Edit distance on GPU clusters using MPI

Antoine Veenstra
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
a.j.veenstra@student.utwente.nl

ABSTRACT

In this paper, we describe a verified implementation of the Levenshtein distance problem on a GPU cluster using MPI and OpenCL. The implementation is based on an existing verified single GPU implementation. The speed of the implementation is higher on a cluster, but the efficiency is affected by the overhead, which is caused by the extra communication between nodes.

Keywords

OpenCL, MPI, Message Passing Interface, C++, Edit distance problem, Levenshtein distance problem, GPU cluster, GPGPU program, case study

1. INTRODUCTION

In recent history the amount of available data has increased, as faster computers acquire more information at a higher rate. As the amount of data increases the need for parallel computation does so too to process said data. This, however, is no small feat. Multiple ways exist to process data in parallel, but one of the most efficient ways to do this is to use the Graphical Processing Unit (GPU). A GPU enables the parallel execution of a single operation on multiple variables, unlike the Common Processing Unit (CPU) which only allows for the execution of a single operation on a single value. Even if the CPU has multiple cores, the maximum amount of data processed in parallel is generally still inferior to that of a GPU.

To decrease the processing time even further a logical step is to increase the number of GPUs [4]. One could add more GPUs to their computer, but this is not a scalable solution since most motherboards only support a limited amount of GPUs. Another solution would be to make multiple devices work together, each containing at least one GPU. This is called a GPU cluster.

Various algorithms have already been implemented on a single GPU device and were verified [3]. The verification of a program is important since it can guarantee the outcome of an algorithm is always correct. If one wants to distribute a verified implementation over multiple nodes, additional steps have to be taken to ensure the implementation is still mathematically correct. Those steps are explored while distributing a verified implementation of the edit distance problem, which has been developed by De Heus [3].

The edit distance problem is used in various fields of research [8], such as Computational Biology, Signal Processing, and Text Retrieval. It is used to compare two strings or sequences of data, such as genome sequences. Section 3.3 will further discuss the edit distance problem.

The existing implementation of the edit distance problem uses a dynamic programming algorithm, which is well-suited for general-purpose computing on GPUs (GPGPU). The implementation was written in C++ using OpenCL, which can run on most GPUs [5]. An alternative for OpenCL would have been CUDA, which has been developed by NVIDIA and runs exclusively on NVIDIA GPUs. OpenCL has been chosen over CUDA in order to ensure compatibility with most GPUs.

To allow interaction between devices in a cluster a protocol is required. One standard which has been around for years is the Message Passing Interface (MPI) [10]. This interface has been used to distribute the algorithm on multiple (multi-)GPU nodes. The implementation in this paper is not the first combining MPI and OpenCL. One example is an implementation of the LINPACK benchmark, which used a cluster containing 49 nodes, each node containing two eight-core CPUs and four GPUs [4].

By implementing the edit distance problem on a GPU cluster instead of a single GPU, the processing time could be reduced as the performance of a cluster exceeds that of a single node [4]. The goal of this research is to implement the edit distance problem on a GPU cluster using MPI and to verify this implementation. This goal results in the research question mentioned in the following section.

2. RESEARCH QUESTIONS

The research question of this paper is:

What are the steps required to distribute a verified implementation of an algorithm on a GPU cluster?

The subquestions are:

1. How can the algorithm be divided in separate processes?
2. How can the algorithm be run on multiple devices using MPI?
3. How can the verification of the implementation be guaranteed?
4. What is the optimal number of GPUs when considering cost, efficiency, and the amount of data compared?

27^{th} *Twente Student Conference on IT,* July 7, 2017, Enschede, The Netherlands.

3. BACKGROUND

Before solving the research question, a background on the topics used to solve it will be given in this section.

3.1 OpenCL

GPGPU programming is the use of GPUs to handle computations which traditionally are done by CPUs. A CPU consists of one or more cores allowing Single Instructions streams and a Single Data stream (SISD). A GPU on the other hand has a Single Instruction stream and Multiple Data streams (SIMD). The number of cores on a GPU is generally much higher than a CPU has, so a GPU can process more data in parallel using its SIMD architecture.

One programming language allowing the developer to run programs on a GPU is OpenCL. OpenCL allows a developer to run a kernel on a GPU or CPU [6]. It is a low level programming language which can run on most GPUs and CPUs and allows general purpose parallel programming across both CPUs and GPUs. The traditional CPU based programming models do not allow the same complex vector operations on GPUs as OpenCL offers without the need to translate their algorithms to a 3D graphics API such as OpenGL. As mentioned before, OpenCL is preferred over CUDA since the support of CUDA for GPUs and CPUs is limited to NVIDIA GPUs [9].

In the OpenCL architecture one CPU based program called the *Host* controls multiple GPUs and CPUs called *Compute Devices*. Each of those *Compute Devices* consists of one or more *work-groups*, of which each contains one or more *work-items*. These *work-groups* execute the OpenCL kernels provided by the host program. Only before and after such kernel is running will the memory of the GPU be accessible to the *Host*. Each *work-item* has a unique identifier (id) to allow for different results, even though the same kernel runs on every *work-item* [6].

3.2 MPI

MPI is a standard specification for communication between computers which enables parallel computing. It is comparable to the traditional forking of threads in C and its derivatives, but it adds additional communication and computation functions. Nodes can send and receive messages both asynchronous and synchronous, read and write memory on other nodes, read and write files on other nodes, compute simple mathematical operations on variables available on each node, and much more [10]. Each node runs the same program and has its own unique id, which is useful while dividing the workload among nodes.

3.3 Edit distance

The edit distance problem is way of measuring how much two strings differ from each other [8]. The distance is measured by the minimal number of operations like inserting, removing, replacing, and rearranging characters. One use for the distance is to find DNA or RNA subsequences after possible mutations. Another use is to offer suggestions for misspelled words, as words with a shorter distance from the misspelled are more likely to be a correct replacement.

The complexity of the algorithm depends on what operations are allowed and the cost of these operations in the implementation. In the existing implementation by De Heus [3] and in the new implementation only inserting, removing, and replacing are considered. The costs of all the operations is set to 1. The edit distance with those conditions is also called the Levenshtein distance [8].

For example, take sequence a (s_a) as "kitten" and sequence b (s_b) as "sitting". The distance with the given conditions is equal to 3, since there are only 3 operations required to get from one sequence to the other. The operations required are:

- Replace the "k" with an "s". "kitten" → "sitten"
- Replace the "e" with an "i". "sitten" → "sittin"
- Insert "g" at the end. "sittin" → "sitting"

The order of the operations is not important, as long as it is the least amount of operations.

As mentioned before, the edit distance problem has already been implemented on a single GPU by De Heus [3]. His implementation is used as base in Section 4.1. As explained in Section 7.1, the performance of the new implementation will not be compared to the performance of this single GPU implementation.

3.4 Verification

Verification is done by describing what an algorithm requires as input and what it ensures as output. The language used is JML with an extension for Permission-based separation logic. The Permission-based separation logic is used to guarantee no read and writes of memory occur at the same time [1]. The separation logic uses simple rules to define permissions on resources like locks do in multithreaded programs, but do not have any effect on the actual program. It is used to verify multithreaded programs, to guarantee no concurrent resource access occurs.

Permissions are claimed by using the $\textsc{Perm}(x, \pi)$ function, where x is the memory address or memory range to claim and π is the permissions required. The value of π should be larger than 0 and no larger than 100. Any value between 0 and 100 grants read access to x, and a value of 100 grants write access. In this paper π is either *read* or *write*, where *read* is an arbitrary value between 0 and 100 and *write* equals 100. These two values should suffice, as no complex constructions are required in the verification. Multiple work-items can share a resource with the *read* permission, but only one can use a resource with the *write* permission.

4. DIVIDING THE ALGORITHM

To answer this question we must first explore the limitations and potential improvements of the previous implementation. The limitations will then be discussed and solved if possible in the following subsections. In the final subsection the algorithms used will be described.

4.1 Original algorithm

The algorithm of De Heus uses a dynamic programming solution [3]. In his paper he describes a way to distribute the computation on multiple work-groups of a GPU. The dynamic programming algorithm fills a matrix with the following rules [2]:

$$
\begin{aligned}
H_{(-1,j)} &= j \\
H_{(i,-1)} &= i \\
H_{(i,j)} &= \min \begin{cases} \mathrm{H}_{(i-1,j)} + 1 \\ \mathrm{H}_{(i,j-1)} + 1 \\ \mathrm{H}_{(i-1,j-1)} + Score \end{cases}
\end{aligned} \tag{1}
$$

where $Score$ equals zero if the characters of the sequences at index i and j are equal; otherwise, $Score$ equals one.

The value of $H_{(i,j)}$ depends on the cells $H_{(i-1,j)}$, $H_{(i,j-1)}$, and $H_{(i-1,j-1)}$. This limits the use of parallelism to speed

up the computation, but it leaves an opening nonetheless. There is no dependency between cells $H_{(a,b)}$ if $a+b$ is constant. The grey cells in Figure 1 are such a group of cells which can be calculated in parallel. Each diagonal is based on the previous two diagonals, because of the dependencies previously mentioned [7]. There is no need to save diagonals prior to those two diagonals, so the implementation can discard the previous diagonals to save memory.

4.2 Partitioning the algorithm

With larger sequences the diagonal becomes too large to calculate in one iteration on a GPU. Dividing the matrix vertically allows one to split the calculation in manageable parts. These parts will be called pillars. Each pillar requires the right most column of the previous pillar due to the dependencies of each cell. This means the other columns can be discarded to save memory.

Each of the pillars mentioned above can be split in blocks. Blocks A to D, E to H, and I to L in Figure 2 are such a partitioning. The dependencies of the individual cells are inherited by the individual blocks. Just like the cells the blocks can also be calculated in parallel if they are not dependent of one another. Block D and F are such blocks as they only require block B and C. With larger sequences the number of independent blocks becomes more significant. As a result, the calculation of multiple blocks in parallel becomes more attractive.

If the blocks where squares, as De Heus suggested [3], the amount of cells processed in parallel would start at 1, continue to *width*, and go back to 1. The amount of cells processed increases or decreases by 1 every iteration, so the average amount of cells processed is approximately $width/2$ cells processed in parallel. Blocks like B and K, on the other hand, have an average amount of cells processed in parallel of exactly *width*. The only disadvantage of this approach is that the top and bottom of each pillar alternate blocks like A and D exist, but the overall performance is still better. Therefore, the pillars will be constructed diagonally to optimise the amount of cells processed in parallel at any given time.

4.3 Storing the diagonals

As mentioned before, two diagonals are required to calculate the following diagonal. Unfortunately, OpenCL offers no support for an array of arrays, so either the two diagonals must be saved in separate variables or they must be combined in one larger array. Using two variables limits the freedom while implementing the algorithm as the rows must be swapped after each iteration. Using a single combined array requires the calculation of indices each time the algorithm accesses the array. The two solutions are not significantly different, so another factor should be considered before opting for one of the two solutions. The most common blocks are shaped like B, C, F, etc. Therefore, the rest of this section will only consider the algorithm required to process those blocks.

An input the algorithm for processing a block should digest is the left column and it should output the right most column of the block. The two diagonals should evolve, but the size should stay constant. In the two variables solution the column values could be divided between and appended to the two diagonals. This requires some preprocessing as the even indices of the column should be appended to one diagonal and the odd indices to the other as shown in Figure 3a. Some post-processing is also required to retrieve the column after processing as shown in Figure 3b. A solution to this problem is to transfer the parts of diagonals

		k	**i**	**t**	**t**	**e**	**n**
	0	**1**	**2**	**3**	**4**	**5**	**6**
s	**1**	1	2	3	4		
i	**2**	2	1	2			
t	**3**	3	2				
t	**4**	4					
i	**5**						
n	**6**						
g	**7**						

Figure 1. Example matrix

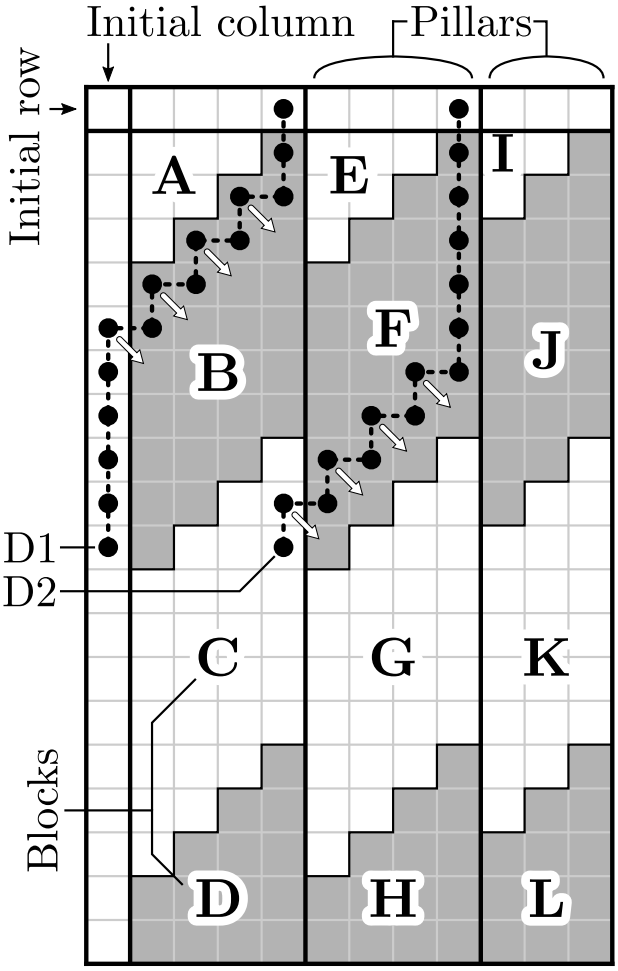
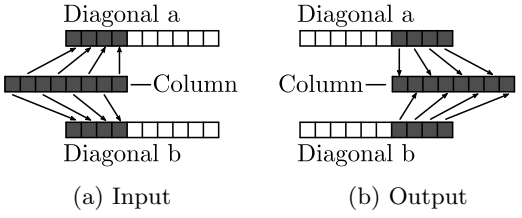


Figure 2. Partitioning of the matrix



(a) Input (b) Output

Figure 3. The two variable implementation

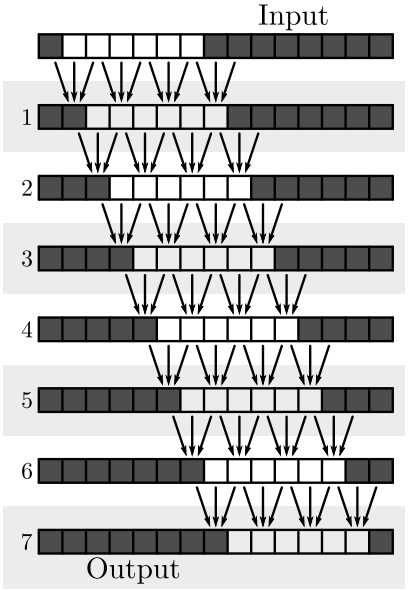


Figure 4. Computation with a single array

a and b as is. The pre- and post-processing steps cancel each other out, so there is no need to do so.

The digestion of the input and the supplying of output is comparable in the single array solution. Figure 4 shows the evolution of the single array throughout the computation of a block. The block used in this figure is block B from Figure 2, which is four columns wide and consists of seven iterations. There is no advantage or disadvantage on which end the input and output are stored. The blocks of memory can be transferred to another process as is, just as in the two variable solution.

We can conclude that there are no significant advantages or disadvantages to either solution. The single variable solution is only slightly more attractive since there is one less variable to worry about. Therefore, the single variable solution will be chosen in the algorithms mentioned in Section 4.6.

The calculation of indices depends on the iteration count of the algorithm. The formula used to get the index of the diagonal to change is $i + n \cdot 2 + 1$ where i is the number of iterations completed and n is the index of the cell in the diagonal. The index of the cell above the targeted cell is determined by subtracting 1 from the index of the targeted cell and the index of the cell to the left of the targeted cell is determined by adding 1 to the index of the targeted cell. This means that the cells used by a thread are three consecutive cells of which the middle one will be replaced with the calculated value. This can be seen in Figure 4, where three consecutive cells calculate the result of the middle cell. This figure also shows how the dependency on the iteration count influences the indices of the targeted cells.

A visual representation of which cells are stored in the array can be found twice in Figure 2 as $D1$ and $D2$. Lines $D1$ and $D2$ cross all the cells stored in the array at one point in the algorithm. The arrows go from the targeted cells to the cell for which the values are calculated during the next iteration. The arrows next to $D1$ represent the third iteration and the arrows next to $D2$ represent the last iteration. The top cell of the lines are stored on index zero and the bottom cell is stored in the last cell of the array. This direction has been chosen to make debugging

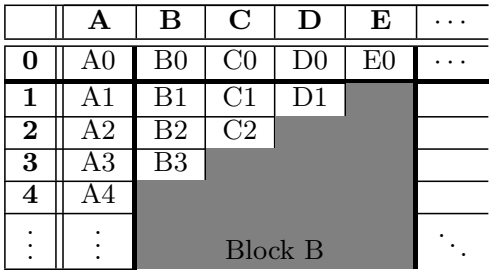


Figure 5. Block A zoomed in

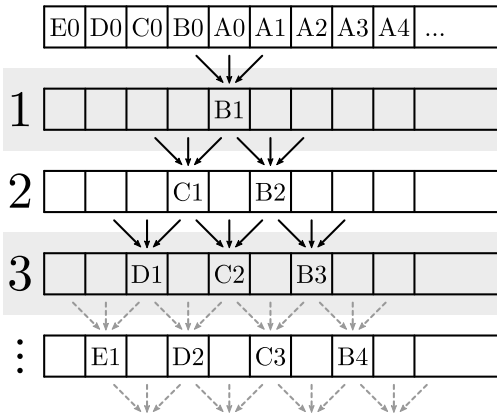


Figure 6. Computation of the starting block

easier as the resulting column will be in the first cells of the diagonal after the run, making them easier to find and compare to known results. There is no other advantage or disadvantage if the result is stored the other way around. Figure 4 also shows this same orientation. The input is equivalent to the left most column as seen in $D1$ and the output equivalent to the right most column in $D2$.

4.4 Constructing the initial diagonals

Now that the storage of the diagonals has been set the array has to be initiated at the beginning of the pillar. In Figure 5 an example of a starting block is shown which is a zoomed in version of block A in Figure 2, but also applies to block E. The width of the pillars is 4, so the block is a triangle of 3 by 3 cells.

In Figure 5 column **A** can be copied from the previous pillar and row **0** can be calculated by adding the offset of the pillar to the cell. In Figure 6 the array containing the diagonals and its transformation is shown. The coordinates in the Figure 6 correspond to the coordinates in Figure 5. Each iteration is numbered and the first iteration of the following block is shown at the bottom.

Figure 6 resembles to Figure 4. Both share the same property of having three consecutive cells defining the middle cell. The obvious difference being the amount of cell being processed each iteration. In the following section you will find the algorithm to compute this block.

4.5 Computation of the last block

At the bottom of every pillar there is a block with a flat base. In Figure 2 D, H, and L are such blocks. If the same method as in Figure 4 is used the array of diagonals will go out of bounds. However, this poses no problem as both the result and the output column are independent of the cells below it. Thus blocks D, H, and L will be treated

Algorithm 1 Parallel algorithm to process blocks

```
1: procedure BLOCK_CALC(id, width, s_a, s_b, height, d)
2:     kernel requires (\forall* t; 0 ≤ t < width; PERM(s_a[t], read))
3:     kernel requires (\forall* i; 0 ≤ i < width + height − 1; PERM(s_b[i], read))
4:     requires s_a ≠ NULL
5:     requires s_b ≠ NULL
6:     requires d ≠ NULL
7:     requires 0 ≤ id and id < width
8:     requires s_a.length ≥ width
9:     requires s_b.length ≥ width + height
10:    requires d.length ≥ width · 2 + height
11:    requires PERM(s_a[width − 1 − id], read)
12:    requires (\forall* i; id ≤ i < id + height; PERM(s_b[i], read))
13:    ensures d[0] = \OLD(d[0])
14:    ensures d[height+width·2−1] = \OLD(d[height+width · 2 − 1])

15:    a ← s_a[width − 1 − id]

16:    loop_invariant x = i + id · 2 + 1
17:    for i is 0 . . . height do
18:        requires i ≠ 0 ⇒ PERM(d[x − 2], read) ∗ PERM(d[x − 1], write) ∗ PERM(d[x], read)
19:        ensures PERM(d[x − 1], read) ∗ PERM(d[x], write) ∗ PERM(d[x + 1], read)
20:        kernel ensures (\forall* t; 0 ≤ t < width; PERM(d[i + t · 2], read) ∗ PERM(d[i + t · 2 + 1], write) ∗ PERM(d[i + t · 2 + 2], read))
21:        BARRIER()

22:        if a = s_b[id + i] then
23:            Score ← 0
24:        else
25:            Score ← 1
26:        end if
27:        ensures a = s_b[id + i] ⇔ Score = 0
28:        ensures a ≠ s_b[id + i] ⇔ Score = 1
29:        x ← i + id · 2 + 1
30:        cell_d ← d[x] + Score
31:        cell_up ← d[x − 1] + 1
32:        cell_left ← d[x + 1] + 1
33:        d[x] ←MINIMUM(cell_up, cell_left, cell_d)
34:        ensures d[x] = MINIMUM(MINIMUM(d[x − 1], d[x + 1]) + 1, d[x] + Score)
35:    end for
36: end procedure
```

the same way as blocks like B. The only difference is the number of iterations required to get the right most column of the block.

4.6 The OpenCL algorithms

Figure 2 shows three kinds of blocks. How blocks like A, E, and I are computed is explained in Section 4.4 and can be seen in Figure 6. How blocks like B, C, F, G, J, K, and E are computed is explained in Section 4.3 and can be seen in Figures 2 and 3. Blocks like D, H, and L are computed like block B as explained in Section 4.5, so they will not be treated differently. Algorithms 1 to 3 contain JML specifications in green, which will be discussed in Sections 6.1 to 6.3.

4.6.1 Algorithm to compute standard blocks

Algorithm 1 is used to handle blocks like B, and D. Variable d contains the diagonals as explained in Section 4.3. The id is the id of the thread. OpenCL allows multiple threads to execute the same algorithm in parallel. The id of the thread is a unique number from zero till the number of threads. This id is used to identify which column of the block the thread is to process. Variable $width$ is the width of the block. The width of the block is equal to the amount of threads OpenCL is using. Variable $height$ is the number of iterations to be executed. There is no limit to this number apart from the length of sequence b, as it is useless to compute more rows than present. Variables s_a and s_b are parts of sequences a and b respectively. The size of s_a is equal to the $width$. The size of s_b is equal to the total height of the block, so this is equal to $height + width - 1$.

As seen on line 15 of Algorithm 1 each thread picks a column according to the following formula: $width - 1 - id$. If the id had been used to pick a column the equations on lines 22 and 29 would be "$a = s_b[width - id + i]$" and "$x \leftarrow width - id + i$" respectively, so there is no significant difference.

Each iteration is separated by a BARRIER() operation on line 21. It guarantees that every thread is at the same point in the program before any can continue. As x is computed with the $id \cdot 2$, the xs at every iteration always differs with at least 2 between threads. This means that no thread can read a cell while another is writing in the same memory, thus guaranteeing that no memory inconsistencies occur during operation.

4.6.2 Algorithm of initial blocks

Algorithm 2 is used to handle blocks like A, E, and I. Variables id, $width$, s_a, s_b, and d are the same as in Algorithm 1. Variable $offset_a$ is the offset of the pillar relative to sequence a, where the first element of the sequence is zero.

Lines 17 to 20 fill the top row, which is row **0** in Figure 5. Since the row is one cell wider than the width, one thread has to set two cells. On line 21 the character of the column being computed is loaded. Each thread loads the character at index id. If the index would be $width - id - 1$ the equations on lines 30, 34, and 41 would be along the lines of "$width - id - 1 \leq i$", "$a = s_b[i - (width - id - 1)]$", and "$x \leftarrow i + width - (width - id - 1) \cdot 2$". These equations are longer and thus more susceptible to bugs.

Lines 23 to 48 are equivalent to lines 17 to 35 in Algorithm 1. The only difference between the two loops is that not all the threads calculate a new value in Algorithm 2 as can be seen in Figure 6. The if statement on line 30 ensures that.

4.6.3 Algorithm of initial column

There is only one column unaccounted for and that is the initial column as indicated in Figure 2. Algorithm 3 does that while taking into the account the structure of the variable containing the diagonals. The variables id, $width$, $height$, and d are the same as in Algorithms 1 and 2. The variable $offset_b$ is the offset of the block relative to sequence b, where the first element of the sequence is zero. Line 8 calculates the total size of the diagonal and line 9 calculates the starting index to fill. The while loop on Lines 11 to 14 fills each cell of the diagonal while respecting the offset of b. There are no BARRIER() statements in this algorithm as the threads only write to the memory.

4.6.4 Conclusion

With these three algorithms and the exchange of columns as described in Section 4.2 the edit distance between two

Algorithm 2 Parallel algorithm to begin pillars

```
1: procedure BEGIN_CALC(id, width, s_a, s_b, offset_a, d)
2:     kernel requires (\forall* t; 0 ≤ t < width; PERM(s_a[t], read))
3:     kernel requires (\forall* i; 0 ≤ i < width − 2; PERM(s_b[i], read))
4:     kernel requires (\forall* i; 0 ≤ i ≤ width; PERM(d[i], read))
5:     requires s_a ≠ NULL
6:     requires s_b ≠ NULL
7:     requires d ≠ NULL
8:     requires 0 ≤ id and id < width
9:     requires s_a.length ≥ width
10:    requires s_b.length ≥ width
11:    requires d.length ≥ width · 2
12:    requires PERM(s_a[id], read)
13:    requires PERM(d[id], write)
14:    requires id = 0 ⇒ PERM(d[width], write)
15:    ensures d[0] = \OLD(d[0])
16:    ensures d[height+width·2−1] = \OLD(d[height+width · 2 − 1])

17:    d[id] ← offset_a + width − id
18:    if id = 0 then
19:        d[width] ← offset_a
20:    end if
21:    a ← s_a[id]
22:    loop_invariant x = i + width − id · 2
23:    for i is 0 ... width − 1 do
24:        requires i = 0 ⇒ PERM(d[id], write)
25:        requires i = 0 and id = 0 ⇒ PERM(d[width], read)
26:        requires i ≠ 0 ⇒ PERM(d[x − 2], read) ∗ PERM(d[x − 1], write) ∗ PERM(d[x], read)
27:        ensures PERM(d[x − 1], read) ∗ PERM(d[x], write) ∗ PERM(d[x + 1], read)
28:        kernel ensures (\forall* t; 0 ≤ t ≤ i; PERM(d[i+width−id·2−1], read)∗PERM(d[i+width−id · 2], write) ∗ PERM(d[i + width − id · 2 + 1], read))
29:        BARRIER()
30:        if id ≤ i then
31:            requires PERM(d[x − 1], read)
32:            requires PERM(d[x + 1], read)
33:            requires PERM(d[x], write)

34:            if a = s_b[i − id] then
35:                Score ← 0
36:            else
37:                Score ← 1
38:            end if
39:            ensures a = s_b[i − id] ⇔ Score = 0
40:            ensures a ≠ s_b[i − id] ⇔ Score = 1
41:            x ← i + width − id · 2
42:            cell_d ← d[x] + Score
43:            cell_up ← d[x − 1] + 1
44:            cell_left ← d[x + 1] + 1
45:            d[x] ←MINIMUM(cell_up, cell_left, cell_d)
46:            ensures d[x] = MINIMUM(MINIMUM(d[x − 1], d[x + 1]) + 1, d[x] + Score)
47:        end if
48:    end for
49: end procedure
```

sequences of arbitrary sizes can be computed. The division in blocks as shown in Figure 2 allows for multiple process to work on the calculation in parallel. This answers the fist subquestion as stated in Section 2.

Algorithm 3 Parallel algorithm to fill the first column

```
1: procedure COLUMN_FILL(id, width, offset_b, height, d)
2:     kernel requires (\forall* x; width · 2 − 1 ≤ x < width + height − 1; PERM(d[x], write))
3:     requires d ≠ NULL
4:     requires 0 ≤ id and id < width
5:     requires d.length ≥ width · 2 + height
6:     requires (\forall* x; width·2−1+id ≤ x·width+id − 1 and x · width + id − 1 ≤ width + height − 1; PERM(d[x · width + id − 1], write))
7:     ensures (\forall x; width·2−1+id ≤ x·width+id − 1 and x · width + id − 1 ≤ width + height − 1; d[x · width + id − 1] = (x − 1) · width + id − 1 + offset_b)
8:     total_size ← width + height − 1
9:     i ← width · 2 − 1 + id

10:    loop_invariant (i − 1 + id) mod width = 0
11:    while i ≤ total_size do
12:        d[i] ← i − width + offset_b
13:        i ← i + width
14:    end while
15: end procedure
```

5. USING MPI

Now that the algorithm is split in manageable pillars in Section 4, the next step is to distribute the algorithm over a cluster. MPI offers various functions to simplify the distribution, but the main task will be exchanging the columns as explained in Section 4.2. How the tasks will be divided is the first problem that will be tackled.

5.1 Distributing the pillars

The distribution of the workload of the edit distance problem can be done by simply passing a few messages between nodes. Each node needs to know which pillar it has to compute itself and then give the next node a column number of where their pillar starts. The next node is the node with one id higher or, if no such node exists, the node with id zero. After a node has finished a pillar, it picks the next pillar that it has to compute itself by adding the sum of the widths of the nodes to the current column number. Section 4.6 explains what the width of a node is. The sum of those widths can easily be computed by using a cluster wide mathematical operation command available in MPI. Note that the distribution described here does not take into account the different speeds of GPUs. The impact of this deficiency is discussed in Section 7.

The height of the blocks can be negotiated between the nodes. The lowest width of the nodes is communicated to all nodes and they multiply that by an arbitrary constant. This constant should not be too small as the overhead of starting a run would impact the overall performance, but it should not be too big either, since not enough blocks will be available for parallel computation.

After the computation is complete, the node which has processed the last block should return the result. MPI shows the output of all the nodes, so there is no need to forward any results to a specific node.

5.2 The MPI algorithm

Algorithm 4 shows how the nodes work together. The initialisation and finalisation phases have been left out, since they are not important to the algorithm. Arguments of the procedure are queried in the initialisation phase. The algorithm contains JML specifications in green, which are discussed in Section 6.4.

Algorithm 4 MPI control

```
1: procedure MAIN(id, column, s_a_length, s_b_length, width, cluster_total_width, blocks_num, max_height)
2:     requires 0 ≤ id and id < width
3:     requires width ≤ cluster_total_width
4:     requires 0 ≤ column and 0 < s_a_length
5:     requires 0 < blocks_num and 0 < max_height
6:     requires (blocks_num − 1) · max_height < s_b_length
7:     requires s_b_length ≤ blocks_num · max_height

8:     loop_invariant column mod cluster_total_width = \OLD(column)
9:     while column < s_a_length do
10:        requires queue.IS_EMPTY()
11:        if column ≠ 0 then
12:            ensures not queue.IS_EMPTY()
13:            queue.PUT(MPI_RECEIVE_COLUMN())
14:        end if

15:        ensures s_a ≠ NULL
16:        s_a ← READ_PART(column, width)
17:        CL_WRITE_MEM(s_a)

18:        loop_invariant queue.SIZE() ≤ 2
19:        for block is 0 ... blocks_num do
20:            if column ≠ 0 and 1 < blocks_num − block
21:                    then
22:                ensures queue.SIZE() = 2
23:                queue.PUT(MPI_RECEIVE_COLUMN())
24:            end if

25:            ensures column_list ≠ NULL
26:            if column ≠ 0 then
27:                ensures 1 < blocks_num − block ⇒ queue.SIZE() = 1
28:                ensures 1 = blocks_num − block ⇒ queue.SIZE() = 0
29:                column_list ← queue.POP(id)
30:            else
31:                column_list ←EMPTY_COLUMN()
32:            end if

33:            ensures height = MIN(max_height, s_b_length − block · max_height)
34:            if 1 < blocks_num − block then
35:                height ← max_height
36:            else
37:                height ← s_b_length − block · max_height
38:            end if

39:            column_list ←EXECUTE_KERNEL(
40:                max_height, height, s_a, width, block,
41:                column_list, queue.PEEK()
42:            )

43:            if not IS_FINAL_PILLAR() then
44:                MPI_SEND_COLUMN(column_list)
45:            else if blocks_num − block = 1 then
46:                PRINT(column_list[height − 1])
47:            end if
48:        end for
49:        column ← column + cluster_total_width
50:    end while
51: end procedure

52: function EXECUTE_KERNEL(max_height, height, s_a, width, block, column_list, column_next)
53:    requires 0 < height ≤ max_height
54:    requires 0 < s_a.length ≤ width
55:    requires 0 < block
56:    requires column_list.length = height
57:    requires column_next.length ≤ max_height

58:    if block ≠ 0 then
59:        diagonal_size ← s_a.length · 2 − 1
60:        CL_COPY_MEMORY(from=max_height, to=0,
61:            size=diagonal_size)

62:        CL_WRITE_MEMORY(to=diagonal_size,
63:            from=column_list[s_a.length − 2 :])
64:        column_size ← max_height − s_a.length + 2
65:        CL_WRITE_MEMORY(to=diagonal_size+
66:            column_size, from=column_next)
67:    else
68:        CL_WRITE_MEMORY(to=s_a.length + 1,
69:            column_list)
70:        CL_WRITE_MEMORY(to=s_a.length + 1+
71:            max_height, column_next)
72:    end if
       ⋮
73:    return CL_READ_MEMORY(1, max_height)
74: end function
```

The *id* is the unique identifier of the process. The *column* contains the column index where the node has to start. During the while loop on lines 5 to 38 this variable will contain the column index of the pillar being processed. Variables s_a_length and s_b_length contain the lengths of sequences a and b. The *width* is the width as explained in Section 4.6. Variable *cluster_total_width* contains the sum of the width variables of all the nodes in the cluster. It is the maximum number of columns the GPUs in the cluster can handle at any given time. The *blocks_num* is the number of blocks in a pillar minus the starting block as that will be combined with the second block. For example, the *blocks_num* in Figure 2 equals 3. The starting block is not counted as its computation will be combined with the next block. The variable *max_height* is equal to the maximum number of iterations a block will have.

Block G in Figure 2 depends on blocks C, D, and F, so the process handling block G should get both most right columns of C and D before starting the calculation. Lines 11 to 14 retrieve the first column if the process is not handling the first pillar. Each iteration of the for loop retrieves another column in lines 20 to 24 before computing the block. This ensures that there are two columns in the queue before line 39 if the id of the process is not zero and if the process is not at the last block of the pillar. At the last block of a pillar no column is received, since the last block depends on only one other block. For example, in Figure 2 block H depends only on block D, given that the diagonals used in block G are still in the memory. The first pillar does not require the retrieval of

columns from other nodes, since Algorithm 3 can be used to fill its column.

Lines 16 and 17 read a chunk of sequence a starting from *column* and store it in the memory of the GPU. The size of this chunk is the number of characters read up to the *width*, so if there are not enough characters left in s_a the width of the chunk is equal to $s_a\_length - column$.

The for loop on lines 19 to 48 iterates through all the blocks of the pillar, where the first and second block are considered as one. For example, in Figure 2 blocks A and B are considered as one. Lines 20 to 38 retrieve the variables required to execute the kernel on line 39. Lines 43 to 47 handle the result of the kernel.

As mentioned earlier, the if statement in lines 20 to 24 manages the retrieval of columns from other nodes. Line 23 is executed if the process is not dealing with the first pillar and if it is not computing the final block of a pillar.

Lines 26 to 32 get a column if needed. As mentioned before, the first pillar does not require a column from other nodes, so an empty list is used instead for the first pillar.

Lines 34 to 38 get the height of the block which is to be processed. Only the height of the last block of each pillar is smaller than the *max_height*.

On lines 39 to 42 the kernel is executed and the result is saved. The function is defined on lines 52 to 74.

Lines 43 to 47 handle the output of the kernel. If the process is not at the last pillar the column will be send to the next node. Again, the next node is the node with one id higher or, if no such node exists, the node with id zero. If the process is at the last block of the last pillar, the result of the problem can be found in the column. This result is printed to the standard output. As mentioned before, MPI will handle the redirection of the output of a node to the output of the parent call.

5.2.1 Executing the kernel

The code on lines 52 to 74 shows the steps required to run the kernel. The kernel will execute a combination of algorithms which can be found in Section 4.6. In the first pillar Algorithm 3 is first executed. In the first block of any pillar Algorithm 2 is executed next. And at last Algorithm 1 is executed. Note that for the combination of blocks A and B of Figure 2, Algorithms 3, 2, and 1 will be executed.

Lines 58 to 72 show what memory operations on the GPU are required to run kernel. What memory operations are executed depend on which block is being processed. If it is not the first block, the array of the diagonals from the last run is moved to the start of the variable on line 60. Then the column retrieved from the previous node is written to the memory after the diagonals, where the column skips the first characters. This is done on line 62. The number of characters skipped of this column is equal to the size of the width minus two. After that, the following column is also appended in the memory on line 65. The copying of the second column is required because the block being processed depends on two columns.

If the process is at the first block of a pillar there is no need to copy the diagonals as the new block will be using Algorithm 2. Only the writing of the columns is necessary. This is done on lines 68 and 70.

To keep the performance of the program decent, the number of blocks in a column should be at least 4, so that multiple blocks can be computed in parallel on multiple nodes. There is however a lower limit on the height of a block because of line 60. If the diagonals at the end of a run overlap with the position where the diagonals are copied to, an error will be encountered. This can be solved by buffering the copy, but this brings too much overhead. As this only happens if the sequences are small, it is better to run the problem on a single node as Section 7 will point out. That is why the program will simply refuse computing the edit distance between two too short sequences on a cluster. It will not fall back to compute the problem on a single node to eliminate any confusion on how many nodes were actually used to solve the problem.

The formula to compute the minimum sequence size is $2 \cdot diagonal\_size$. The $diagonal\_size$ is equal to $s_a.length \cdot 2 - 1$ as line 59 shows. The width of a pillar on one node used for testing was 1024, so sequences run on a cluster including that node will refuse any sequence size smaller than $2 \cdot (1024 \cdot 2 - 1) = 4094$.

6. VERIFYING THE IMPLEMENTATION

In Algorithms 1 to 4 verification has been added in the form of *ensures* and *requires*. The verification guarantees that the work-items do not write to the same memory location. This is required to ensure the determinism of the algorithm, so that the result is consistent. As explained in Section 3.4, permission-based separation logic is used to state which work-item has access to what resources.

Most of the lines are trivial or explained in Section 4.6. Non-trivial lines will be explained in this section. The OpenCL algorithms are executed as a single group, so the work-group verification is equal to the kernel verification Therefore, the work-group verification has been left out.

6.1 OpenCL Algorithm 1

In Algorithm 1, lines 18 to 21 describe how the barrier distributes the read and write permissions among the work-items. Line 18 reclaims all permissions the work-items have on d. The following line redistributes the permissions, and line 20 describes how the permissions are distributed over the kernel. Since the difference of the xs between two consecutive threads is 2, no threads should have read permissions on a cell with write permissions of another work-item.

Lines 13 and 14 refer to the fact that the first and last cells in the array of diagonals are not edited, as Figure 4 illustrates.

6.2 OpenCL Algorithm 2

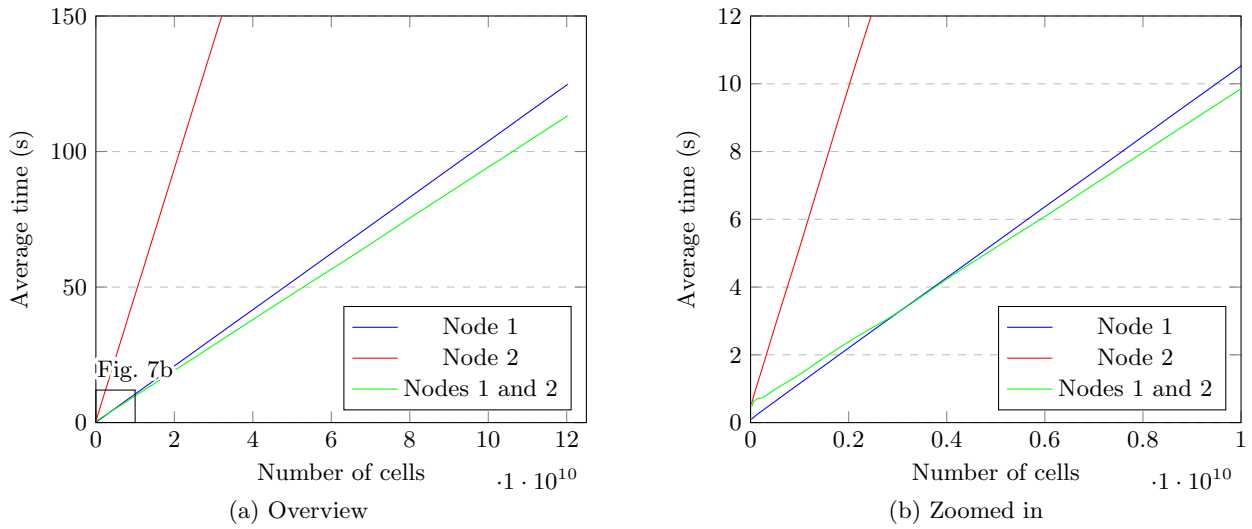
The explanation of the previous section also holds up for Algorithm 2, but a few line have been added. Lines 4, 13, and 14 are required in this algorithm to enable the writes in lines 17 to 20. Lines 24 and 25 reclaim those permissions. Lines 26 to 28 of Algorithm 2 are equivalent to lines 18 to 20 from Algorithm 1. The if statement on line 30 helps with the enforcement of permissions, as less work-items process cells at the same time.

6.3 OpenCL Algorithm 3

Verification of Algorithm 3 is trivial, as no work-item requires read permissions and no write permissions overlap. Lines 2 and 6 describe what write permissions are required.

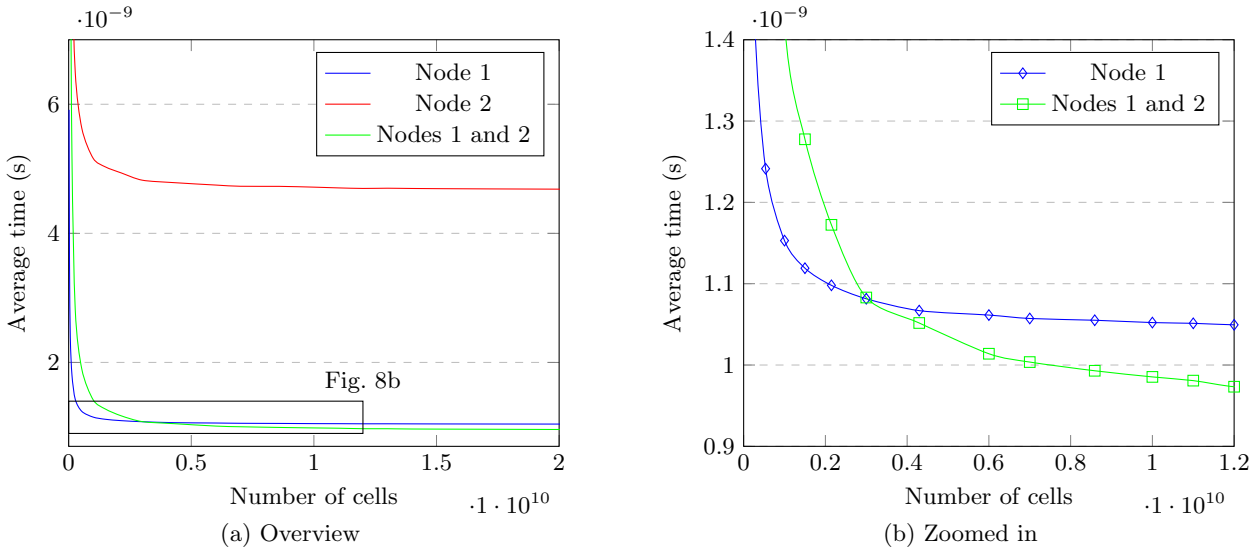
6.4 MPI Algorithm 4

As the nodes in a cluster only communicate the right most columns of blocks, no concurrent variable manipulation can occur. Therefore, permission-based separation logic is not required for this algorithm. There are no circular dependencies, so no deadlocks can occur. This cuts down



(a) Overview (b) Zoomed in

Figure 7. Average time to compute edit distance



(a) Overview (b) Zoomed in

Figure 8. Average time per cell

the need for verification even further, leaving only trivial *requires* and *ensures* statements.

6.5 Verification with VerCors

Unfortunately, OpenCL is not sufficiently supported by VerCors to do automatic verification. Rewriting the kernel in PVL, the native language of VerCors, does not help, as there is no support for kernel arguments. In further research support could be built into VerCors, but in the mean time the manual verification provided in Algorithms 1 to 4 will have to do.

7. TESTING THE PERFORMANCE

The implementation has been tested on a cluster of two nodes. Node 1 has an NVIDIA GTX 960M with a maximum work-group size of 1024, which was first introduced in 2015. Node 2 has an AMD R9 270X with a maximum work-group size of 256, which was first introduces in 2013. That means that the *width*, as discussed in Section 4.6, are 1024 and 256 on Nodes 1 and 2 respectively. The implementation has been run on the nodes separately and on the nodes as a cluster. This allows us to compare the time the program needs to return the result. The values used in Figures 7 and 8 are averages of ten separate runs. The x-axis represents the product of the lengths of the sequences compared, which is equal to the number of cells in the solution matrix as mentioned in Section 4.1. The y-axis represents the average time it takes to solve the problem in Figure 7 and the average time it takes for one cell to be computed in Figure 8.

Figure 7 shows that the implementation works faster on a cluster than on the individual nodes, if the number of cells becomes larger than approximately $0.4 \cdot 10^{10}$ cells. It also shows that Node 2 takes longer than Node 1 to solve the problem. This is due to the fact that the width of Node 2 is smaller than the width of Node 1. If the number of cells is equal to $1.0 \cdot 10^{10}$, Node 1 takes 10.5 seconds to complete, Node 2 takes 47.2 seconds to complete and the cluster takes 9.8 seconds to complete. The width used in Node 1 is 4 times as big as the width in Node 2, but the time required to solve the problem is 4.50 times as long. Therefore, we can conclude that Node 2 will be the

bottleneck in the cluster. The implementation does not take into account the difference in performance between nodes, so Node 2 defines the maximum performance of the cluster.

The total width of the cluster is equal to 1024+256 = 1280, and the width of Node 2 is 256. So Node 2 is going to handle $256/1280 = \frac{1}{5}$ of the pillars. That means that the time the cluster should take is at least one fifth of the time it takes Node 2 to solve the problem. There is however a small difference between $47.2/5.0 = 9.44$ and 9.8. This is probably due to the fact that the use of a cluster takes more time to set up and the communication provides some overhead.

The theoretical optimal speed of the cluster is $(1/10.5 + 1/47.2)^{-1} = 8.59$, which is 12% faster than the actual speed. This theoretical speed does not include the extra time to set up the cluster. Section 8.1 will discus optimisations required to approach this theoretical speed.

The set up taking more time can be seen in Figure 8. These graphs show the time taken per cell, so extra time taken independent of the number of cells will be visible in these graphs. The steep decline in time per cell at the left side of the graph is a clear indication that the time to set up is indeed significant. In these graphs it is also visible that the cluster becomes faster than Node 1 at approximately $0.4 \cdot 10^{10}$ cells, as the amount of time per cell in the cluster drops below the time per cell in Node 1.

From these results we can conclude that optimal number of GPUs in a cluster depend on the size of the sequences compared, when considering the speed of the calculation. For the specific cluster used in this section $0.4 \cdot 10^{10}$ cells is the size when the cluster is faster than the individual nodes. The cluster will however never be more efficient than the individual nodes, as the overhead on a cluster persist no matter the size of the cluster. This means that the user should consider whether the superior speed outweighs the inferior efficiency.

7.1 Comparison of algorithms

Comparison with the algorithm of De Heus is not useful, since it cannot solve the edit distance of sequences longer than the *width* of a node [3]. The implementation of this paper does not focus on such small sequences, as extra complexity and features results in a significant overhead while computing them. It is safe to say that for comparing small sequences the algorithm of De Heus is more suitable, as it does not suffer from the same overhead.

8. CONCLUSION

The steps required to distribute the verified implementation of the edit distance are described in Sections 4 to 6. The first step is explained in Section 4, which is the division of the algorithm as shown in Figure 2. The second step is explained in Section 5, which is the distribution of the pillars among nodes and the communication between the nodes. Finally, Section 6 explains the verification of the new algorithms used in the previous two steps. The three steps answer the first three subquestions mentioned in Section 2. Together they answer the research question mentioned in the same section.

The final subquestion is answered in Section 7, which discusses the performance of the implementation.

8.1 Future work on the implementation

Section 6.5 explains that VerCors offers insufficient support for OpenCL. A solution would be to improve the tool so that all functionality of OpenCL can be used. The code of the implementation described in this paper could be used as a simple test. Once the tool supports OpenCL any ambiguities or inconsistencies can be fixed in the implementation.

There are certain optimisations possible for the MPI algorithm. As mentioned in Section 7, one node can be the bottleneck of a cluster. This can be solved by either using the same hardware for every node, or to dynamically divide the work load between nodes. The dynamic division should allow nodes to negotiate what their tasks will be.

Another optimisation is the use of the CPU parallel to the GPU. The OpenCL algorithms can be converted to a CPU implementation so that it can help solve the edit distance problem.

The OpenCL algorithms could also be optimised. The memory used could be limited by using the algorithm of Meyers [7]. The advantage is that the columns shared between nodes takes less space, so the overhead for sending and receiving the columns should be reduced. However, a disadvantage is that the algorithm of Meyers requires extra operations per cell and thus reduces the performance of the algorithms. Therefore, the overall performance of the implementation is not guaranteed to be better than the current implementation.

9. REFERENCES

[1] A. Amighi, S. Darabi, S. Blom, and M. Huisman. *Specification and verification of atomic operations in GPGPU programs*, volume 9276 of *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 2015.

[2] K. Balhaf, M. A. Shehab, W. T. Al-Sarayrah, M. Al-Ayyoub, M. Al-Saleh, and Y. Jararweh. Using gpus to speed-up levenshtein edit distance computation. In *2016 7th International Conference on Information and Communication Systems, ICICS 2016*, pages 80–84, 2016.

[3] S. de Heus. A Case Study for GPGPU Program Verification. In *20th Twente Student Conference on IT*.

[4] G. Jo, J. Nah, J. Lee, J. Kim, and J. Lee. Accelerating LINPACK with MPI-OpenCL on Clusters of Multi-GPU Nodes. *IEEE Transactions on Parallel and Distributed Systems*, 26(7):1814–1825, 2014.

[5] Kronos Group. Conformant products. `https://www.khronos.org/conformance/adopters/conformant-products#opencl`.

[6] A. Munshi. *The OpenCL Specification*. Khronos OpenCL Working Group, November 2012.

[7] G. Myers. *A fast bit-vector algorithm for approximate string matching based on dynamic programming*, volume 1448 LNCS of *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 1998.

[8] G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, Mar. 2001.

[9] NVIDIA Corporation. CUDA GPUs. `https://developer.nvidia.com/cuda-gpus`, October 2017.

[10] University of Tennessee, Knoxville, Tennessee. *MPI : A Message-Passing Interface Standard*.