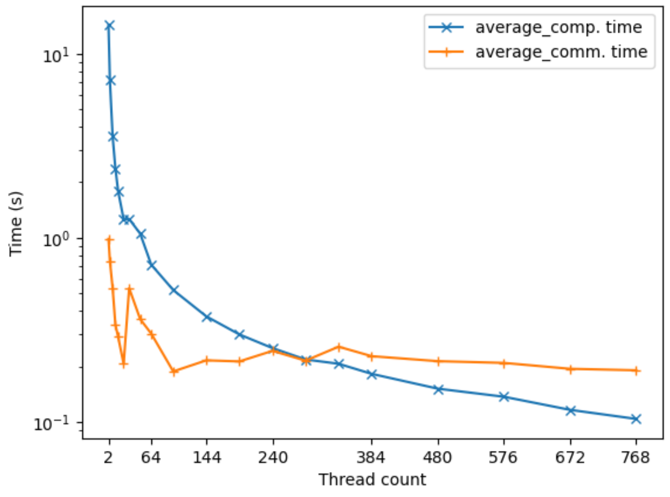


Figure 20: Comparison between computation and communication time per thread count. The y-axis is in logarithmic scale. Results averaged over 10 runs from implementations 2, 4, 5, and 6. Experiment ran on DAS6 using mean set 3.

7 Discussion

This thesis focuses on generating communication code that optimizes the balance in the computation vs communication trade-off. Results from Section 3 showed that different implementation variants can achieve the best performance on different hardware architectures. The experiments from Section 6 did not result in different variants performing best on the different hardware. However, there were still noticeable relative performance differences between the same variant being run on different hardware. These relative performance difference may not always result in different best performing variants, but the result from Section 3 demonstrates that it is possible. This possibility highlights the importance of platform-specific optimization; and most importantly, balancing the computation vs communication trade-off.

The nature of tUPL-specifications gives more freedom to the compiler in applying transformations. This allows tUPL to better optimize the implementation for a specific platform. This holds for communication schemes as well as regular transformations. The two available communication schemes already resulted in 10 different implementations. The result of defining transformations for this expands the amount of possible transformation paths, which may lead to better implementation variants. The same could be said about regular transformations such as localization.

By default, MPI has a static amount of processes called ranks. Comprés et al. [47] created an MPI extension that allows an elastic mode of execution. This extension enables runtime resource adaptation [48]. In MPI, this refers to adapting the amount of MPI ranks at runtime. In the context of communication schemes, the question then becomes whether changing the amount of MPI ranks at runtime affects communication; and if so, is it beneficial to change communication schemes when altering the MPI rank amount?

Using Sympy to automate the deduction of communication code in the synchronization delay transformation looks promising. Sympy offers a lot of functionality and additional operations can be added manually. However, currently, there are some limitations and shortcomings in the process. The recurrence solver only accepts recurrence with one generator, where each indexed variable is seen as a generator. Because of this, the recurrence solver is not able to solve Equation 3 for example. However, it should be used whenever possible. In addition, the generator should be a function, not an indexed variable; otherwise an 'IndexException' is raised. This can easily be overcome by using a mapping function that maps indexed expressions to function expressions and vice versa. It might be better to transform the process to support function-based expressions instead of index-based expressions. This depends on future additions and requirements. The biggest limitation is the simplification to sum-notation. Currently, it is build specifically for k-means Equation 3. It only supports step size 1 and does not support constant expansion and more complex operations that include the recurrence variable `n`. Fully developing and testing this functionality would take a considerable amount of time. One idea to improve upon the current implementation is to track what is generated at each step and use this in combination with the base step to construct a sum-notation. This way, more complex patterns could be recognized.

8 Conclusions

In this thesis, we introduced a number of transformations to deduce and insert communication code in materialized tUPL-specifications. With these new transformations, we demonstrated that the gap in the k-means derivation process from [19] can be closed. The new transformations resulted in new k-means implementations; some of which slightly improved upon the previous best variants. The tUPL transformation process enables the derivation of implementations with different computation and communication characteristics. This allows better balancing of computation and communication, and enables platform-specific optimizations. This proved beneficial in cases such as the observed relative performance difference between DAS5 and DAS6. The results indicated that scalability is limited by communication. This statement is supported by claims from different research [1], [2], [7], [8]. The gap between computation and communication scaling shows the importance of balancing the two. With such a big scaling difference, it is important to maximally utilize the computational scaling, while minimizing communication. We demonstrated the general applicability of the new transformations by applying them to tUPL pseudocode of sample sort.

A reproducibility study was performed in order to receive an accurate and reliable performance baseline. Randomness in both the dataset and the algorithms themselves produced results that were not statistically sound. To prevent this in future work, randomness was parameterized or removed entirely. Seeds used in experiments were reported. Additionally, all files required to reproduce the experiments are included in the project on GitHub [10]. The dataset generator used for the performance baseline tries to create a consistent data pattern for all input sizes. Point doubling was used to get a similar, consistent distance between point vectors.

The tUPL sample sort pseudocode was received by applying a concept bucketing partitioning transformation based on the divide-and-conquer transformation from [44]. Future work includes extending the partitioning transformation, including the applied bucket partitioning concept, in order to derive multiple sorting algorithms and variants.

The guided deduction through Sympy showed the possibilities of deducing reduction code using symbolic mathematics. The current process is designed specifically for k-means, limiting the options for general application. To automate the derivation of tUPL communication code, the derivation of reduction code using Sympy needs to be automated, for which a proposal was made in Section 5.2.4.

In this thesis, the primary use of loop blocking was to encapsulate the core loop and the communication code. However, Section 4.2.3 mentions the use of loop blocking to partition data, providing more allocation options. Currently, only MPI `C++` derivations are made. However, using loop blocking, this could be extended with shared-memory OpenMP and hybrid OpenMP/MPI derivation options. Future work might explore the possibilities of extending the current framework with support for OpenMP and hybrid derivations.

Synchronization delay delays the communication by moving the communication code up one scope. It is also possible to add another level of delay by encapsulating the core loop in a simple `forelem` loop that performs `n` iterations. Here, `n` is the amount of core loop cycles you want to perform before communicating. Another direction for future work is to explore the possibilities of extending the options of current transformations, possibly leading to more optimization options.

Other directions for future work are to include more communication scheme transformations and extending the derivation process to other algorithm classes.

[10] https://github.com/DennisBuurman/tUPL-to-MPI

References

[1] K. Bergman, S. Borkar, D. Campbell, *et al.*, “Exascale computing study: Technology challenges in achieving exascale systems,” *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, vol. 15, p. 181, 2008.

[2] M. T. Goodrich, “Communication-efficient parallel sorting (preliminary version),” in *Proceedings of the twenty-eighth annual ACM symposium on Theory of Computing*, 1996, pp. 247–256.

[3] D. W. Walker and J. J. Dongarra, “MPI: A standard message passing interface,” *Supercomputer*, vol. 12, pp. 56–68, 1996.

[4] L. Dagum and R. Menon, “OpenMP: An industry standard API for shared-memory programming,” *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[5] J. Diaz, C. Munoz-Caro, and A. Nino, “A survey of parallel programming models and tools in the multi and many-core era,” *IEEE Transactions on parallel and distributed systems*, vol. 23, no. 8, pp. 1369–1386, 2012.

[6] V. Amaral, B. Norberto, M. Goulão, *et al.*, “Programming languages for data-intensive HPC applications: A systematic mapping study,” *Parallel Computing*, vol. 91, p. 102 584, 2020.

[7] A. E. Helal, C. Jung, W.-c. Feng, and Y. Y. Hanafy, “CommAnalyzer: Automated estimation of communication cost and scalability on HPC clusters from sequential code,” in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, 2018, pp. 80–91.

[8] S. Li, M. A. Maddah-Ali, Q. Yu, and A. S. Avestimehr, “A fundamental tradeoff between computation and communication in distributed computing,” *IEEE Transactions on Information Theory*, vol. 64, no. 1, pp. 109–128, 2017.

[9] Q. Yan, S. Yang, and M. Wigger, “Storage-computation-communication tradeoff in distributed computing: Fundamental limits and complexity,” *IEEE Transactions on Information Theory*, vol. 68, no. 8, pp. 5496–5512, 2022.

[10] D. B. Loveman, “High Performance Fortran,” *IEEE Parallel & Distributed Technology: Systems & Applications*, vol. 1, no. 1, pp. 25–42, 1993.

[11] B. L. Chamberlain, D. Callahan, and H. P. Zima, “Parallel programmability and the chapel language,” *The International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.

[12] P. Charles, C. Grothoff, V. Saraswat, *et al.*, “X10: An object-oriented approach to non-uniform cluster computing,” *Acm Sigplan Notices*, vol. 40, no. 10, pp. 519–538, 2005.

[13] C. H. Koelbel, *The High Performance Fortran handbook*. MIT press, 1994.

[14] K. Kennedy, C. Koelbel, and H. Zima, “The rise and fall of High Performance Fortran: An historical object lesson,” in *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, 2007, pp. 7–1.

[15] K. F. D. Rietveld and H. A. G. Wijshoff, “Forelem: A versatile optimization framework for tuple-based computations,” in *CPC 2013: 17th Workshop on Compilers for Parallel Computing*, Citeseer, 2013.

[16] K. F. D. Rietveld and H. A. G. Wijshoff, “Towards a new tuple-based programming paradigm for expressing and optimizing irregular parallel computations,” in *Proceedings of the 11th ACM Conference on Computing Frontiers*, 2014, pp. 1–10.

[17] K. F. D. Rietveld and H. A. G. Wijshoff, “Optimizing sparse matrix computations through compiler-assisted programming,” in *Proceedings of the ACM International Conference on Computing Frontiers*, 2016, pp. 100–109.

[18] B. van Strien, K. F. D. Rietveld, and H. A. G. Wijshoff, “Deriving highly efficient implementations of parallel pagerank,” in *2017 46th International Conference on Parallel Processing Workshops (ICPPW)*, IEEE, 2017, pp. 95–102.

[19] A. Hommelberg, B. van Strien, K. F. D. Rietveld, and H. A. G. Wijshoff, “A new framework for expressing, parallelizing and optimizing big data applications,” *arXiv preprint arXiv:2203.01081*, 2022.

[20] S. Lloyd, “Least squares quantization in PCM,” *IEEE transactions on information theory*, vol. 28, no. 2, pp. 129–137, 1982.

[21] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer networks and ISDN systems*, vol. 30, no. 1-7, pp. 107–117, 1998.

[22] E. Gabriel, G. E. Fagg, G. Bosilca, *et al.*, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, Sep. 2004, pp. 97–104.

[23] E. Holk, W. E. Byrd, J. Willcock, T. Hoefler, A. Chauhan, and A. Lumsdaine, "Kanor: A declarative language for explicit communication," in *Practical Aspects of Declarative Languages: 13th International Symposium, PADL 2011, Austin, TX, USA, January 24-25, 2011. Proceedings 13*, Springer, 2011, pp. 190–204.

[24] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of parallel and distributed computing*, vol. 74, no. 12, pp. 3202–3216, 2014.

[25] E. Suggs, S. Olivier, J. Ciesko, and A. Skjellum, "View-aware message passing through the integration of kokkos and ExaMPI," in *Proceedings of the 30th European MPI Users' Group Meeting*, 2023, pp. 1–10.

[26] G. R. Andrews, "Foundations of multithreaded, parallel, and distributed programming," *Wesley, University of Arizona, USA*, 2000.

[27] T. Kadosh, N. Hasabnis, T. Mattson, Y. Pinter, and G. Oren, "Quantifying OpenMP: Statistical insights into usage and adoption," in *2023 IEEE High Performance Extreme Computing Conference (HPEC)*, IEEE, 2023, pp. 1–7.

[28] H. Richardson, "High Performance Fortran: History, overview and current developments," *Thinking Machines Corporation*, vol. 14, p. 17, 1996.

[29] J. Dongarra, R. Graybill, W. Harrod, *et al.*, "DARPA's HPCS program: History, models, tools, languages," in *Advances in Computers*, vol. 72, Elsevier, 2008, pp. 1–100.

[30] D. Callahan, B. L. Chamberlain, and H. P. Zima, "The Cascade high productivity language," in *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments, 2004. Proceedings.*, IEEE, 2004, pp. 52–60.

[31] M. De Wael, S. Marr, B. De Fraine, T. Van Cutsem, and W. De Meuter, "Partitioned Global Address Space languages," *ACM Computing Surveys (CSUR)*, vol. 47, no. 4, pp. 1–27, 2015.

[32] L. V. Kale and S. Krishnan, "Charm++ a portable concurrent object oriented system based on c++," in *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, 1993, pp. 91–108.

[33] D. Gelernter, "Generative communication in linda," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 7, no. 1, pp. 80–112, 1985.

[34] P. Ciancarini, "Distributed programming with logic tuple spaces," *New Generation Computing*, vol. 12, pp. 251–283, 1994.

[35] K. F. D. Rietveld and H. A. G. Wijshoff, "Automatic compiler-based data structure generation," *arXiv preprint arXiv:2203.07109*, 2022.

[36] D. Lehmann, A. Pnueli, and J. Stavi, "Impartiality, justice and fairness: The ethics of concurrent termination," in *Automata, Languages and Programming: Eighth Colloquium Acre (Akko), Israel July 13–17, 1981 8*, Springer, 1981, pp. 264–277.

[37] M. E. Celebi, H. A. Kingravi, and P. A. Vela, "A comparative study of efficient initialization methods for the k-means clustering algorithm," *Expert systems with applications*, vol. 40, no. 1, pp. 200–210, 2013.

[38] D. Arthur, S. Vassilvitskii, *et al.*, "K-means++: The advantages of careful seeding," in *Soda*, vol. 7, 2007, pp. 1027–1035.

[39] A. Meurer, C. P. Smith, M. Paprocki, *et al.*, "SymPy: Symbolic computing in Python," *PeerJ Computer Science*, vol. 3, e103, Jan. 2017, ISSN: 2376-5992. DOI: `10.7717/peerj-cs.103`. [Online]. Available: `https://doi.org/10.7717/peerj-cs.103`.

[40] A. Hommelberg, H. A. G. Wijshoff, and K. F. D. Rietveld, "Using the forelem framework to express and optimize k means clustering," *LIACS*, 2017. [Online]. Available: `https://theses.liacs.nl/pdf/AnneHommelberg3.pdf`.

[41] D. Żurek, M. Pietroń, M. Wielgosz, and K. Wiatr, “The comparison of parallel sorting algorithms implemented on different hardware platforms,” *Computer Science*, vol. 14, no. 4), pp. 679–691, 2013.

[42] E. Solomonik and L. V. Kale, “Highly scalable parallel sorting,” in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, IEEE, 2010, pp. 1–12.

[43] N. M. Amato, R. Iyer, S. Sundaresan, and Y. Wu, “A comparison of parallel sorting algorithms on different architectures,” *Technical Report TR98-029, Department of Computer Science, Texas A&M University*, 1996.

[44] A. Kling, “Automatically deriving sorting algorithms in tUPL,” *LIACS*, 2022. [Online]. Available: https://theses.liacs.nl/pdf/2022-2023-KlingA.pdf.

[45] P. Sanders and S. Winkel, “Super scalar sample sort,” in *European Symposium on Algorithms*, Springer, 2004, pp. 784–796.

[46] G. Shainer, P. Lui, T. Liu, T. Wilde, and J. Layton, “The impact of inter-node latency versus intra-node latency on HPC applications,” in *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems*, 2011, pp. 455–460.

[47] I. Comprés, A. Mo-Hellenbrand, M. Gerndt, and H.-J. Bungartz, “Infrastructure and API extensions for elastic execution of MPI applications,” in *Proceedings of the 23rd European MPI Users’ Group Meeting*, 2016, pp. 82–97.

[48] A. Mo-Hellenbrand, I. Comprés, O. Meister, H.-J. Bungartz, M. Gerndt, and M. Bader, “A large-scale malleable tsunami simulation realized on an elastic MPI infrastructure,” in *Proceedings of the Computing Frontiers Conference*, 2017, pp. 271–274.

A Reproducibility Appendix

Link to project on GitHub: `https://github.com/DennisBuurman/tUPL-to-MPI`

A.1 Initial mean sets

Initial mean set 3:

```
0: 8.1915 2.24708 6.10631 0.907039
1: 7.45863 10.6556 9.1931 0.620995
2: 4.83418 9.62381 8.77769 4.58893
3: 5.46514 0.758757 5.71774 4.03456
```

Initial mean set 7:

```
0: 4.21031 8.41086 9.74818 4.69075
1: 7.45863 10.6556 9.1931 0.620995
2: 4.83418 9.62381 8.77769 4.58893
3: 5.46514 0.758757 5.71774 4.03456
```

A.2 Compiling the code

Start by compiling the programs in the `generator` and `src` directories located in the `tupl-kmeans-39f1079` directory. To do this on DAS5, first load the Open MPI and gcc modules:

```
module load gcc/6.4.0
module load openmpi/gcc
```

For DAS6, the openmpi module is loaded using the command '`module load openmpi/gcc/64`' instead. After loading the modules, the code can be compiled using the Makefiles in the `generator` and `src` directories. This generates executables for all implementations and the data generator. The new dataset generator can be compiled using the Makefile in the `dataset_generator` directory. This generator was used to generate datasets for the experiments performed in this thesis.

A.3 Generating the datasets

Dataset creation is done through the `datasets.py` script in the `tupl-kmeans-39f1073/support` directory. All datasets can be created using one simple command:

```
./datasets.py   --seed 971 \
                --size 24 25 26 27 28 \
                --clusters 4 \
                --dimension 4 \
                --datapath <datapath> \
                create
```

This creates the datasets used for the experiments in this thesis. On the DAS systems, it is recommended to set `<datapath>` to '`/var/scratch/<account_name>/kmeans`', where `<account_name>` refers to your DAS account name.

A.4 Running the code

The performed experiments can be repeated by executing their corresponding `benchmark.py` configuration. Note, this only applies to DAS5 and DAS6. To do this, first set the `account_name` variable in `benchmark.py` to your DAS account name. Then, provide execute rights to the benchmark, preprocess, and visualization scripts using the command:

```
chmod +x benchmark.py input-size-variation.py
         thread-count-variation.py timing-stack.py
         scaling-factor.py comm-comp-tcv.py process-timing.py
```

The different experiments present in `benchmark.py` can be run using the following command:

```
./benchmark.py --c <cluster> \
               --e <ex_num> \
               --init-seed 340632450 \
               --m <means_set>
```

, where `<cluster>` is the compute cluster (DAS5 or DAS6), `<ex_num>` the experiment number (1-9), and `<means_set>` the initial means set to use. Note that there are more experiments available than reported in the thesis. This is because some experiments are variations of the same experiment.

After running `benchmark.py`, commands for visualizing the produced results are outputted. Execute the printed command(s) to create the graphs corresponding to the experiment performed. To perform the visualization manually, you can use the commands:

```
./input-size-variation.py   --compute-cluster <cluster> \
                            --file-date <date> \
                            --datapath <path> \
                            --ex-num <ex_num>
./thread-count-variation.py --compute-cluster <cluster> \
                            --file-date <date> \
                            --datapath <path> \
                            --ex-num <ex_num>
./timing-stack.py           --compute-cluster <cluster> \
                            --file-date <date> \
                            --datapath <path> \
                            --ex-num <ex_num>
```

, where `<cluster>` is the compute cluster (DAS5 or DAS6), `<date>` the date suffix from the results file produced by benchmark.py (format ‘dd-mm-yyyy’), and `<ex_num>` the experiment number (1-9).

In order to add experiments to the benchmark, you could add a configuration following the dictionary structure of other experiment configs. To enable the new experiment, it must be added to the `ex_nums` list and the experiment selection if-else sequence in `main()`.

New Implementation variants should be added to the `tupl-kmeans-39f1073/src` directory. Add the source code of the new variant and update the Makefile. Additionally, update the `names` list in `initial_benchmarks/common.py` as well as the `execs` list in `initial_benchmarks/benchmark.py`. The `benchmark.py` script uses the `submit-exp.py` script from the `tupl-kmeans-39f1073/support` directory. Adding implementation variants to `benchmark.py` therefore requires you to also add them to `submit-exp.py`!

B Mathematical Deduction

The deduction of communication code presented in Section 5.2.2 is based on this mathematical deduction. This mathematical deduction shows how the guided process from Sympy came to be. Recall, k-means has the following 'core' operations:

$$\begin{aligned}
\text{PM\_C}[\text{PM}[i]] &= \frac{\text{PM\_C}[\text{PM}[i]] * \text{PM\_S}[m] - \text{PC}[i]}{\text{PM\_S}[m] - 1} \\
\text{PM\_S}[\text{PM}[i]] &= \text{PM\_S}[\text{PM}[i]] - 1 \\
\text{PM\_C}[m] &= \frac{\text{PM\_C}[m] * \text{PM\_S}[m] + \text{PC}[i]}{\text{PM\_S}[m] + 1} \\
\text{PM\_S}[m] &= \text{PM\_S}[m] + 1 \\
\text{PM}[i] &= m
\end{aligned}$$

First, assume shared space localization has been applied to `PM_C` and `PM_S`. `PM_C` is chosen as main target, `PM_S` is included as write-dependency. Before any deduction can be made, the operations are written as post-conditions:

$$post(\text{PM}[i]) = m \tag{1}$$
$$post(\text{PM\_S}[\text{PM}[i]]) = \text{PM\_S}[\text{PM}[i]] - 1 \tag{2}$$
$$post(\text{PM\_S}[m]) = \text{PM\_S}[m] + 1 \tag{3}$$
$$post(\text{PM\_C}[m]) = \frac{\text{PM\_C}[m] * \text{PM\_S}[m] + \text{PC}[i]}{post(\text{PM\_S}[m])} \tag{4}$$
$$post(\text{PM}[i]) = m \tag{5}$$

where $post(x)$ refers to the value of x right after the execution of the operation. Communicating `PM_C` comes down to a simple fraction $a = \frac{b}{c}$, where $a = post(\text{PM\_C}[m])$, $b = \text{PM\_C}[m] * \text{PM\_S}[m] + \text{PC}[i]$, and $c = post(\text{PM\_S}[m])$. This equation can be rewritten to $b = a * c$. Now, we can also communicate

$$post(\text{PM\_C}_g[m]) = \frac{\sum_{i \in p} post(\text{PM\_C}_i[m]) * post(\text{PM\_S}_i[m])}{\sum_{i \in p} post(\text{PM\_S}_i[m])} \tag{6}$$

where p is the set of processes and g denotes the globally synced version of `PM_C`. Note that the $_i$-subscript in `PM_C` and `PM_S` denotes a local partition. Otherwise, p times the mean would be communicated. Equation 6 denotes a multiplication variant of the recalculation scheme. This equation can be rewritten to receive the incremental (derived scheme) variant:

$$post(\text{PM\_C}_g[m]) = \frac{\sum_{i \in p} post(\text{PM\_C}_{ig}[m]) * post(\text{PM\_S}_{ig}[m]) - (p-1) * pre(\text{PM\_C}_g[m] * \text{PM\_S}_g[m])}{\sum_{i \in p} post(\text{PM\_S}_{ig}[m]) - (p-1) * pre(\text{PM\_S}_g[m])} \tag{7}$$

Here, $pre(x)$ refers to the value of x right before entering the current iteration. The additions from applying the derived scheme do not require extra deduction steps. Applying the incremental updates to the communicated parts is sufficient. The original recalculation scheme as presented in [19] performs a full recalculation of points before communicating. This changes Equation 6 to

$$post(\text{PM\_C}_g[m]) = \frac{\sum_{i \in p} \left(\sum_{x \in |\text{PM\_C}_i|} \text{PC}[x] \right)}{\sum_{i \in p} \left(\sum_{x \in |\text{PM\_S}_i|} 1 \right)} \tag{8}$$

Deducing Equation 8 does require additional steps. First, we rewrite "$post(\text{PM\_C}[m]) * post(\text{PM\_S}[m]) = \text{PM\_C}[m] * \text{PM\_S}[m] + \text{PC}[i]$" (referring to $a * c = b$) to

$$\text{PM\_C}_{i+1} * \text{PM\_S}_{i+1} = \text{PM\_C}_i * \text{PM\_S}_i + \text{PC}_i \tag{9}$$

Let's say we start at step $i = 0$. If we assume we start with empty shared spaces PM_C and PM_S we receive

$$\text{PM\_C}_1 * \text{PM\_S}_1 = \text{PC}_0 \tag{10}$$

Both PM_C and PM_S are 0. For $i = 1$ we get

$$\text{PM\_C}_2 * \text{PM\_S}_2 = \text{PM\_C}_1 * \text{PM\_C}_1 + \text{PC}_1 \tag{11}$$

This can be simplified to

$$\text{PM\_C}_2 * \text{PM\_S}_2 = \text{PC}_0 + \text{PC}_1 \tag{12}$$

Inductively, for any integer number of data points n, with $n > 0$, this comes down to

$$\text{PM\_C}_n * \text{PM\_S}_n = \sum_{i=0}^{n-1} \text{PC}_i \tag{13}$$

Equation 13 can be expanded, resulting in

$$\text{PM\_C}_n * \text{PM\_S}_n + \text{PC}_n = \sum_{i=0}^{n} \text{PC}_i \tag{14}$$

which is a substitution candidate for b. Given that the core specification starts with initialized shared spaces `PM_C` and `PM_S`, it might be hard to assume a size of 0 for `PM_C` and `PM_S` without knowing semantics. Without assumptions, we get

$$\text{PM\_C}_n * \text{PM\_S}_n = \text{PM\_C}_{n-i} * \text{PM\_S}_{n-i} + \sum_{j=n-i}^{n-1} \text{PC}_j \tag{15}$$

$$\sum_{j=n-i}^{n-1} \text{PC}_j = \text{PM\_C}_n * \text{PM\_S}_n - \text{PM\_C}_{n-i} * \text{PM\_S}_{n-i} \tag{16}$$

where $n \geq 1$, $i \geq 1$, and $n - i \geq 0$. Equation 16 shows that the difference between 2 versions, n and i, is the sum of coordinates `PC` in between them. Now, if $n = i$, we can create a base case

$$\sum_{j=n-i}^{n-1} \text{PC}_j = \text{PC}_{n-i} \tag{17}$$

Then, if $n = i = 1$, we have

$$\sum_{j=1-1}^{1-1} \text{PC}_j = \text{PC}_0 \tag{18}$$

reaching the same conclusion, but without assumptions.

Equation 14 can be used to substitute the numerator in Equation 3, resulting in

$$\text{PM\_C}_n = \frac{\sum_{i=0}^{n-1} \text{PC}_i}{\text{PM\_S}_n} \tag{19}$$

The last step is to rewrite PM_S. Equation 4 can be rewritten as follows:

$$\begin{aligned}
\text{PM\_S}_{n+1} &= \text{PM\_S}_n + 1 \\
\text{PM\_S}_{n+1} - \text{PM\_S}_n &= 1 \\
\text{PM\_S}_{n+2} &= \text{PM\_S}_{n+1} + 1 \\
\text{PM\_S}_{n+2} &= \text{PM\_S}_n + 1 + 1 \\
\text{PM\_S}_n &= \text{PM\_S}_{n-i} + \sum_{j=n-i}^{n-1} \\
\text{PM\_S}_n &= \sum_{i=0}^{n-1} 1 \qquad (20)
\end{aligned}$$

where $n \geq 0$, $i \geq 0$, and $n - i \geq 0$. Note that, changing the step index in the post-condition from n to $n+1$ only affects the domain of n and i. It has no effect on the actual outcome. Substituting Equation 20 into Equation 19 presents the summation variant of the recalculation scheme (as used in [19]):

$$\text{PM\_C}_n = \frac{\sum_{i=0}^{n-1} \text{PC}_i}{\sum_{i=0}^{n-1} 1} \qquad (21)$$

Because PC is read-only, the summation variant from Equation 21 is always applicable when PM_C is set as main target. This is not the case for the multiplication variant used in Equations 6, 7. The final result for the multiplication variant is received by substituting the summation of PC in Equation 21 with PM_C*PM_S from Equation 13. However, this option is only available if both PM_C and PM_S are included in the targeted set.

C Complementary Results

	m3	m7	Diff (%)
Calc.	0.7724765	8.063634	**943.87**
Init.	0.058645	0.08890001	**51.59**
Reassign.	0.4864187	6.008993	**1135.35**
Comm.	0.1301984	2.134715	**1539.59**
Iterations	5	71	**1320**

Table 5: Comparing the time spent per calculation component for mean set 3 and 7. Implementation 1 is used. Time is denoted in seconds. Results are taken from benchmark experiment 1 from DAS6, see Figure 8.

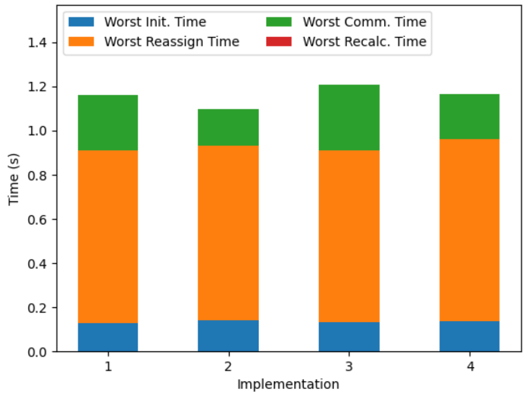
	m3	m7	Diff (%)
Calc.	0.6603727	6.210005	**840.38**
Init.	0.05837327	0.05221226	**-10.55**
Reassign.	0.4758292	6.016805	**1164.49**
Comm.	0.0574466	0.4526363	**687.93**
Iterations	5	71	**1320**

Table 6: Comparing the time spent per calculation component for mean set 3 and 7. Implementation 2 is used. Time is denoted in seconds. Results are taken from benchmark experiment 1 from DAS6, see Figure 8.

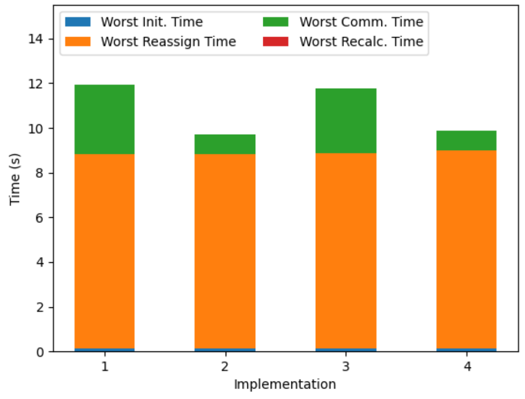
Algorithm 15 Applying a second synchronization delay to 4

```
whilelem ($i \in [0, |\text{PM}| - 1]$)
  forelem ($m \in [0, k - 1]$)
    if ($\text{PM}[i] \mathrel{!}= m$ && $\text{dist}(\text{PC}[i], \text{PM\_C}_l[m]) < \text{dist}(\text{PC}[i], \text{PM\_C}_l[\text{PM}[i]])$) {
      $\text{PM\_C}_l[\text{PM}[i]] = (\text{PM\_C}_l[\text{PM}[i]] * \text{PM\_S}_l[\text{PM}[i]] - \text{PC}[i])/(\text{PM\_S}_l[\text{PM}[i]] - 1)$
      $\text{PM\_S}_l[\text{PM}[i]] -= 1$
      $\text{PM\_C}_l[m] = (\text{PM\_C}_l[m] * \text{PM\_S}_l[m] + \text{PC}[i])/(\text{PM\_S}_l[m] + 1)$
      $\text{PM\_S}_l[m] += 1$
      $\text{PM}[i] = m$
    }
forelem ($i \in [0, |\text{PM}| - 1]$) {
  $\text{PM\_S\_B}[\text{PM}[i]] += 1$
  $\text{PM\_C\_B}[\text{PM}[i]] += \text{PC}[\text{PM}[i]]$
}
forelem ($m \in [0, k - 1]$)
  $\text{PM\_C}_l[m] = \text{PM\_C\_B}[m]/\text{PM\_S\_B}[m]$
```

C.1 Benchmark experiment 1



(a)



(b)

Figure 21: Stacked bar plot of DAS5 input size variation experiment. Bars show component timing per implementation from input size 28 using mean set 3 (a) and mean set 7 (b). Results are from runs presented in Figure 7.

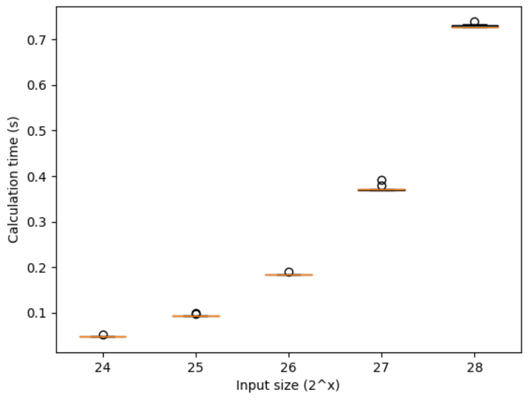


Figure 22: Box plots comparing calculation times per run, per input size on Implementation 3. Runs are taken from DAS6 benchmark experiment 1 were used, see Figure 8.

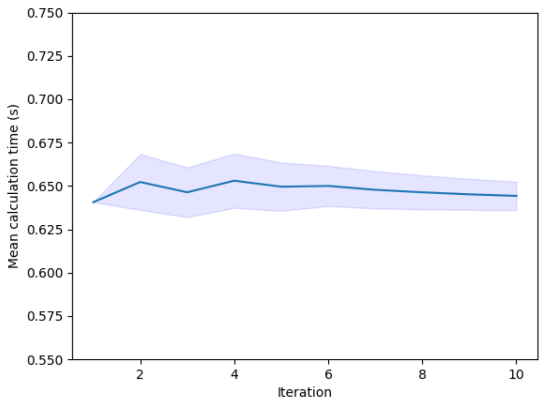
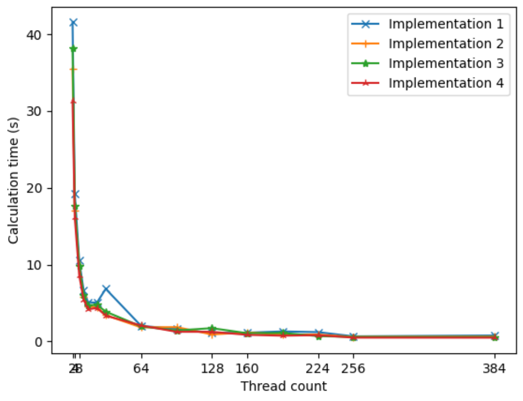
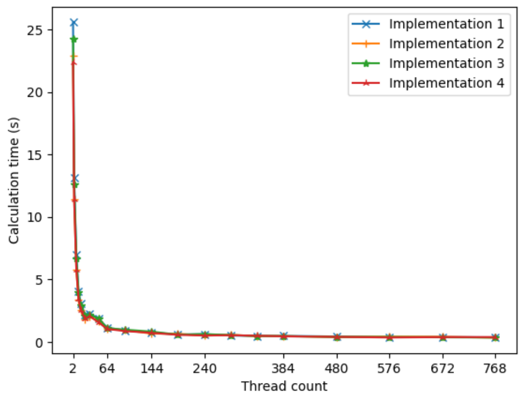


Figure 23: 95% CI plot over the 10 runs on Implementation 4 from DAS6 benchmark experiment 1, see Figure 8. The blue line represents the mean, while the area around it represents the CI.

C.2 Benchmark experiment 2



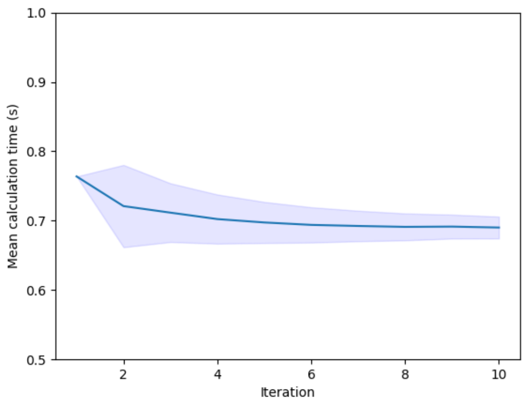
(a)



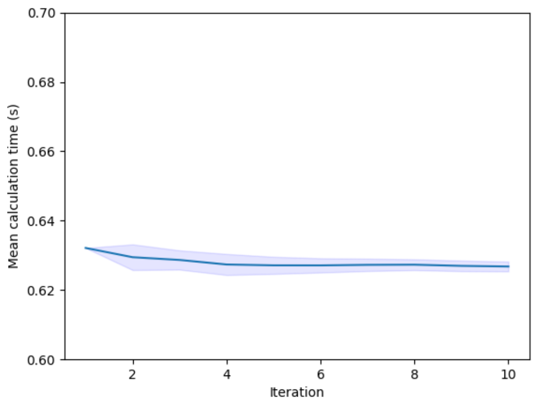
(b)

Figure 24: Thread count variation on implementations 1 to 4. Results are averaged over 10 runs. Dataset was of size 28. A convergence delta of 0.0001 was used. Experiment ran on DAS5 (a) and DAS6 (b). Initialization seed is set to 340632450. Initialized using mean set 3.

C.3 Experiments with new variants

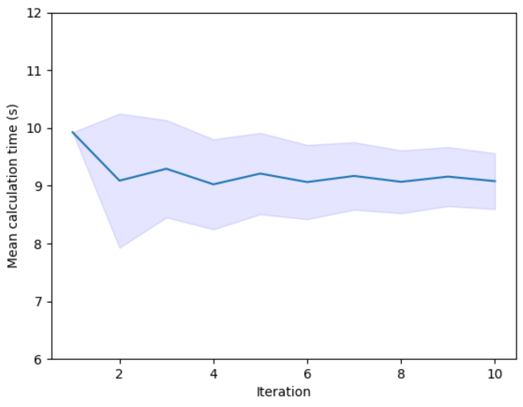


(a)

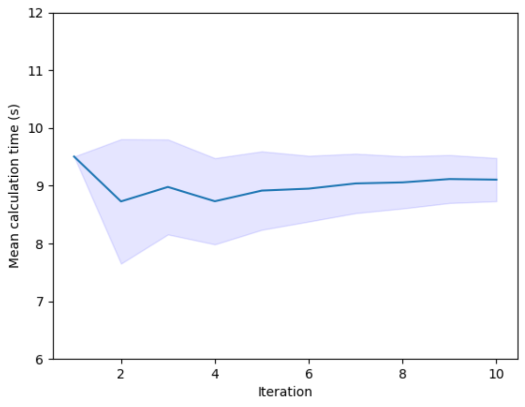


(b)

Figure 25: 95% CI plot over the 10 runs on Implementations 5 (a) and 6 (b) from DAS6 experiment 1, see Figure 13. The blue line represents the mean, while the area around it represents the CI.

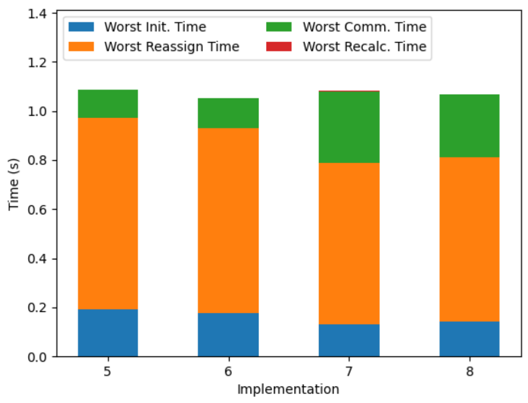


(a)

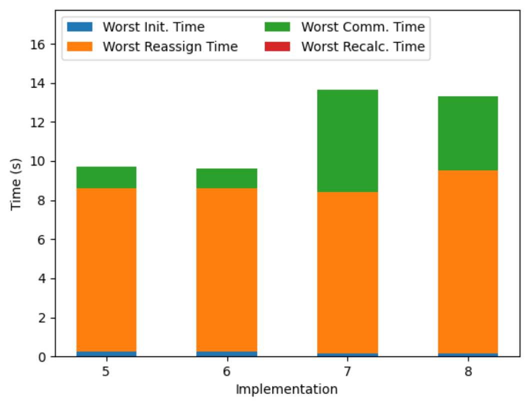


(b)

Figure 26: 95% CI plot over the 10 runs on Implementations 5 (a) and 6 (b) from DAS6 experiment 1, see Figure 14. The blue line represents the mean, while the area around it represents the CI.

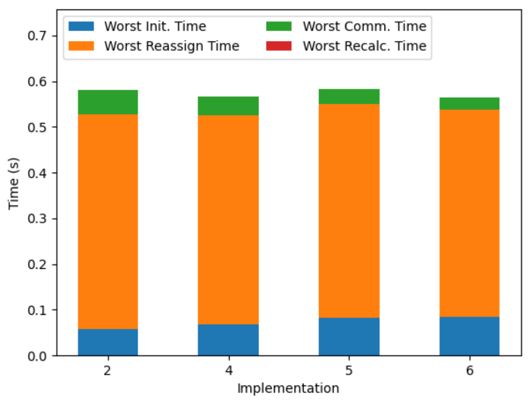


(a)

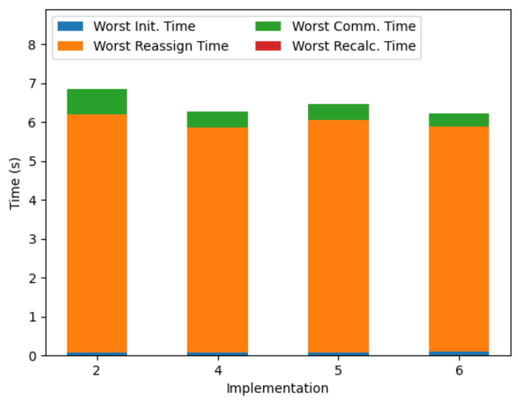


(b)

Figure 27: Stacked bar plot of DAS5 experiment 1, see Figure 14. Bars show component timing per implementation from input size 28 from mean set 3 (a) and mean set 7 (b).

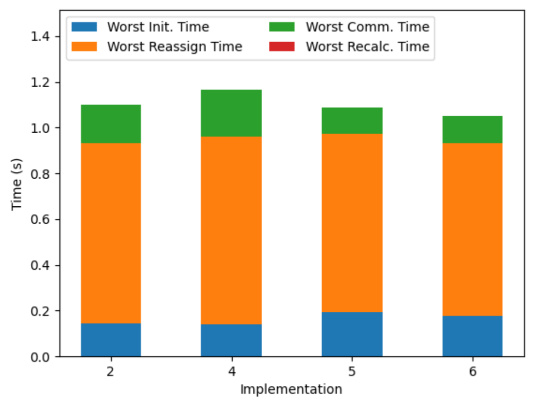


(a)

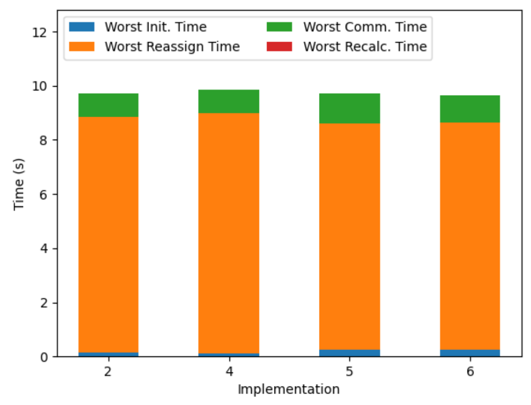


(b)

Figure 28: Stacked bar plot of DAS6 experiment 2, see Figure 16. Bars show component timing per implementation from input size 28 using mean set 3 (a) and mean set 7 (b).



(a)



(b)

Figure 29: Stacked bar plot of DAS5 experiment 2, see Figure 16. Bars show component timing per implementation from input size 28 using mean set 3 (a) and mean set 7 (b).

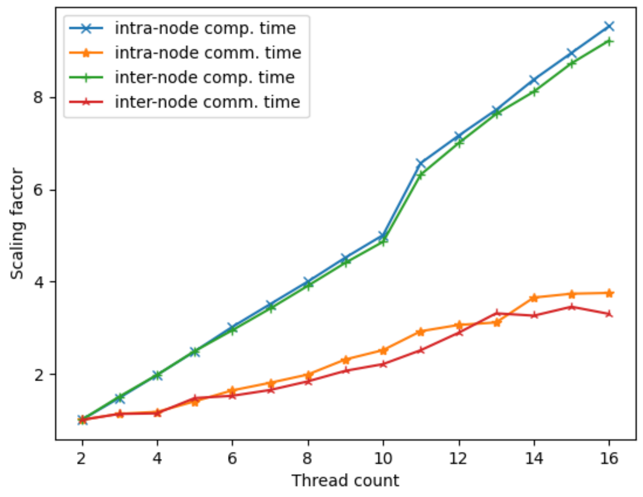


Figure 30: Comparison between scaling factors of intra and inter-node computation and communication times. Results averaged over 10 runs from implementation 2. Experiment ran on DAS6 using mean set 3.

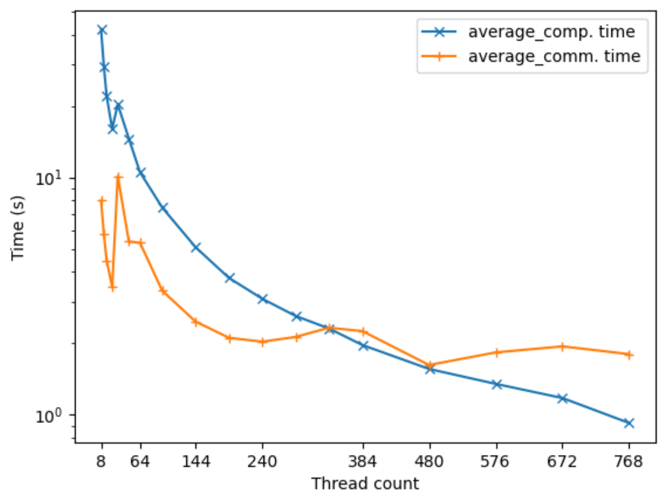


Figure 31: Comparison between computation and communication time per thread count. The y-axis is in logarithmic scale. Results averaged over 10 runs from implementations 1, 2, 3, and 4. Experiment ran on DAS6 using mean set 7.