Internal Report 09-04 March 2009

Universiteit Leiden

Opleiding Informatica

Distributed Approaches for Discovering Unique Factors in the Human Genome

Kristian Rietveld

BACHELORSCRIPTIE

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

Voor Ginny

Contents

1 Introduction **3**
1.1 Previous work . . . 3

2 The problem **5**
2.1 General algorithm . . . 5
2.2 Distributing the work . . . 6
2.3 Multi-core processors versus multiple cluster nodes . . . 8
2.4 Development of the resulting application . . . 9
2.5 Hardware used for research . . . 9
2.6 Data set used for research . . . 10

3 Splitting using the index table **11**
3.1 Details of the algorithm . . . 11
3.1.1 Splitting up the work . . . 11
3.1.2 Combining the work . . . 12
3.2 Future research . . . 14

4 Splitting using substring prefixes **16**
4.1 Details of the algorithm . . . 16
4.1.1 Splitting up the work . . . 16
4.1.2 Work done in parallel . . . 17
4.1.3 Combining the work . . . 18
4.2 Performance . . . 19
4.3 Future research . . . 21

5 Conclusions **22**

A Description of programs written **24**
A.1 Pthread-based version . . . 24
A.2 MPI-based version . . . 25

1 Introduction

The Human Genome, which is found in all of our body cells, is a fascinating large data set. It contains 3×10^9 nucleotides, or "characters", resulting in about 3 gigabyte of ASCII data. Compared with other contemporary data sets this may not be very large, however the entire genome is replicated many million times in our body, which is one of the facts that makes it very fascinating. Finding patterns in this data is much more time consuming than one would expect and the useful data that results as output can be in the order of many gigabytes. Much research is nowadays done in the area of computational biology and bioinformatics, pioneered by Gusfield [2].

DNA is built up from nucleotides, any of which contains one of the 4 bases ("A", "C", "G" and "T"). These 4 characters form the alphabet from which strings can be created. In cells the DNA has a double helix structure with two strands of nucleotides, both containing the same information. The second strand is the reverse complement of the first one. Complements are formed by considering the pairs between nucleotides: "T" is the complement of "A" and "G" of "C". We are speaking of a reverse complement because the second string is read in the other direction.

In most problems the reverse complement of the genome also has to be considered (even though it can be generated from the first DNA strand), which will double the size of the problem space. What we actually have to analyze is now a string of 6 gigabytes. A string of such a length calls for the development of new and more advanced algorithms if we want to get any calculations involving the entire genome done in reasonable time.

One of the problems that is useful to tackle is finding all unique substrings in the genome. A substring is unique if it only appears once in both the first DNA strand and its reverse complement. These unique substrings have several useful applications. One of these is the creation of primers. Researchers often want to isolate a specific piece of DNA. Several techniques exist for this and they all (obviously) need a way to mark exactly which part of the DNA should be isolated or duplicated. Primers are unique or rare substrings of arbitrary length that are used for making these marks.

Efficient algorithms for extracting unique substrings from the genome have already been researched. These algorithms do run in a reasonable amount of time. They are limited by the length of the substrings that they can process efficiently. When used on the entire human genome, it appears that as soon as the length of the substrings requested increases above 15, the time that is needed to complete the request exponentially increases, caused by the fact that current technology machines do not have enough main memory available. In this thesis we will describe distributed approaches for solving the problem of finding unique factors in the human genome. Instead of a single processing unit, we will make use of super computer clusters that have a large amount of processing power and main memory available.

Next we will briefly look at previous work that has been done in this area. In chapter 2, we will look at a more general formulation of the problem and the basics of solving this in a distributed fashion. Chapters 3 and 4 provide details on two distributed approaches that have been implemented and experimented with.

1.1 Previous work

In Laros [4] several methods are discussed for determining all unique primers. It was concluded that solving this problem is possible up to a certain length. The main problem was that the

primers and the number of times that they were encountered could not be stored in main memory all at once. If this were possible, the problem would be solvable in linear time. A distributed approach can help here by exploiting the fact that many cluster nodes combined have a lot of main memory available.

The main improvements in [4] were established by encoding the ASCII DNA data into a binary file. This binary file format compresses the original data by a factor of 2. Also it improves the calculation speed as we can now use simple binary arithmetic instead of more expensive string operations. The best performance was achieved by calculating the solution in multiple passes. Because of memory limitations it is not feasible to process the entire problem space at once. In each pass the program searches for all unique factors starting with a given prefix. This prefix changes with each pass through the DNA.

We will re-use the binary file format and other tools devised during the work for this thesis.

Van Vuurden discusses the usage of truncated suffix trees [7] in [11]. Suffix trees ([10] and [6]) can describe the uniqueness of strings of arbitrary length. Memory is saved by setting a maximum on this length, which truncates the suffix tree. If the substrings also need to satisfy one or more non-decreasing functions, the suffix tree can be pruned even more, further decreasing the amount of memory required. As long as the suffix tree fits in main memory, creation and analysis of this tree can be done in linear time.

Application of these methods on DNA has been said to be possible up to a maximum substring length of 50 and with the functions for GC-percentage and melting temperature applied. As the ideal length for a primer laid between 10 and 30 at the time of this research, the method was deemed to be suitable for discovering unique substrings. Unfortunately, the tree representing the entire human genome could not be fitted into main memory with the current technology at that time.

2 The problem

The problem that we want to solve can be formulated as follows: we want to find all unique substrings in the genome, with a fixed length of 32 and larger, in a reasonable amount of time. We have seen in the introduction that the methods discussed in previous work do not scale beyond a length of about 15. In this chapter we will discuss the basic algorithm that has been devised to solve this problem. We will show how this algorithm can be parallelized. At last we will discuss the first steps that were taken to implement this approach, which lays the basis for Chapters 3 and 4.

2.1 General algorithm

We consider the genome as a long string of data. We can then say that a substring starts at each offset in the string. Each of these substrings will have to be considered for uniqueness, also all substrings in the reverse complement. The reverse complement substrings can be found by walking the same string in the reverse direction.

Space complexity is the most important factor to consider when designing the algorithm. We have chosen to only store offsets to substrings in the genome in the main memory and not the substrings themselves. Considering that the length of the human genome is about 3×10^9 nucleotides, we need space for 3×10^9 offsets. An offset fits in a normal 4 byte integer. This means we need about $4 \times 3 \times 10^9$ bytes (or about 12 gigabytes) of main memory to store all the offsets. To account for all offsets for analyzing the reverse complement, this amount has to be doubled. So, in total about 24 gigabytes of main memory is needed to perform this calculation excluding the memory needed for storing the genome itself.

Another reason for storing offsets into the genome and not the substrings themselves is that this gives us linear space complexity no matter what the requested length of the substrings to extract is. This will allow us to work with any substring length we would like without increasing the memory requirements. Now the size of the table containing the offsets will stay more or less constant (it will actually be smaller, because the genome contains less substrings of length 64 than length 15 for example), we can expect the time required for sorting to be constant as well as long as the number of substrings to sort is much smaller than the total number of possible strings 4^{length} that can be formed with the alphabet. So our algorithm will be both constant in space and time for a given genome (linear in the length of the genome).

After we have filled a table in memory with offsets that all point to substrings in the DNA, we have to sort this table. We sort this table by comparing the substrings in DNA pointed to by the offsets. This means that the offsets in the resulting table are not sorted according to their actual integer value, but to the value of the substring they point to. Any sort method can in theory be used for this step. We have chosen to use quicksort, as this algorithm is known to be one of the most efficient ones available with well optimized versions in each C library. In the future an implementation of the faster radix sort algorithm can be investigated. A simple DNA substring comparison function is used as the compare function in the quicksort routine. The output is a truncated suffix array ([5], [3]), which means that every substring of length ℓ *and smaller* can be found efficiently with this array. In a normal suffix array, *every* substring can be found.

Finally, we can simply iterate over the sorted table of offsets to count how often each substring occurs in the genome. We need to compare each string a single time. We know that all strings are in a sorted order, so as soon as two strings passed to the compare function are

not equal, we can start the count for the a string. The count that is saved is either zero, one, two or three, corresponding to whether the substring is unique, the substring occurs twice, thrice, or four times or more. The algorithm that is used for this unique counting is shown in Algorithm 1. In this algorithm, *current* and i are indices into the array of offsets. A call to $substring(i)$ returns the substring pointed to by the ith offset in the offset table. The result of the algorithm is a filled table *substring_count* that contains how often a substring exists in the genome. Index i into *substring_count* corresponds with the substring located at the offset described by the ith element of the offset table.

Algorithm 1 Unique counting algorithm

```
1:  i ← 0
2:  while i < #offsets do
3:     count ← 0
4:     current ← i
5:     i ← i + 1
6:     while i < #offsets and substring(current) = substring(i) do
7:        count ← count + 1
8:        i ← i + 1
9:     end while
10:    if count > 3 then
11:       count ← 3
12:    end if
13:    for j = current to i − 1 do
14:       substring_count(j) ← count
15:    end for
16: end while
```

The table that is the result of executing this algorithm uses a (truncated) suffix array to count substrings. The same array also allows us to find unique substrings that are smaller than the substring count that has been used to creating the suffix array.

2.2 Distributing the work

One of the advantages of the above algorithms is that it can be very easily distributed over multiple processors. This can be done by splitting the genome in several pieces. Each processor can be assigned one of these pieces. The sorting of the offsets table can then be done in parallel. The hardest part is adapting the unique count procedure, which has to become a "distributed" algorithm. We have to merge sort several tables, that are this time not all located in local memory, but in the memories of the cluster nodes used. Then we can apply unique counting to the entire table. This part of the algorithm can not be done in parallel, as the same substring might be on several nodes.

The core of this algorithm is that we request the current smallest substring from each node. If we have split up the genome over n nodes, we will receive n substrings. One of these substrings must be the smallest that is currently available and will be picked. We need to signal the node corresponding to the picked substring, so that it can increase the pointer that points to its current smallest substring. The algorithm will terminate when all nodes are out of substrings. So instead of walking over a single offsets table, we will walk over a virtual table

that is created by filling it with the next smallest item from the n nodes until the available substrings at the nodes are exhausted.

Incorporating these changes, the distributed unique count algorithm is listed as Algorithm 2. Note that *current* and *substring*() now use the actual offsets contrary to an index to the offset in the array as in algorithm 1. The function *get_smallest* will return the number of the slave that contains the smallest item (in variable *smallest*) and the offset of the smallest substring. This function will also notify the node that sent the offset of the smallest substring. When all nodes are out of substrings, -1 will be returned in the variable *smallest*.

Algorithm 2 Distributed merge sort and unique counting algorithm

```
1:  i ← 0
2:  smallest, offset ← get_smallest()
3:  while true do
4:    count ← 0
5:    current ← offset
6:    if smallest = −1 then
7:      return
8:    end if
9:    push offset
10:   smallest, offset ← get_smallest()
11:   while smallest ≠ −1 and substring(current) = substring(offset) do
12:     if count < 3 then
13:       push offset
14:     else if count = 3 then
15:       for i = 0 to count do
16:         pop tmp_offset
17:         substring_count(offset) ← count
18:       end for
19:       substring_count(offset) ← count
20:       count ← count + 1
21:     else if count > 3 then
22:       substring_count(offset) ← 3
23:       count ← count + 1
24:     end if
25:   end while
26:   if count ≤ 3 then
27:     for i = 0 to count do
28:       pop tmp_offset
29:       substring_count(tmp_offset) ← count
30:     end for
31:   end if
32: end while
```

2.3 Multi-core processors versus multiple cluster nodes

These days the de facto standard processors sold for desktop machines are multi-core processors. As of this writing processors with 2 or 4 cores are popular, basically giving you the power of a 2 or 4 processor system, all with access to the same memory. For this kind of systems we can load the genome into memory once and this can be shared amongst calculation processes running on the available processor cores. We can split up the problem into 2 or 4 pieces and process these pieces simultaneously. This can be easily done by making the program multi-threaded.

We have written such an implementation using the `pthread` library, as is very common on UNIX platforms. There are several advantages to this kind of implementation with regards to testing: all processing happens locally, there is no network traffic involved; and `pthread`-based programs are very easy to debug compared to programs that run on clusters. The code written using `pthread` has been used as a test bed for testing the algorithm for correctness before we moved on to running this code on clusters.

Another advantage of doing all processing locally is that the implementation of *unique count* has direct memory access to all tables to be processed. It is directly clear that this gives us a speed advantage.

Multi-core processors give us a lot of processing power, but often the memory available is still limited. Machines that have the required ±30 gigabytes of main memory installed are rare. By using a super computer cluster, we get the required amount of memory and even more processing power at our disposal. An important difference with multi-core computers is that the processors do not all have access to the same memory area, each node has its own main memory. During *unique count* data will have to be exchanged over a local network.

Cluster nodes have to communicate with each other, to announce how the work will be divided between the nodes and to share the results obtained by each node. Communication happens by passing messages. Standards have been developed for exchanging information between cluster nodes. One of the most used is MPI [1], which stands for "Message Passing Interface". MPI defines a series of library calls that can be used for sending data to other cluster nodes. There are calls like the standard `send`, `receive` and `broadcast` calls, but also calls more aimed at usage in a distributed setting such as `gather`. This call will collect an equally sized amount of data from each cluster node and merge all of these into a single array that will be delivered to the receiving cluster node.

The MPI library calls are all strictly defined. Several implementations exist of this set of calls. There is also a simple shared memory implementation of the MPI library, which (again like the `pthread`-based version described above) allows for local processing and thus easy debugging. On clusters, high-end implementations are used, making use of very fast Myrinet network interfaces commonly available on super computer clusters. We have been testing our code on clusters that indeed have Myrinet networking available.

In general, it is known that when writing multi-threaded programs the amount of locking and synchronization has to be brought to the absolute minimum. Using too much locks can be detrimental to the performance of the program, because many processor cycles will be spent waiting to acquire a lock. This is also known as "lock contention". For clusters the same holds for all communication that has to happen between nodes. The only communication that has to happen for the algorithm listed above is dividing the work between the nodes and the final unique counting phase. During development it appeared that this unique counting phase is

indeed critical and causes many performance problems, more on this in Chapter 3.

Modern clusters actually consist of multiple nodes that again have dual or quad core processors built in. There is no need to build a threaded version of the cluster application, such that each cluster node will run a single instance of the application and this application will spawn several threads to occupy all processor cores available on the cluster node. Instead we can just run more instances of the same MPI program on a single cluster node. If we use memory mapping to load the file containing the genome information into main memory, then the same memory pages with this information will be shared amongst the several MPI processes running on the same cluster node.

We have designed the structure of the source code in such a way that most of the code will be shared between the threaded and MPI-based applications. The only code that differs is the code that deals with communication. Obviously the threaded version uses shared memory communication and for the MPI version communication is done through the MPI library calls.

2.4 Development of the resulting application

A distributed version of the algorithm outlined above that runs on super computer clusters has been incrementally developed. We have outlined in the previous section that both the `pthread` and MPI versions of the code take advantage of memory mapping to load the genome into memory only once and share it between the calculation processes. The original code as written by Laros [4] uses another method to load the data from file (see also Section 2.6). The first step that we took was to write code to memory map the binary file containing the DNA data into memory and properly read and write the results to and from this file. This code has been abstracted away into `libdnabin` so that it can be easily re-used in multiple programs.

This library has been tested first in a simple single core implementation of the algorithm outlined above. Once this version worked and gave correct results we adapted it to use threads using the `pthread` library. A few first timings have been done on a quad-core computer. This also allowed us to make a few optimizations.

When the `pthread` version was feature complete and well tested, it was easy to turn this into a program that was suitable for usage on super computer clusters. To start with, only the code portions that handled communication had to be replaced. However, this was not enough to achieve proper performance. In the next chapters we will discuss the details of these distributed implementations of the algorithms and show which changes we have made to improve the performance.

2.5 Hardware used for research

We have developed our cluster code on the Distributed ASCI Supercomputer (DAS) 2 cluster available at Leiden Institute for Advanced Compuer Science (LIACS). On the DAS3 cluster sites at both LIACS and the Computer Science department of the Vrije Universiteit (VU) in Amsterdam we have conducted the experiments. The DAS3 system is a distributed supercomputer with clusters at five sites. Next to LIACS and the VU, these sites are located at the University of Amsterdam, Delft University of Technology and the MultimediaN Consortium. The cluster sites are interconnected with a very high speed network. Each cluster has a different configuration, for the two clusters we have used for the experiments the configurations

are listed in Table 1.

DAS3 is used for research on parallel, distributed and grid computing. Any student or staff member at the participating universities can get access to the supercomputer. On the DAS3 website (`http://www.cs.vu.nl/das3/`) more information can be found about the different cluster sites, research and procedures for requesting access to the machine.

In the next chapters, comparisons will be made between the performance of the same algorithm using a `pthread`-based configuration on a multi-core machine and MPI-based configurations on both multi-core machines and clusters. To allow for a fair comparison of these results, we will list the hardware that was used for these measurements in Table 1. All machines were running the Linux operating system.

Multi-core machine	Intel Core Quad (Q6600) at 2.4Ghz 4Gb main memory
DAS3 at LIACS	32 available nodes 2 AMD Opteron at 2.6Ghz 4Gb main memory Myri-10G interconnect
DAS3 at VU	85 available nodes 2 Dual-core AMD Opteron at 2.2Ghz (4 cores total) 4Gb main memory Myri-10G interconnect

Table 1: Concise overview of the hardware configuration of our testing equipment

2.6 Data set used for research

As test data we have used the human genome provided by the University of California in Santa Cruz [8]. The data is stored per-chromosome in FASTA format, which is a plain ASCII listing of the data. Laros [4] describes a binary file format in which both the genome data is saved and the occurrence count (once, twice, three times, or four times or more). By binary encoding this information the file size is decreased by 50%. Because we memory map these files directly into memory, this also saves 50% of main memory.

We have re-used the `comb` tool written by Laros for converting FASTA files into binary files. Using `libdnabin` we have written our own version of the `statistics` tool that outputs the sum of substring counts for verification purposes.

3 Splitting using the index table

In this chapter we will look at the first of the two distributed approaches that have been attempted. This first approach closely resembles what has been outlined in Section 2.1. We will discuss some of the details of the implementation and the problems that arose during the development. Solutions for these problems and ideas for future research will be given.

3.1 Details of the algorithm

Intuitively, the algorithm is quite simple. The work has to be divided and distributed to the available nodes. The number of available nodes as well as the sizes of the different "chunks" of work are completely arbitrary. Note that the amount of DNA to process does not have to be the same for each node, however, the more equalized these amounts are the more efficient the calculation will be performed as there will be less time spent by nodes waiting for other nodes to finish. A merge sort and *unique count* follows the sorting of the offset arrays. In this section we will describe how the work is divided amongst the nodes and how the merge sort and *unique count* are performed. Recall from the previous chapter that these are the only two tasks where communication between the nodes is involved.

3.1.1 Splitting up the work

Sequential DNA data contains "gaps" (for various biological reasons) that do not contain any data that is useful to process. In the binary files that we use for processing, these gaps are not included to save space. The offsets of the substrings in the DNA should still match with the DNA data that does contain the gaps. To resolve this, the binary file contains an index table that contains offset and length information about the DNA fragments in the binary file that are suitable for processing.

We have used the separation, in the form of gaps, between the data chunks as points to split up the work. Using the offset and length information from the index table and the requested substring length, we can calculate how many substrings can be found in each of the fragments. The first step is to calculate the total amount of substrings that are to be found in the entire data set to process. With the given number of nodes that are available for calculation, we can determine the average amount of substrings each node has to process with a simple division.

The DNA fragments that are in the binary files are all different in length. Some fragments are very small, others are very large. In order to equalize the amount of work between all nodes, we will try to create sets of fragments in such a way that the number of substrings in each set is about the same. The target number of substrings for each set is of course the average number of substrings to be processed by each node.

We will create those sets with a simple loop iterating over the index table. The number of sets to create equals the number of nodes available for computation. (Note that we are not considering the reverse complement here yet, this will follow later). Now we iterate over the fragments in the index table and continue adding fragments to this set until it contains 65% or more of the average number of substrings per node. Then we will start filling the next set of fragments. The 65% figure has been chosen empirically by looking at how equal the number of substrings to process is between the different sets. One drawback of this approach is that sets can only be filled with subsequent fragments. An algorithm that considers all fragments and

creates sets from that would be more complex but should result in combinations of fragments that have more equal sizes.

For the reverse complement, the number of substrings per fragment are exactly the same. The fragments for the reverse complements can be handled separately from the other fragments. We can create the sets for the reverse complements by simply cloning the respective sets for the normal case. This means we will end up with exactly twice the amount of sets. So when creating the sets we will not use the full number of nodes that are available but half of that. Combined with the sets for the reverse complement there will be a work set of fragments for each node.

In case the super computer cluster has dual core nodes, it is an interesting optimization to make sure that both the normal and reverse complement work sets that address the same fragments run on the same dual core node. Then both processes will work on the same piece of the genome and this piece can be shared in memory and only has to be read from disk once.

A few words on what exactly specifies the number of available nodes is in order. For the `pthread` version, the number of threads to spawn can be specified as a command line argument. The most efficient is to account for a single thread per available processor core. Alternatively, we could change the code to automatically detect the number of available processor cores and set that as the number of threads to create. In the MPI version it depends on the cluster environment how the number of nodes is specified. If one starts the MPI program directly from the command line (typically done using `mpirun`, see the documentation of the implementation of MPI of your choice) it can be specified on the command line. When using a job submission system (like DAS3), you have to specify the amount of slots (nodes) to reserve for your job. In case each node contains multiple processor cores, you will have the number of reserved slots times the number of processor cores per node available. Our program code uses the appropriate MPI function calls to determine the amount of nodes available at runtime.

Once the work sets have been created they can be handed out to the different nodes. Each node will then, in parallel, create a suffix array that contains offsets of all substrings that are contained within the set of fragments. These suffix arrays are sorted based on the comparisons between the substrings pointed to by the offsets in the array as explained in the previous chapter.

3.1.2 Combining the work

Finally, the fragments of the sorted suffix array that are spread out over the different nodes have to be merged into a single suffix array. In this suffix array we can easily count the number of times a substring occurs in the entire genome. These count values are written back to the binary genome file.

The algorithm that is used for this is a distributed variant of the *unique count* algorithm as outlined in Section 2.2. A master-slave model is used for executing this algorithm. One node will be designated as master and do the actual merge of the separate arrays and comparing the consecutive smallest substrings for the unique counting. The `pthread` version of the code uses a shared memory model and thus a "master" thread has direct access to the suffix arrays of the other threads. No communication between nodes is needed here. Below, we will compare the performance of the distributed *unique count* algorithm implemented using MPI with the `pthread` version. The MPI code has been run using the shared memory back-end on exactly the same hardware as the `pthread` version. Because of the overhead of the MPI library and

simulation of the communication calls, the MPI version was expected to be marginally slower than the `pthread` version.

We use a *get_next_smallest* call that will ask each node for its current smallest entry from the suffix array. This will result in *#nodes* entries. The smallest of these values is chosen and of the respective node the pointer to the current smallest entry is incremented.

In the algorithm as listed in Section 2.2 the resulting count values are written to a *substring_count* array. Instead of creating a new full count array, the master node that does the counting will immediately write the count the to binary file through `libdnabin`.

The main bottleneck that puts an upper bound on the performance of this algorithm is the speed by which the master node can receive the smallest entries from the nodes and compare the consecutive smallest substrings. The communication between the nodes is crucial, as we have discussed in Section 2.3. Several different approaches to exchanging information have been investigated in order to bring performance to an acceptable level.

One of the first attempts that has been made, transferred the smallest entries one at the time. We used MPI's `gather` primitive to transfer these values. The `gather` primitive works by asking each node simultaneously for a value of the same type, these will be "gathered" and returned at the same time in an array (with size equal to the number of nodes taking part in the communication) to a given node. In our case this given node is our designated master node. As a standard MPI practice, most often node number 0 is used as master node. The number of the node that delivered the smallest substring will be notified via broadcasting. This way the respective node can increment the pointer to its current smallest entry and all other nodes know that a smallest has been chosen. All nodes will then send their current smallest entries. The `gather` and `bcast` calls are both also a kind of synchronization primitive and thus have to be called in exactly the same order on all nodes to avoid deadlocks.

Immediately several disadvantages of this approach come to light. Firstly, a lot of values are sent over the network more than once. In a network of 32 nodes a single entry will be chosen as smallest, so all of the 31 other values will be submitted again. This is a waste of network resources. Next to that the latency is too high to be able to handle a single item at the time. A lot of time will be spent waiting for other nodes to synchronize. As we scale up the network to include more nodes, more and more time will be lost this way. Even with a small number of nodes, the performance of this approach was nowhere near acceptable values.

A logical next step was to look into creating a "batched" version of this communication protocol, where the offsets of the smallest substrings would be sent to the master node in batches. Instead of gathering a single entry we will gather an array of entries. We denote the size of this array by *batch size*. The main node can process these local arrays of smallest entries (one array per node) without doing another `gather` call until one of the arrays is out of entries. As soon as this happens the arrays are thrown away and a new gather call is done. The broadcast of selected smallest entries still happens on the basis of one call per processed entry, causing the corresponding node to increment its pointer to the current smallest entry. When the new gather call is done, all nodes will return *batch size* elements from the sorted arrays starting at the locations of the current smallest substrings. We assume here that the worst case where all consecutive smallest entries have come from the array of the same node (and thus all other arrays have not been touched) does (almost) never happen for large batch sizes.

The expectation was that this protocol would bring down the amount of network traffic wasted by trying to re-use as much of transferred data as possible. It appeared that still much time was spent on communication. Also doing the broadcast of the selected smallest

values in a batched version resolved these problems to some extent. In the last incarnation of the protocol, a loop is executed that does a `gather` for *batch size* smallest entries, these are processed followed by a `bcast` of *batch size* numbers of selected nodes.

Increasing the batch size to larger numbers indeed decreased the time needed for communications and thus the runtime. This confirms that the batched approach does have a positive effect on the runtime. With small batch sizes (in the order of 30000) performance was still not up to par. The batch size has been increased to up values of 16 million. Further increasing the batch size did not yield a decrease in runtime, suggesting that the size of the batches was no longer the bottleneck.

Of the entire runtime of the algorithm, less than 10% was spent on quicksort with all remaining time being spent on merging the tables from the participating nodes. (On DAS3, testing with a portion of the entire genome, we have seen figures where 20 seconds out of 12,5 minutes were spent on quicksort, the remaining time on merging). The merging should be much less computationally expensive compared to the quicksort. The master node is on its own responsible for running the entire *unique count* algorithm with data submitted from all participating nodes. We suspect that this task is too large to be accomplished in reasonable time by a single node.

The solution for the unique count performance problems is to also distribute the unique counting over several nodes. There certainly are possibilities for this. However, because a different approach to splitting up the work, which is presented in the next chapter, was both easier to implement and more probable to yield an increase in performance, the implementation of a fully distributed merge and unique count algorithm was not pursued.

3.2 Future research

The algorithm that has been presented in this chapter has been shown to work and delivers correct results. As discussed in the last section the performance of the algorithm is still poor, mostly caused by the implementation of merge sort and *unique count* running on a single node. A single node seems unable to handle the massive network traffic involved and do the sheer amount of necessary comparisons for unique counting. Still, there are several possibilities for improving the performance of this algorithm that could be explored in future research.

By changing the *unique count* algorithm to distribute the work load over several nodes, we think that a big performance gain can be achieved. Let us distribute the unique count work load over 4 nodes, then one node will be in charge of handling all substrings that start with "a", and the three other nodes will handle "c", "g" and "t" respectively. All nodes that have a sorted suffix array can find out where in the table the substrings starting with the different nucleotides are located using a fast binary search. Simultaneously nodes will be submitting parts of their suffix array starting with the different nucleotides to the corresponding nodes. When the 4 nodes are done with the merge sort and *unique count* calculations, the 4 resulting arrays can simply be concatenated to form the final complete array.

It is trivial to see that this approach to merging and unique counting sorting can be expanded to use more nodes by enlarging the prefix. We can distribute the unique counting over 16 nodes by designating one node for all substrings starting with aa, ac, ag, and so on. For this method to work most efficiently, each processor node has to process a single prefix. This means that the number of nodes must be a power of 4, which will be a problem as soon as the prefix length becomes longer.

As more nodes are used for the *unique count* calculation, CPU power should no longer be the main problem. The network traffic will become more complex, because there will not be just communication with one master node. Designing a well performing communication protocol will be the main challenge of this research.

We have seen in Section 3.1.1 that the equalization of the work between the nodes is not optimal. This gives us an opportunity for a small performance improvement. A better algorithm can be designed that is not constrained to iterating over the fragments once and in a single direction and thereby decreasing the difference between the average number of substrings to process per node and the sum of the fragments found in a work set. The decreases in runtime achieved by this optimization will become more visible when the size of data to process becomes larger.

4 Splitting using substring prefixes

Although the distributed algorithm discussed in Chapter 3 is able to quickly result in separate sorted suffix arrays, we have not been able to merge these arrays in a reasonable amount of time. This chapter will present an alternative distributed approach. By using another property of the problem for splitting up the work, we are able to remove the problematic merge sort and unique counting stage. We will describe the algorithm and discuss its performance. Again, we will conclude with some recommendations for future research.

4.1 Details of the algorithm

The algorithm that we will describe in this chapter does not differ much from what was discussed in the previous chapter. Basically, both could be outlined as follows:

1. Split up the given genome in some way.
2. Each node fills an array with offsets to substrings in the work set it has received. This suffix array will then be sorted, yielding a suffix array of a particular portion of the genome.
3. The separate suffix arrays created by the nodes are all merged.

We will discuss the details of all of these three steps in turn.

4.1.1 Splitting up the work

Merge sorting arrays spread out over multiple machines proved to be a problem performance-wise. It was suggested to try to make this merge sort dependent on multiple nodes by splitting up the merge sort based on prefixes of substrings. Instead of doing this only for the final merge sorting phase, we could also do this right from the start. In other words, we will not split up the work by creating sets of fragments from the index table of the genome, but rather give each node the entire genome and a prefix of the substrings that should be extracted and processed. Basically this is a distributed version of the multiple passes method as developed by Laros [4].

This has serious implications for the memory usage of the algorithm. Whereas the previous approach only needed to keep parts of the genome in memory, this algorithm needs the entire genome. Because we are using a memory mapped approach for reading the genome from disk, we do not have to think about the best way to page parts of the genome in and out, the operating system will take care of this for us. There is no need to keep the entire genome in memory all the time, but it is the best option, otherwise we would be reading the same parts of the genome from the disk more than once.

Splitting up the work is done by choosing a prefix length, let this be $plength$. If we decide to process all prefixes simultaneously and using a single node for each prefix (we will later change this assumption, loosening the requirements), this gives $4^{plength}$ nodes to get the work done. The prefixes to pass to each node are easily generated from the given alphabet of nucleotides. This method does work well; considerable time is saved by the fact that we have eliminated the need for merge sort and the unique counting that can now be done in parallel. Still, we can improve this in two areas: the requirement of the amount of nodes available and the inequality of the size of the subset of substrings starting with the different prefixes.

For this splitting algorithm to work, the number of nodes available must be a power of 4. This is not a problem, until *plength* becomes larger than 3. A prefix length of 4 requires 256 nodes, this is not something that is widely available in super computer clusters. Doing calculations on very large numbers of nodes has additional problems as we will see in Section 4.2. There is no need for all work sets to be processed in parallel, we can also process the work sets with increasing prefix in sequential order and concatenate them. By changing the model of work distribution by not doing all calculations on the nodes simultaneously, but working on pieces of the problem in turn, we can drop the constraint on the number of nodes. This will enable us to take advantage of, for example, 64 nodes if they are available (which is not a power of 4), which will give a speed-up in runtime compared with 32 nodes.

Let $\#nodes$ be the number of available nodes that is chosen freely. Given this number, we want to choose *plength* such that

$$4^{plength} \geq \#nodes$$

We will say that this results in $4^{plength}$ work sets or "batches" that are to be processed by $\#nodes$ nodes. Batches are handed out in sequential order. As soon as a node has finished processing its batch, it will submit the results to the master node. The master node will then hand out a new batch to process (as long as there are still batches left). In the previous model, nodes that were already done processing data had to wait for all other nodes to finish before the data could be merged together. With this model a node that has finished its batch, will get another one to process. The utilization of the available CPU power is hereby improved.

To be able to run this algorithm, a separate node has to be reserved that does not take part in processing a part of the problem. Otherwise, a node that has finished its work set would block on the master finishing its work set. For determining the amount of batches to work with, we will subtract one from the given number of nodes to reserve a master node. The above equation can be solved using ${}^4log\ \#nodes$. In code, to avoid dealing with the `log` function call and floating pointing numbers, we will increment *plength* starting at $plength = 1$ until the above equation is satisfied.

In case the nodes have limited memory available, one could opt to increase the number of batches. A larger amount of batches means less data per batch and thus a lower memory requirement per batch. However, this decrease in memory complexity increases the time complexity because the entire genome has to be walked through for each batch. If the genome does not fit entirely into memory, it will have to be fetched from the disk again which is a very costly operation.

4.1.2 Work done in parallel

Every node will have to read the entire genome from the disk. With current disk technology, read speeds of about 40 megabytes per second can be easily achieved. Given 1,5 gigabyte of genome data, we can expect this to be read into memory in under 40 seconds. The memory mapped approach will also benefit multi-core cluster nodes, the genome has to be read from disk only once and all calculation processes running on that node will share the same memory pages with genome information.

What each node will have to do with the given genome and prefix is the following:

1. Walk over the entire genome and save the offsets of substrings that match the given prefix into an array. Both substrings in the normal strand and the reverse complement are considered.

2. Sort the array, yielding a truncated suffix array.

3. Iterate over the suffix array and count how often each substring occurs.

4. Submit all this data to the master node. The separate suffix arrays of the nodes can be directly concatenated, there is no additional processing by the master node required.

Next to the genome, the offsets of the substrings to consider and whether they are in the reverse complement or not have to be stored in the main memory. The amount of substrings in the human genome exceeds the maximum amount that can be stored in signed integer values (2^{31}), requiring unsigned integers to be used. Hence, we cannot use the sign bit of these integers to store the reverse complement status. We will use a separate byte array for storing this additional information. After we have sorted the array, we will count how often each substring occurs. This count will be stored in the same byte array as the reverse complement information is saved in. In total we need 5 bytes of storage for each substring to consider.

When sorting the array we need both the offset values from the array and the reverse complement status from the byte array. The default implementation of quicksort in the C library can only handle a single array as input. To get around this limitation we could have chosen to use a struct that packs an unsigned integer and a char, create an array of these structs and pass this to quicksort. Due to architectural constraints, this combination of variables in a struct will require each structure (and thus each array element) to span 8 bytes. Three bytes will be left unused, in our case the suffix array will have many indices so the amount of memory wasted would be substantial.

Instead we have implemented our own variant of quicksort that will sort the array (which requires the usage of the reverse-bit in the byte array) and at the same time also swap the values in the byte array so both stay synchronized. Default implementations of quicksort "from the textbook" do not deliver the performance required for real-world computational problems. (In fact, it does not even come close). We have adapted an implementation of quicksort that has been extracted from the GNU C Library [9]. The performance of this adaptation is comparable with the default `qsort` library call on Linux systems (which are generally using the GNU C Library).

4.1.3 Combining the work

Although the batches are handed out sequentially, the results can be merged in any order. This is because the DNA fragments covered by each batch are partitions of the entire genome. As soon as a node is done processing the given batch, it will signal the master node. The master node will respond and the results, that consist of both an array of substring offsets and an array of occurrence counts, will be streamed to the master node. Streaming the results again happens with a batched transfer, the size of the batch will mostly influence the efficiency of the network transfer and the size of the buffer required at the receiving end. The master node will simply write the given count for the given substring offset directly back to the binary file.

When the master node has received results for all work sets, the calculation is finished and the program will terminate.

4.2 Performance

The runtime required by the final algorithm to process the entire human genome comes close to what we had initially expected. Utilizing the entire capacity of the DAS3 cluster at LIACS (spawning 64 processes), we were able to finish computation within 27 minutes for substrings of length 15. For substrings of length 64 the computation took 29,5 minutes. The difference in requested substring length does not seem to have much influence on the runtime, confirming our expectation that the time complexity of the algorithm is constant in terms of substrings length, if the length of the genome is much smaller than the total number of possible substrings $4^{substring\_length}$.

During the different experiments we have conducted on different configurations of cluster nodes, we have found that performing calculations on larger amounts of nodes does not always result in a decreasing runtime. We have done an experiment using 240 processes on 60 quad-core nodes on the DAS3 cluster at the Vrije Universiteit. It took a very long time to spawn all processes, some processes were started while others had already been running for 9 minutes. Based on this observation, we felt a need to get an idea of the differences between different configurations of nodes.

We have done several experiments with different configurations on the DAS3 cluster at both LIACS and the Vrije Universiteit. For these experiments we have only used a part of the genome so that we did not have to continously exceed the maximum allowed runtime of 15 minutes at the DAS3 cluster.

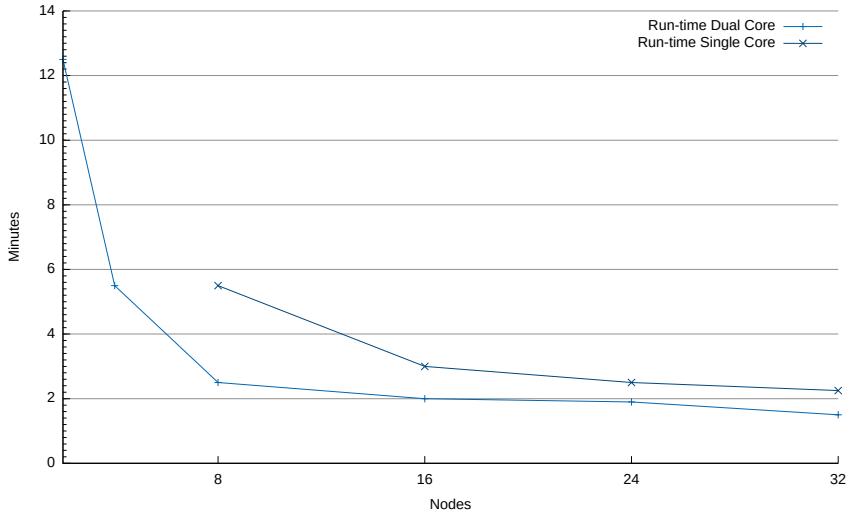

Figure 1: Run-time needed to process the first human chromosome on single-core and dual-core configurations on the DAS3 cluster at LIACS.

On the cluster located at LIACS we have done tests with only the first human chromosome. In the compressed format, as devised by Laros [4], the size of this chromosome is 108 megabytes. The results are depicted in Figure 1. When running on dual-core nodes (to get the total amount of processes used, multiply the number of nodes by two) we see a very sharp decline in runtime as more nodes are added. However starting at 16 nodes this decline flattens. We see a likewise development for the runs with single-core nodes. For 8 nodes, the dual-core run

is indeed about twice as fast as the single-core run. It is likely that the graph flattens out for many nodes as the cost of initialization for the calculation are greater than the cost of doing the actual calculation.

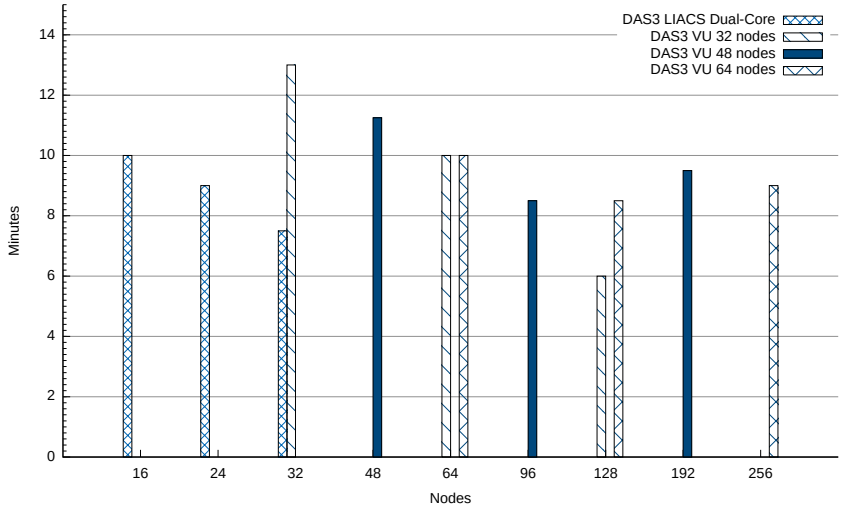

Figure 2: Run-time needed to process the 10th up to the 19th human chromosomes on various configurations at LIACS and the Vrije Universiteit.

For greater amounts of nodes we have also conducted tests with a larger part of the genome. This data set contains the 10th up to and including the 19th human chromosomes, with a size of 767 megabytes in compressed form. Figure 2 shows the results. The tests have been run at the LIACS cluster site on 16, 24 and 32 dual-core nodes. From the results for 16 and 32 nodes we observe that using twice the amount of nodes does not give us a 200% speed-up.

On the cluster located at the Vrije Universiteit, we have ran the tests on 32, 48 and 64 nodes and in a single, dual and quad core configuration. We can compare the result obtained at LIACS result for 32 dual-core nodes with the result obtained at the VU for 32 dual-core nodes and 64 single-core nodes. Both results obtained at the VU cluster are, interestingly, the same and indicate that these cluster configurations need more time to finish the calculation due to the lower clock frequency of the processors. When testing with 32 VU nodes increasing the number of cores used gives a sharp decrease in runtime required. In addition to using more processor cores for the calculation we think that re-using the genome data amongst the different processes on the same node also brings some benefit.

For 48 and 64 VU nodes, the results are different. In this case we observe that running in a quad-core setting is slower than with the same amount of nodes in a dual-core setting, even though the quad-core case has twice the amount of processing power available. As we have said above, we believe this is due to the amount of overhead involved in setting up such huge numbers of nodes.

Also the outcome for 128 nodes is interesting. Using 32 quad-core nodes, the calculation is considerably faster than by using 64 dual-core nodes. An explanation for this could be that the 64 node configuration can only share the genome data read from disk twice, the 32 node configuration can share the same data four times. This is however not supported by the measurements with 64 nodes; one would expect 64 single-core nodes, that have more I/O

overhead, to be slower than 32 dual-core nodes.

4.3 Future research

The method described in this chapter has proven to work very well. It scales well when the number of available nodes increases, but up to a certain amount. At some point the overhead of starting yet another node becomes higher than performing the calculation with less nodes.

Currently, we put the genome to process on the file server of the cluster. This means that it has to be transferred to each node over the network. There are possibilities to put the file to process on the local hard drives of all nodes. Because the DAS3 nodes are interconnected by gigabit networking, which can transfer data faster than it can be read from hard drives, we do not expect this to make a very large difference. The same holds for trying to replace writing back to the file via the network via local writes.

aa	42825917	ca	32751759	ga	26913119	ta	28586316
ac	22621931	cc	24507557	gc	19877632	tc	26913102
ag	32097435	cg	4563388	gg	24507506	tg	32751801
at	33531818	ct	32097507	gt	22621918	tt	42825818

Table 2: Number of substrings to process found in the first human chromosome for prefixes of length 2

We have seen that batches can be processed out of order without problems. The size of the different batches varies quite a lot, as can be seen in table 2. It can happen that the node that finishes its first work set last gets assigned another work set that happens to be the last available and very large. The computation won't be finished until this last large batch has also been processed. Instead of handing out the batches sequentially, we could look into counting the number of substrings for each batch first. This would require the master node to make a full pass over the genome, before work is handed out. With knowledge of the size of each batch, we can hand them out in such a way that work is equalized. Situations like the one described will not happen and the runtime might be improved again.

5 Conclusions

We have developed and analyzed two approaches for extracting and counting substrings from DNA data. Both methods operate by going through 3 stages: analyzing and distributing the work, executing as much of the work as possible in parallel, and finally combining the work. For both methods the space and time complexity only depends on the size of the genome to process and is constant for changes in the requested substring length.

The first method works by splitting up the DNA data by making use of the gaps in the data. The resulting suffix arrays from these fragments were created in parallel. At the end, the separated suffix arrays had to be merged together using merge sort. This proved to be a major bottleneck. As a recommendation for future development we suggested to look into distributing the work load of this merge sort.

Alternatively, we created a distributed version of the work done by Laros [4]. We split up the DNA data by giving each node the entire genome and a prefix. The node extracts and sorts all substrings from the genome that match the given prefix. This results in suffix arrays that are partitions of the genome and can therefore be easily concatenated. No merge sort is necessary and we can count how often each substring occurs in the separate suffix arrays in parallel. Measurements on DAS3 have shown this algorithm to finish processing the entire human genome in reasonable time: in about 30 minutes.

References

[1] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. `http://www.mpi-forum.org/docs/docs.html`, November 2003.

[2] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University press, 1997.

[3] Dong Kyue Kim and Kunsoo Park Jeong Soep Sim, Heejin Park. Linear-time Construction of Suffix Arrays. In *Combinatorial Pattern Matching*, volume 2676 of *Lecture Notes in Computer Science*, pages 186–188. Springer Berlin / Heidelberg, 2003.

[4] Jeroen F. J. Laros. *Unique factors in the human genome*. Master's thesis, Leiden University, May 2005.

[5] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, October 1993.

[6] Edward M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976.

[7] Joong Chae Na and Kunsoo Park Alberto Apostolico, Costas S. Iliopoulos. Truncated suffix trees and their application to data compression. *Theoretical Computer Science*, 304(1-3):87–101, 2003.

[8] University of California Santa Cruz. UCSC Genome Browser, Human Genome. `http://hgdownload.cse.ucsc.edu/downloads.html#human`, November 2008.

[9] Michael Tokarev. Inline QSORT() implementation. `http://www.corpit.ru/mjt/qsort.html`, January 2009.

[10] Esko Ukkonen. Online construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[11] Klaske van Vuurden. *Searching for unique strings of variable length in DNA using pruned and truncated suffix trees*. Master's thesis, Leiden University, August 2007.

A Description of programs written

In this appendix we will briefly describe the programs that have been written for this research and how these should be used. Both take a binary file with genome data and will output the results in the same file. They differ in how they use multiple processing units; one uses threads and the other one MPI communication.

A.1 Pthread-based version

The `pthread`-based version of the software is meant to be run on computers with single or multiple processors that share the same memory. This program does not work by handing out work sets sequentially as described in Section 4.1.1. Currently the prefix length has to be chosen to match the number of available cores in the system as close as possible. For example, this makes a prefix length of 1 a good choice for quad core systems, as 4^1 cores are needed for this operation. In the future the program could be improved to allow for the computation of work sets sequentially, like its MPI counterpart.

Further command line options that are supported are:

```
usage: ./process-pthread [-i file] [-p length] [-l length] [-a directory]
[-v] [-s]

        -i file to process
        -p prefix length to use for work distribution (default=1)
        -l length of substrings to be extracted (default=15)
        -a output suffix array files into directory
        -v verbose mode
        -s exit after determining split points (implies -v)
```

The option `-i` specifies a binary genome file to process. This option is always used, together with the `-p` option for the prefix length and the `-l` option for the length of substrings to extract. `-v` will enable display of more information, useful for debugging. Another debugging option is `-s`, which will make the program exit as soon as the work sets have been created, useful for debugging the creation of work sets without having to wait for the entire computation to finish.

Finally the `-a` option allows for outputting the suffix arrays created during the computation to files. This is done after the separate suffix arrays have been created, but before they are merged and unique counted. For each work set a separate file is created. The file names encode the prefix and the substring length of the data that the file contains, for example `prefix-ac-length-0064.out`. These files are all placed in the directory given as argument to the `-a` command line option. If this directory does not yet exist, it will be created. The file format of these output files is simple:

- (4 bytes) number of array elements in this file; say n.
- (n times 4 bytes) unsigned int values encoding offsets of substrings into the DNA data of the binary genome file that has been processed.
- (n times 1 byte) char values encoding whether or not the offset points to a substring in the reverse complement. A value of zero means the offset points to the normal strand, a value of one to the reverse complement strand.

A.2 MPI-based version

The MPI counterpart supports less options, all three work exactly the same way as described above:

```
-i file to process
-l length of substrings to be extracted (default=15)
-a output suffix array files in directory
```

During the experiments, we have placed the binary file to process on a file system that is shared between all cluster nodes. Alternatively, this file could be placed on the local hard drives of all nodes, as long as the path to the file is the same on each node. The same holds for the `-a` option.

Due to the usage of a different work distribution algorithm, as described in Section 4.1.1, the MPI version lacks an option for specifying the prefix length. The program will determine an appropriate prefix length itself, based on the number of available nodes.

This program should be run through the `mpirun` command or through specific start-up scripts of your cluster environment. Please refer to the manual of your local installation of MPI and/or your cluster site for more information.