Distributing Local Sequence Alignment using Volunteer Computing

Alignment@Home

Haraldur Davíðsson
2809428
VU Amsterdam
h.b.davidsson@student.vu.nl

Paul Groß
2776170
VU Amsterdam
p.gross@student.vu.nl

Niclas Haderer
2766114
VU Amsterdam
n.haderer@student.vu.nl

Simon van Noort
2758494
VU Amsterdam
s.l.j.van.noort@student.vu.nl

Daniël Voogsgerd
2829940
VU Amsterdam
d.j.m.voogsgerd@student.vu.nl

Enrico Zeilmaker
2707877
VU Amsterdam
e.b.zeilmaker@student.vu.nl

Dr. Tiziano De Matteis
Supervisor
VU Amsterdam
t.de.matteis@vu.nl

Abstract

Sequence alignment is a technique for arranging DNA, RNA, and protein sequences, which is crucial for understanding relationships between organisms and analysing genetic variation. We propose, "Alignment@Home", a coordinator-worker-based distributed system for crowdsourced local sequence alignment, which leverages the Smith-Waterman algorithm for parallel processing across multiple nodes. The system supports numerous features, which include many-to-many alignment, intelligent capacity-based work scheduling, top-k results, an easy-to-use CLI tool, and fault tolerance. Our study analyses scalability and performance through experiments at system-, worker- and algorithm-level. The results of these experiments indicate limitations in the scalability of our system, particularly regarding the Coordinator and the communication between nodes. The source code and experiment details are publicly available[1].

1 Introduction

This paper introduces a distributed system for deoxyribonucleic acid (DNA) local sequence alignment, a technique that is used to identify regions of similarity between DNA sequences [1].

1.1 DNA Sequence Alignment

DNA is the molecule that carries the genetic instructions used in the growth, development, functioning, and reproduction of all known living organisms and many viruses [2]. As depicted in figure 1, it is made of two strands, each consisting of nucleotides, which are small units that include a nitrogen base (cytosine [C], guanine [G], adenine [A], or thymine [T]), a sugar called deoxyribose, and a phosphate group. The bases in the two DNA strands are linked together following specific pairing rules: adenine [A] pairs with thymine [T], and cytosine [C] pairs with guanine [G] [2].

Sequence alignment involves arranging DNA, RNA, or protein sequences to identify similar regions indicative of functional, structural, or evolutionary links. These alignments, presented as rows in a matrix, include gaps to align identical or similar elements and are applicable not only in biology but also in fields like natural language processing and finance to measure distance costs between strings [3]. While global sequence alignment compares entire sequences to find the best possible alignment over their full length, the focus of this system is local sequence alignment, which identifies regions of maximal similarity between a query sequence and a target sequence [4].

1.2 The Smith-Waterman algorithm

The Smith-Waterman algorithm is an algorithm that is commonly used to perform DNA sequence alignment [6]. The algorithm is used to find the best sub-sequence between a given query DNA sequence and a target DNA sequence.

The algorithm consists of two parts; a matrix filling, where the query and target sequences are aligned in an array, and an LCS (Longest Common Subsequence) [7] approach is applied. The second part is the backtracking phase, where the best sub-sequence match is found. The respective time and memory complexities of these phases are $O(mn)$ and $O(n)$, where m and n denote the length of the target and query sequences, respectively. In modern applications, a modification of this algorithm is used [8], which introduces additional parameters for a more intuitive result, a mismatch penalty, and a gap penalty.

[1] See github.com/Noorts/DLSA and github.com/Noorts/DLSA-Experiments.

1.3 Parallelization techniques of the Smith-Waterman algorithm

The parallelization of the Smith-Waterman algorithm can be split into two subclasses.

Intra-sequence Parallelization. The intra-sequence parallelization is the term that is commonly used to describe the parallelization of a single alignment (so one query-target pair). It can significantly accelerate the alignment process by exploiting concurrency within the alignment of these sequences. Multiple approaches have been developed to achieve this, such as the anti-diagonal layout [9], the sequential layout [10] and the striped layout [11]. Parallelization at this level is also referred to as *one-to-one* parallelization.

Inter-sequence Parallelization. The inter-sequence parallelization is the parallelization of multiple alignments (so more than one query-target sequence pair). The *one-to-many* layout, for instance, aligns multiple target sequences against a single query sequence, which is beneficial when one query is compared against a database of sequences [12]. One of the alternatives is a *many-to-many* layout [13], which refers to the alignment of many queries and many targets. The latter is particularly useful when comparing large-scale genomic analyses of many sequences.

In this paper, we detail the specific parallelization techniques used in every component of the system and provide a rationale for their application in the relevant context.

1.4 Motivation

This project draws significant inspiration from the Folding@Home paper written by Larson et al. [14]. Folding@Home is a distributed computing project that uses the processing power of volunteers' computers to simulate protein folding, aiding in understanding diseases and developing new therapies.

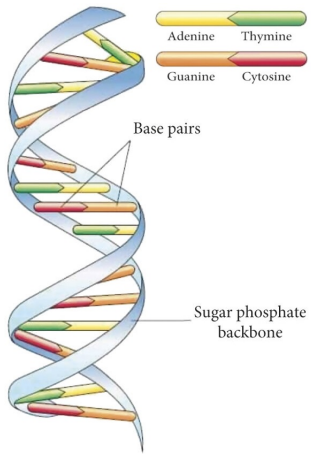

Figure 1. Structure of the DNA double helix [5]

After 20 years, Folding@Home has significantly advanced the understanding of protein folding and other biological processes, leading to insights into drug development and rapid responses to challenges like COVID-19. In addition to that, Folding@Home has become the world's first exascale computer in the process [15].

Similar to protein folding, sequence alignment played a crucial role in COVID-19 research as it aids in accurately identifying and analysing viral genetic variations, which is crucial for tracking the virus's evolution, understanding its transmission, and developing effective treatments and vaccines [16–18].

This project aims to design and implement a distributed system which is designed to leverage the collective processing power of volunteers' computers, much like Folding@Home, to facilitate more efficient and capable alignment of genetic sequences. With the increasing frequency of pandemics such as COVID-[19], it is important to enhance research methodologies to better prepare for and respond to such global health challenges.

1.5 Related Work

The Smith-Waterman Sequence alignment algorithm has been the subject of extensive research with different focus points. It is one of the algorithms that can be used to align sequences. In the following section, a review of some work related to the Smith-Waterman algorithm and its distribution will be made.

Xia et al. provide a review of different levels of local parallelism which, among other things, includes thread-level and process-level parallelisms [20]. This will be relevant for worker implementation, as there, different forms of parallelism will become vital to achieving good performance.

The challenge of reducing the memory space — one key limiting factor of the Smith-Waterman algorithm — was addressed by Zhang et al. using a combine and extend technique, allowing for the combination of intermediate results. In addition to that, each processor only stores a fraction of the traceback matrix, further reducing the memory size requirements [21].

The paper *Acceleration of the Smith–Waterman algorithm using single and multiple graphics processors* improves the performance of the Sequence Alignment algorithms using GPUs and concludes that the different cache architecture of GPUs is one of the reasons the algorithm scales well on GPUs. In addition to that, they reimagined the algorithm to only use sequential memory access, thereby further improving performance [22].

Zhang et al. use FPGAs to align sequences. They introduce a multistage processing element that reduces the FPGA usage, thereby allowing for more parallelism. Like other papers, they try to reduce the memory footprint of the algorithm. In this case, this was done using a compressed substitution matrix structure [23].

Efficient Distributed Smith-Waterman Algorithm Based on Apache Spark is closest to our problem, as they distribute the algorithm using Apache SPARK and add on-node parallelism using SIMD. This allows them to achieve 529 Giga Cell Updates Per Second (CUPS) [24] with 50 nodes and unknown hardware.

Szajda et al. focus on the challenges associated with ensuring privacy in a crowdsourced distributed computing environment. They intend to obfuscate the sequence alignment data, to make it less likely that private information can be read by peers in the distributed system. While the proposed privacy mechanism generally works, there are cases in which it does not [25].

Finally, Folding@Home addresses a similar challenge in a slightly different field. Protein folding is a very compute-intensive task that can take a long time on a single machine. However, distributing the workload and using multiple nodes to calculate the solution helps. Folding@Home does this using an intelligent scheduling mechanism that not only considers the computing power of a node but also its network speed, as large amounts of data have to be sent over the network, resulting in a non-negligible difference between the nodes caused by the network alone. In addition to that, they verify that the data is correct by signing the data and attaching the checksum to the computed result [14].

1.6 Structure

In this paper, we will outline the necessary requirements for our system and discuss its overall design (section 2 and section 3). Subsequently, we will assess how both the system and our algorithm implementation adapt to varying problem sizes (section 4). Finally, we analyse our findings and suggest possible enhancements for the system (section 5 and section 6). Section 7 discusses the project's approach to reproducibility and contains references to the project's source code and experiment archive.

2 Background

The planned architecture has a Coordinator and multiple workers, where the Coordinator is responwsible for distributing the jobs to the worker nodes. These worker nodes can register themselves, get a workload, compute it, and send it back. The Coordinator then sends the aggregated result back to the client, which requests for the sequences to get aligned.

The functional and non-functional requirements are included below (bonus features are marked by the (Bonus) prefix).

(FR1) The system MUST be able to distribute the Smith-Waterman algorithm's work across multiple machines.

(FR2) The system MUST be able to match sequences of varying lengths and sizes, up to large (~3GB) genomic datasets.

(FR3) The results produced by the distributed system MUST be correct. The system must arrive at the same answers as the sequential Smith-Waterman algorithm would on a single machine.

(FR4) (Bonus) The system SHOULD support multiple sequence alignment schemes. This includes support for one-to-one, many-to-one, one-to-many, and many-to-many alignments.

(FR5) (Bonus) The system SHOULD return the best k alignments for a sequence combination and have different algorithm parameters.

(FR6) (Bonus) The scheduler of the system SHOULD take the compute capacity of the individual devices into account.

(FR7) The system SHOULD be able to deal with heterogeneous worker hardware and assign work accordingly.

(FR8) The system should verify if the returned result by a worker is correct.

(NFR1) The system SHOULD provide a speedup over the sequential algorithm.

(NFR2) (Bonus) The system MUST tolerate worker node failures.

(NFR3) The experiments SHOULD be easy to reproduce.

3 System Design

The system is based on a Coordinator-Worker architecture and supports both strong and weak scalability (see section 4 for more details). Strong scalability refers to a system's ability to effectively utilize more processing power to decrease the time required to complete a single task, whereas weak scalability measures the system's capacity to maintain efficiency while the workload and processing power both increase proportionally.

3.1 Overview

On a very high level, three components constitute the distributed sequence alignment system (see figure 2).

1. User
2. Coordinator
3. Worker

As a system **User**, one has the option to request the Coordinator to perform sequence alignments. For this to work, the user simply provides the sequences that should be aligned with given parameters. Thereafter, the user periodically checks in to obtain the progress of their alignment jobs.

The **Coordinator** is responsible for communicating with the user and taking requests for sequence alignments from said user. These requests are then distributed among the registered workers (**FR1**). Once the worker has completed the task and aligned all sequences, and the Coordinator has aggregated the alignment results sent to it by the workers, the user can then collect their outcomes.

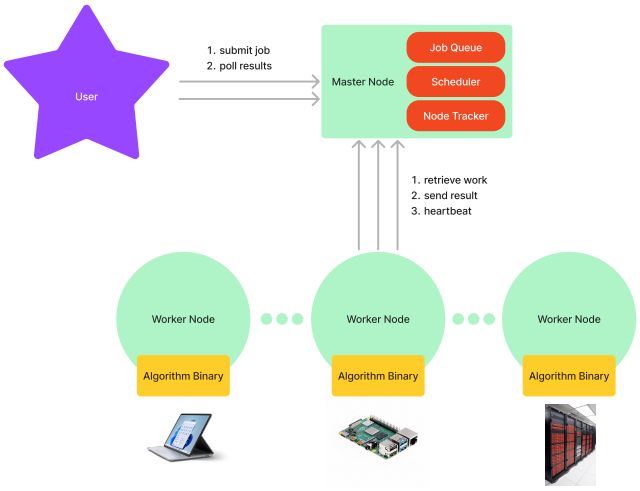

Figure 2. High-Level System Overview

The aforementioned **Worker** registers with the Coordinator and gives the Coordinator an estimate of their computing capacity. This capacity can be used by the Coordinator to schedule jobs more fairly (see section 3.1.3). A worker node periodically checks for new jobs and, in case it gets a job assigned, computes the alignments using the Smith-Waterman algorithm and returns the result.

3.1.1 CLI. The system has a built-in CLI to upload sequences to the Coordinator. The CLI polls for results and the Coordinator responds with the status of the job. The job can either be in queue, that is, waiting until calculations commence on the submitted sequences, or in progress, where computations have begun. In the latter case, the server sends a percentage back to the client, indicating what percentage of the total workload has been computed. This is displayed on the client side in a progress bar.

3.1.2 Features. The system has several features. By using the CLI, the user inputs the name of the query and database file that should be used. Each file can either be in the Fasta or multi-Fasta format, meaning, the user can specify up to multiple query sequences and multiple target sequences (**FR4**), where each sequence can be of varying lengths (**FR2**). In addition to that, an output path can be specified, where the results will be saved once the alignment has been completed. The user can choose to supply the top-k (**FR5**) parameter, for example, if top-k = 5, the top 5 highest scores will be saved in the output file for each query sequence in the query data set. The alignment scoring and penalties of the Smith-Waterman can also be provided as, **match-score**, **mismatch-penalty** and **gap-penalty**, representing the match score, mismatch penalty, and gap penalty parameters of the algorithm (**FR5**). This allows the user to adjust the algorithm to their needs.

3.1.3 Compute capacity and intelligent scheduling. Allocating resources for a job without prior knowledge of its characteristics, such as runtime and resource requirements, is generally inadvisable. Employing a scheduling algorithm under such conditions can result in suboptimal performance. Therefore, it is necessary to understand the intricacies of said task. In our case, the only metric the scheduler has to consider is the runtime because the RAM and special hardware are not an issue for this algorithm section 3.3.

To estimate the time needed for a task, the performance metric CUPS is used for the worker nodes. The time a node needs to align a sequence is thereby the number of bases in the query sequence N_q times the number of bases in the target N_t divided by the CUPS (equation (1)).

$$t = \frac{N_q \cdot N_t}{\text{CUPS}} \tag{1}$$

Using this time estimation, the Coordinator can estimate how long a worker node might take to align a particular query-target pair. We developed two scheduling algorithms that make use of this metric.

Proportional work distribution. This scheduling algorithm assigns every available worker node a set of sequences corresponding to the performance of said worker node (see figure 3). The jobs themselves are scheduled using the FIFO policy. This, however, has the disadvantage that new nodes do not get any new work assigned if no new jobs are getting added. Therefore, the new worker node is not utilized at all during calculations.

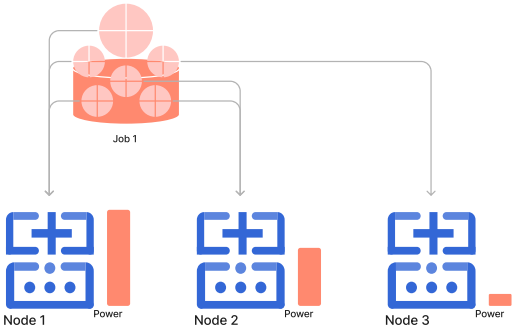

Figure 3. Proportional work distribution. Assign work packages according to worker performance

Time-sliced work distribution. In this method, each worker node is assigned tasks intended to keep it busy for a specified duration (e.g., 3 minutes, as shown in figure 4). This is an advantage as it allows even newly added worker nodes to take on tasks that have not been started yet. In the

least favourable scenario, there might be a delay of up to 3 minutes for the user if a task is assigned to a less efficient worker node. However, this period can be adjusted. Using this scheduling strategy enables the system to meet **FR6** and **FR7**, and potentially **NFR1**.

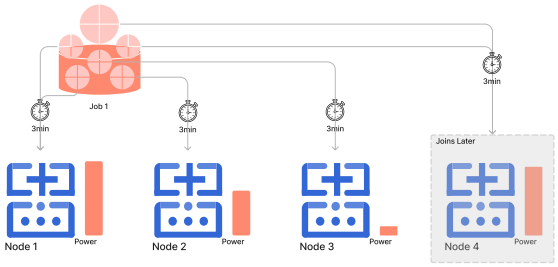

Figure 4. Time-sliced work distribution. Assign each worker 3 minutes of work

One limitation of this approach is the lack of performance consistency on the worker node. For example, the start of a work-intense task, such as a Windows update, requires CPU usage that would otherwise be allocated to the computation, and this would delay the completion of the task. However, because there is an estimation of how long it takes for the given alignment to finish, it is possible to preemptively stop the calculation on that node and redistribute the work to other nodes if the node exceeds its time budget. This has, however, not been implemented yet.

3.1.4 Fault-tolerance. The system is completely fault-tolerant at the worker level (**NFR2**). The worker sends a heartbeat every n seconds to the Coordinator to let it know that it is alive. If the Coordinator does not receive this within the given time frame, it removes the node from the network. It then reallocates the non-completed sequences of the worker to other worker nodes in the network. In the absence of other nodes within the network, the Coordinator remains idle, awaiting the arrival of a new node, at which point it will distribute the remaining tasks appropriately.

Making the Coordinator fault-tolerant, as well, would require storing the internal state of the Coordinator using a type of fault-tolerant storage. In the occurrence of a Coordinator crash, the newly started Coordinator node would simply be able to read the last saved state from the database and continue to operate normally. Because work packages are currently the only units being tracked, this would be an effective solution.

3.1.5 Result validation. The Coordinator, upon receiving a worker's results, will verify whether the result is correct given the provided query-target sequence pair. This is a partial correctness check (**FR8**), as it does not check whether the result provided is also the best. Performing a full correctness check would entail recomputing the full results, which we've decided not to do for this project.

3.2 Workflow

1. The CLI parses the FASTA files and gives each FASTA sequence a UUID. Internally, a mapping between said UUIDs and the FASTA IDs is created. The parsed data is then sent to the Coordinator using its HTTP API.
2. When the data arrives at the Coordinator, the Coordinator adds the job to the queue, and once the data is at the top of the queue, it applies the scheduling technique described in section 3.1.3 and sends the corresponding query and target IDs to the worker.
3. Once the worker receives its work, it iterates through the query and target ID pairs that make up an alignment job and makes an HTTP request to the Coordinator to get the associated sequences. This was done to prevent a request timeout resulting from querying large amounts of data from the overwhelmed Coordinator.
4. After the worker has received all the sequence data, it iterates through each pair in parallel, spawning multiple goroutines [26] such that each core is utilized, thereby implementing inter-sequence parallelization.
5. The worker uses the Coordinator's HTTP API to send the data back to the Coordinator, once a query-target pair has been aligned and the workers' result buffer is full. With this approach, the Coordinator can provide an accurate progress report to the client.
6. Once all sequences have been aligned, and the job marked finished, and the user can retrieve its result.

3.3 Algorithm

Three different versions of the Smith-Waterman algorithm have been implemented for this project:

- A classic rectangular matrix version.
- An anti-diagonal implementation using Single instruction multiple data (SIMD) instructions
- An anti-diagonal SIMD version with a circular array

We will start with some quick notes on the three different versions

Classic. This implementation is the typical implementation as described in the original Smith-Waterman article [6]. This version has no parallelism at a node level and has a low level of complexity. For this project, it served as the reference implementation, but as we will see later, it is also our fastest implementation in certain circumstances.

Scoring Matrix. We have to make a deliberate choice in the size of integers for the scoring matrix, as the size of the integers has a measurable impact on both the performance of the algorithm and the resources needed to compute such

a sequence. The speed is impacted because the smaller the size of the integer chosen, the more parallel SIMD lanes one can use in the fixed-size SIMD instructions. Additionally, the memory footprint is proportional to the size of the integer used as well as the vast majority of the used memory originates from the dynamic programming matrix in the Smith-Waterman algorithm.

To know what size integer, we need, we have to calculate the maximum value we could feasibly get in the scoring matrix. The highest score in the matrix is naturally the score of a full match, which can be calculated beforehand:

$$V_{\max} = \|\text{query}\| \times s_{\text{match}} \tag{2}$$

As we will be working with both positive and negative values, we opt for signed integers. From this, we can conclude that 16-bit signed integers allow for scores up to 32767. This can be calculated at runtime, and the implementation could switch to larger-sized integers if necessary.

Anti-diagonal layout. The Smith-Waterman algorithm can be parallelized effectively using an anti-diagonal structure. This involves shifting the second column of the matrix down by one cell, the third column by two cells, and so on. When you analyse how each cell in a row depends on other data, you'll find that this layout eliminates dependencies within the same row. As a result, each row can be processed simultaneously. To accommodate this, one could either create a function that re-indexes to match the original matrix or increase the height of the scoring matrix by the length of the query, providing space for the shifted cells.

SIMD. By using the anti-diagonal layout, we find we can fully parallelize the computation of a single row. There are several common ways of doing so, the two most prominent possibilities in consumer hardware are either GPUs or SIMD instructions. For practical reasons, we chose SIMD for this project. However, one could make the argument that using SIMD is the most commonly available technique. We use the portable SIMD implementation of the Rust programming language, as it allows us to write a single version of the algorithm that can compile to different SIMD instruction sets.

Circular arrays. We can further optimize this implementation-specific, but very common case. If either the query or target is much larger than the other, we can use the fact that there is an upper bound on the match length to limit the larger dimension of the scoring matrix. To limit the height of the scoring matrix, we transform the matrix into a 2D circular array where the y-axis wraps around after the maximum possible match length.

As an illustration of the impact of this optimization, consider an alignment of the SARS-CoV-2 virus spike protein genome ($\sim 4 \times 10^3$ base pairs) to the entire human genome ($\sim 3 \times 10^9$ base pairs). Assuming that, we can use 16-bit integers to represent the scoring matrix (see section 3.3). In the classic implementation, we end up with a matrix of size of 24 TB. Using circular arrays, this is reduced to 128 MB, assuming a match score of 3 and a gap penalty of 1.

4 Experiments

This chapter contains experiment descriptions, setups, results, and analyses. Section 7 contains additional details about the experiment setups (e.g., the DAS5 environment). References to the project's source code and the experiment setup and results archive can also be found there.

Workloads. In the experiments, two types of workloads are used, real sequences and synthetic sequences.

The real sequences are sequences publicly available from a database (e.g., NCBI) and were chosen based on sequences specified in the course's competition document (see appendix A).

The synthetic sequences were generated by a custom-made script (*/utils/generate_synthetic_data.py*).

Experiment levels. A handful of experiments were run to gain insight, at multiple levels, into the scalability and performance of the system. The experiments are grouped into the following three levels.

1. System level. These encompass the entire system, including the user, coordinator, and worker(s) (section 4.1).
2. Worker level. How well does the algorithm scale for multiple cores (section 4.2).
3. Algorithm level. Takes a look at single core performance for different hardware (section 4.3).

4.1 System-level experiments

The first two (system-level) experiments test the complete distributed system. This includes the aforementioned CLI (see section 3.1.1) and the Coordinator and worker nodes. As the CLI is the intended way to interact with the system, the CLI is used as the tool to invoke a job and to fetch the result.

The system-level experiments were run on DAS5's VU compute cluster [27], which uses the SLURM workload manager. This cluster consists of 200 nodes [28]. To make sure our experiments do not trouble other users of the cluster, a maximum of 16 nodes were used simultaneously. Furthermore, the high-speed InfinityBand interconnect technology of the cluster was used for internode communication (i.e., CLI to Coordinator, and Coordinator to worker(s)).

In terms of metrics, the CLI collects two types. The *elapsed_time* is the time between the starting of a job through the CLI and the retrieval of the alignment job's results. In other words, this is the time a user has to wait. This is also the same metric used for the competition. The second metric, *computation_time*, is tracked by the Coordinator node and sent to the CLI and is defined as the time between adding a job to the scheduler's queue and the job's completion on

the Coordinator (this thus excludes CLI to Coordinator communication latency). By default, both metrics are denoted in milliseconds.

Lastly, to track the nodes' CPU and RAM utilization, the *top* program's output was collected.

4.1.1 System-level Scalability.

The initial step in conducting the system-level scalability experiments involved selecting the parameters, specifically the number of sequences and their respective lengths. The approach used for this was to balance the sizes around the computation_time metric such that an alignment job does not take too little time (e.g., less than 3 seconds[2], which would see the measurement become increasingly more vulnerable to overheads) nor should a complete experiment take too much time (e.g., 8+ hours). In addition to that, 10 measurements were collected per parameter configuration (A parameter configuration in this case refers to a set-up in which all dimensions are held constant. For example, in the first strong scalability experiment, 5 query-target file pairs are tested for 5 configurations (i.e. a different number of nodes (1, 2, 4, 8, 16)), thus the total number of measurements is $10 \times 5 \times 5 = 250$.). For these system-level scalability experiments, the median, as well as confidence intervals are used for statistical insight. The Coordinator's default *proportional* scheduling policy was used.

4.1.2 Exp. 1 – Strong scalability.

Goal. The goal of the first experiment is to gain insight into the strong scalability of the overall system. Strong scalability, in this case, is the ability of the system to distribute a fixed number of sequence alignments across multiple worker nodes.

Setup. For this, 5 datasets were used, 3 synthetic datasets that differ in the number of sequence alignments, but in which the query and target lengths stay constant across datasets [3], and the extra large and large datasets described in the competition document, which differ both in length and number of sequences but also represent more realistic use cases.

[2]An exception to this are the competition's small and medium datasets with their sub-second computation_time.

[3]The query sequences' length ranges between 200 and 1000 characters. The target sequences' length ranges between 10 000 and 200 000 characters.

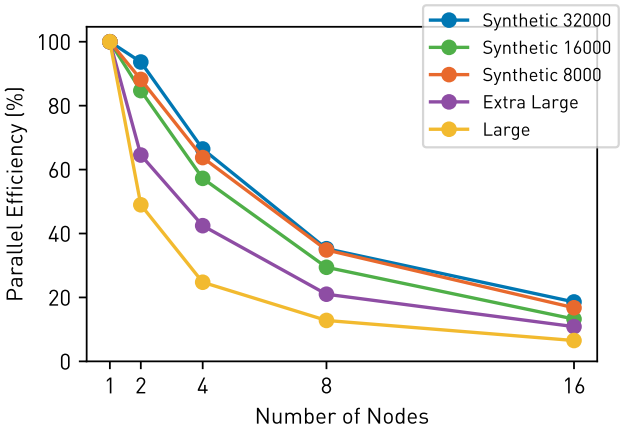

Figure 5. Exp. 1 – Strong scalability of the system. The points indicate the median of the data.

Expectation. Ideally, the parallel efficiency would be almost 100 % across all node counts, slowly tapering off downward. This would mean that the system is strongly scalable and can thus utilize additional computing resources to efficiently process a fixed workload more quickly.

Results. Figure 5 shows the results of the experiment using the achieved parallel efficiency percentage. The parallel efficiency percentage in this case is defined as the median of the sequential result divided by the median of a given point (e.g., for 2 workers) divided by the number of workers times 100. E.g., with 1 node the median computation_time is 80, with 2 nodes it is 50, then the parallel efficiency percentage is $80 \div 50 \div 2 \times 100 = 80$ %.

Unfortunately, even for the best dataset ("Synthetic 32 000", which has 32 000 alignments to process), the parallel efficiency drops rapidly. This seems to indicate that the current system has a large overhead.

Furthermore, whereas using 2 nodes to process the synthetic datasets results in a parallel efficiency of ~88 %, processing the datasets with fewer sequence alignments (L: 7, XL: 705[4]) is drastically worse. This could be due to load imbalance caused by a low number of sequences and a large variance in the length of the sequences (e.g., XL has a minimum length of ~13 000 and a maximum length of ~21 million). An approach to reducing the impact of such outliers could be the use of a more intelligent scheduling policy, such as the *time-sliced work distribution* policy (section 3.1.3).

A shallow dive into the CPU utilization for one of two worker nodes when processing 32 000 synthetic alignments shows that the 1-minute load average is between 20 and 26 virtual cores. This seems fine given that the available number of virtual processors on the node is 32.

[4]Our Fasta file analyser was used to determine the number and the length of the sequences. See */utils/fasta_file_analyzer.c.*

The strong scalability experiment also included a run with a synthetic dataset with 64 000 alignment pairs (the rest of the setup was the same; see section 7 and the experiment archive repository for more information). Processing these alignments using a single worker takes almost 3 minutes. Using 2 workers led to instability. The Coordinator was unable to successfully manage the alignments (which included handling intermediate results and sending out sequences to align) and thus this led to the workers timing out.

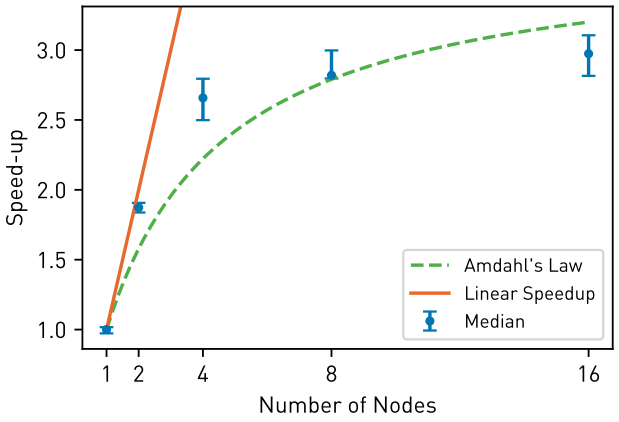

Figure 6. Exp. 1 – Strong scalability of the system processing the synthetic dataset with 32 000 alignments. The error bars represent the 95 % confidence interval.

Finally, figure 6 highlights the best-performing dataset from figure 5, this time with the speed-up on the y-axis. The speed-up is defined as the time to process a workload by 1 worker divided by the time to process the workload on N workers. E.g., $70 \div 40 = 1.75$. This speed-up has been achieved on the "Synthetic 32 000" dataset (the strongest scaling one from the figure 5). Fitting Amdahl's law to the median of the measurements produces a parallel fraction of 73 %.

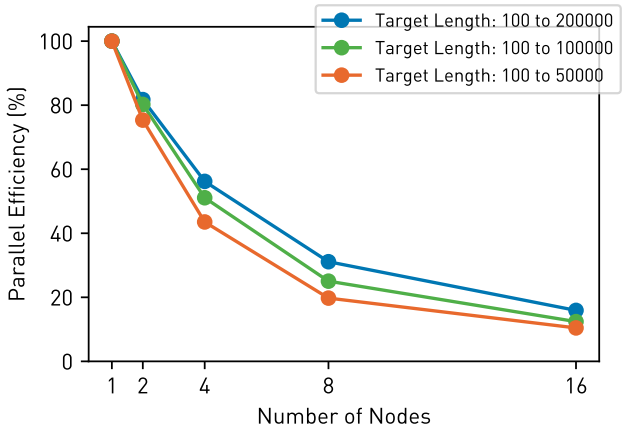

Figure 7. Exp. 2 – Weak scalability of the system processing a synthetic dataset.

4.1.3 Exp. 2 – Weak scalability.

Setup. The goal of the second experiment is to provide insight into the weak scalability of the system. Weak scalability, in this case, is the ability of the system to process a fixed workload per worker. For example, 1 worker can process 2000 alignments, whereas 2 workers can process 4000 alignments.

Synthetic sequences were used such that the workload could easily be kept at a fixed rate per worker. 3 datasets were used, which differ by their target sequence length range (e.g., 100 to 200 000 characters). This range influences the length of the sequences that are generated, and thus the total workload of the system.

Results. Figure 7 displays the parallel efficiency percentage of the weak scalability of the system. As the number of nodes increases, the ideal scenario would see the line initially remain high and then gradually descend towards a lower parallel efficiency. However, in this case, it quickly drops off to around 50% efficiency with only 4 workers. The system does seem to benefit from additional workload, as the sequence length range [100 to 200 000] achieves higher parallel efficiency than the smaller ranges.

The comparison between the parallel efficiency observed in the strong scalability experiment and that observed in the weak scalability experiment, tells us that the efficiency of strong scalability exceeds that of weak scalability. This does not adhere to our understanding of scalability, where achieving strong scalability is typically more challenging than achieving weak scalability. Upon analysis, we found the main culprit to be the coordinator, which is overwhelmed quickly, resulting in substantially longer times to get the work for the worker and return the work results to the coordinator. Another part of the reason could be that the dataset sizes were too small (a computation time between 3 and

40 seconds) resulting in a lot of communication and work requesting overhead.

4.2 Worker-level scalability

On the worker level, we measured the strong scalability of the system by increasing the number of cores used. The experiments were run on a MacBook M1 machine on an almost idle CPU. For more setup details, see section 7.

4.2.1 Exp. 3 – Worker-level strong scalability.

Setup. We measured the workers' thread-level scalability by supplying a fixed number of used threads in the parallel section of the code. We split every query-sequence pair into equal chunks and start multiple threads on the machine, where each thread receives the same number of query-sequence pairs to compute.

Expectation. We expect high parallel efficiency, but less so for imbalanced data sets, due to the load balancing issue our method incurs.

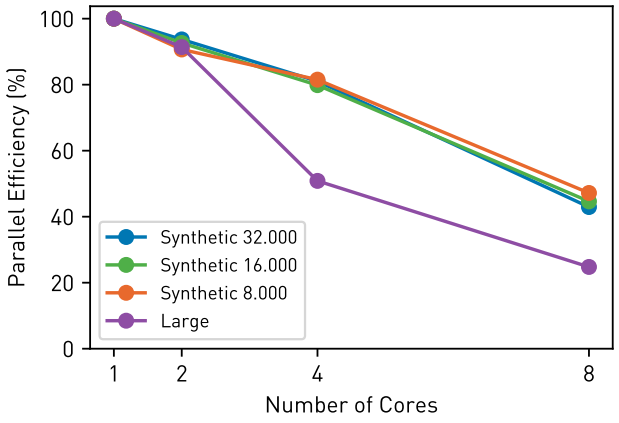

Figure 8. Exp. 3 – Strong scalability of the worker.

Results. This method lacks robustness against variations in sequence lengths, as it leads to load imbalance. We, therefore, decided to implement an approach that uses work stealing to resolve this issue and reevaluated the performance in 4.2.2.

4.2.2 Exp. 4 – Worker-level strong scalability with work stealing.

Setup. In this experiment, we used Rust's Rayon[29] package, which implements work stealing efficiently for parallel code iterations. Similar to the previous experiment, we manipulated the number of threads spawned within the program to measure this.

Expectation. We expect the parallel efficiency to stay the same for the balanced synthetic data sets, but the work-stealing should address the discrepancy in the Large one.

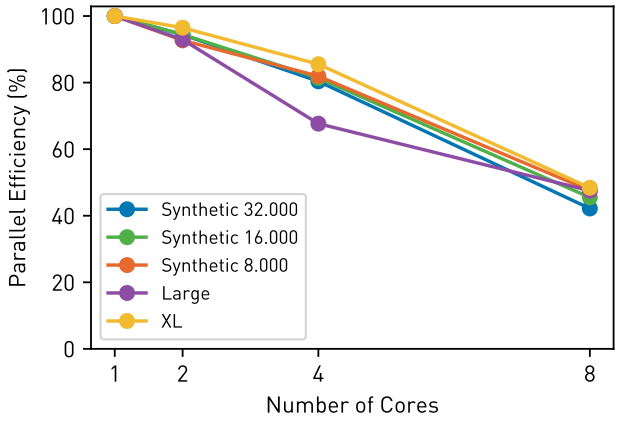

Figure 9. Exp. 4 – Strong scalability of the worker using the work-stealing approach.

Results. The parallel efficiency of the Large data set has indeed improved by a large margin, making the worker more robust to imbalanced sequences. We observed improved performance for synthetic data sets with larger query- and target ranges, although this was not measured. We fitted the average speed-up of the experiment in this approach to Amdahl's law (figure 10), which yielded a parallel fraction of 0.85.

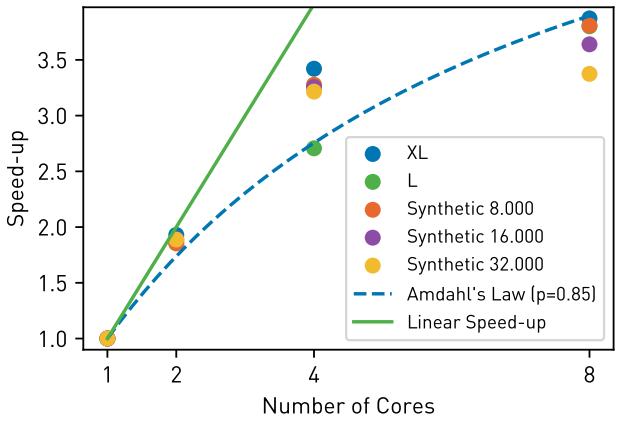

Figure 10. Exp. 4 – Average speed-up of each dataset fitted to Amdahl's law.

Unfortunately, due to time constraints, the work-stealing approach was not included in the system-level experiments (section 4.1). We hypothesize that this approach would have improved the strong scalability in the system-level experiments, in figure 5, making the efficiency of the Large and Extra-Large datasets more consistent with the values reported in the synthetic ones.

4.3 Algorithm performance

For this project, we wrote a custom implementation of the Smith-Waterman algorithm using SIMD instructions and circular arrays. Firstly, we will investigate the scalability of the algorithm. Secondly, we will look into the influence of different targets on the performance.

4.3.1 Exp. 5 – SIMD lane scalability.

Setup. For this experiment, we have chosen a query size of 320 and a target size of 131072. The reason for these sizes is fairly trivial: The intergroup competition used a query size of 346, and 320 is the closest multiple of 64 which in turn is the highest lane count supported by the Rust portable SIMD library. The target size is the largest power of two for which the benchmark would run in a reasonable amount of time.

Expectation. A classic Smith-Waterman implementation should have performance almost independent of the query and target contents, as long as they remain of the same size. This is because regardless of the contents of the query and target, the algorithm would have to calculate the entire matrix with complexity $O(nm)$, after which it will have to do a single traceback taking $O(\min\{n, m\})$ time. However, this is not the case for our circular array implementation. Since we are overwriting our score matrix, we need to do the traceback as soon as a new maximal score is found. Therefore, the worst case would be a query that is equal to the target.

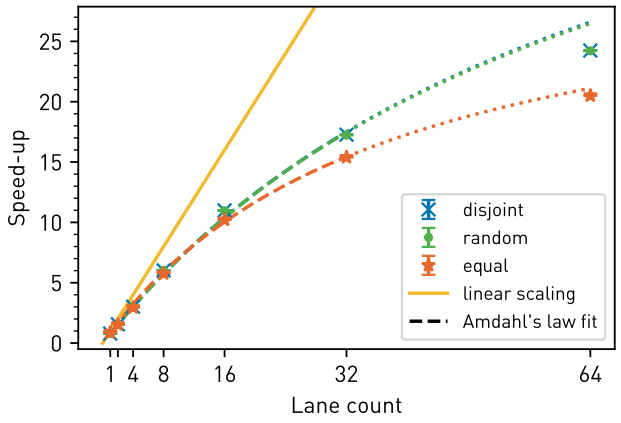

Figure 11. Exp. 5 – Strong scalability of the SIMD circular array algorithm regarding the number of SIMD lanes used for different target sequences. The error bars represent the 99 % confidence interval. Amdahl's law has been fitted to the supported 32 lanes, after which it has been extrapolated.

Results. We ran the algorithm with a query and target consisting of identical nucleotides (equal), different nucleotides (disjoint), and uniformly distributed random nucleotides in the target and a single repeated nucleotide in the query (random). The measured speed-ups are shown in figure 11. As expected, the target with identical nucleotides performs the worst, as the most amount of tracebacks have to be performed. The disjoint target performs slightly better than the random target, but the difference is negligible. We find parallelisable proportions of 98.6 %, 98.6 %, and 97.4 % for the disjoint, random, and equal fit respectively. We can also see that when we try to use more SIMD lanes than available, the performance is getting worse than you would assume from extrapolation of Amdahl's law. This is as expected, as it would compile to multiple SIMD instructions. It still does perform better than with a lower lane count, which could potentially be attributed to compiler optimizations. However, we did not investigate this further.

4.3.2 Exp. 6 – Target size dependence.

Since the goal of the project is to find alignments on many different machines, we ran a target scalability experiment on all machines of the mentioned authors.

Setup. For this experiment, we again take a query size of 320 nucleotides, but now we vary the size of the target. We use the disjoint target here, as we are interested in the speed at which we construct the score matrix. We run this benchmark on a varied set of CPUs.

Results. The results of this experiment are shown in figure 12. We observe an exponential increase in performance until a target size of 131 072, after which the throughput plateaus.

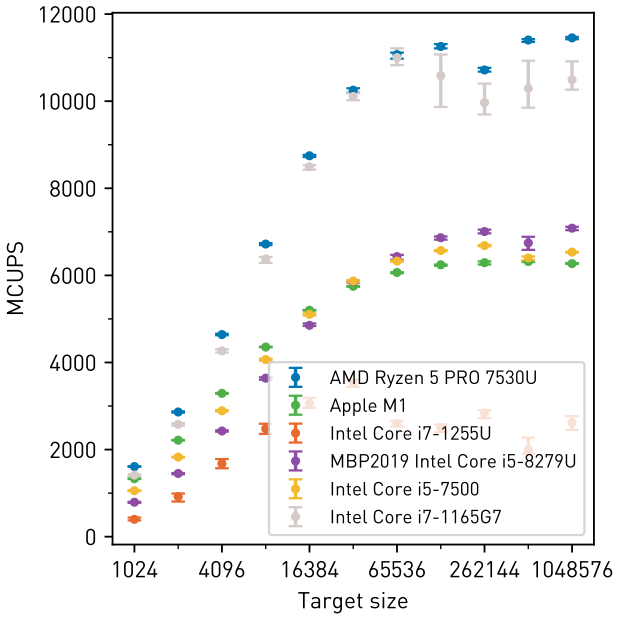

Figure 12. Exp. 6 – Throughput of the circular array algorithm for a query of size 320 with a disjoint target of different sizes for different CPUs. The error bars represent the 99 % confidence interval.

5 Discussion

5.1 Coordinator

The main limitation of the system lies in the Coordinator. For now, it does not scale well to massive workloads, as it has trouble distributing the sequences efficiently.

The main problem hereby is the choice of the programming language. Python does not allow more than one thread to execute Python bytecode at once [30] which is a problem as simply the act of distributing larger sequences to multiple workers overwhelms the asynchronous Python server. Future work could include another implementation written in a programming language with multiple concurrent threads (like Golang where every request is executed in a separate goroutine [31]), allowing for multiple worker nodes requesting work at the same time. We believe this step would already improve the scalability substantially. This problem expands to limitations in the number of jobs a worker can handle.

Alternative designs of a scheduler could be implemented as well. In a production setting, a scheduler that uses a more fair scheduling policy at a job level would be more feasible, instead of a simple FIFO, which is the current policy of our system.

5.2 Communication

Another fault lies in the way data is processed and communicated between components. More efficient implementations, such as reducing the data size by using a custom data format, and persisting massive datasets not in memory but in a database, would make the whole process more scalable and robust to larger workloads. However, we assessed that this lies outside the scope of this project, but future work could include an improvement in this area.

5.3 Worker

At the worker level, we used thread-level parallelization in combination with inter-sequence parallelization to achieve many-to-many parallelization. This is a similar approach to existing state-of-the-art tools such as Seq-An [13]. An alternative approach would be to partition the alignment into a tile and give each process a tile to compute [20]. We deemed the former approach to be simpler and more fitting in a distributed setting, since the latter appears to be more optimized for a one-query, many-targets setting. A performance analysis measurement of these two approaches would be an interesting addition.

The main bottleneck of the approach is the parallel efficiency of the strong scalability, which can be seen in Figure 9. From Amdahl's law, we obtained a value of $p = 0.85$. This suggests that 15% of the workload cannot be parallelized. We expected the p to be a lot larger as the only overhead should be when the data is communicated between threads, the context switching and communication overhead.

5.4 Experiment limitations

Heterogeneous validation. Performance validation of heterogeneous systems is notoriously difficult and a very active area of research [32]. Due to the scale of our system, we decided instead to focus on measuring the scalability with homogeneous workers and analysing the performance at every level of the application. An alternative approach to the experiments could have included measuring how robust the system is to heterogeneous worker nodes and how it scales. We have only verified that the system works and performs well in these conditions, but we have yet to measure it. Our intelligent work scheduling suggests the performance would be stable.

Communication Overhead. Another performance metric we would have liked to measure was the communication overhead. The communication overhead was enormous due to the inefficiency of the communication between the nodes. An example of this can be seen in the difference in the time it took to compute the competition datasets Table 1. Local computation, on an Apple M1 8 core processor of the XL and L datasets took 4326 ms and 1310 ms respectively, while the median values for the distributed approach in the competition were 110 579 ms and 5512 ms. However, a full experiment with different workloads would be needed to fully analyse this. While our results highlight the system's efficiency at the worker level, they also underscore the potential negative impact of communication overhead on overall performance.

Scheduling Performance Comparison. A performance comparison of the different scheduling techniques would also be interesting. Integrating this with heterogeneity experiments and conducting a stress test to observe how the system scales with an increasing number of users and jobs would offer substantial insights into the system's dependability and viability as a production-level application. Unfortunately, limitations in the current Coordinator design, such as scalability and fault tolerance constraints, made such an analysis unfeasible. This limitation points to a critical area for future development and improvement.

Scale. Currently, the system has only been tested with a limited number of nodes (a maximum of 16 on DAS5). However, in a production environment, there would be several orders of magnitude more devices in the system. Because of the limitation of the Coordinator (see section 5.1) it is apparent that the system in its current form would not be able to handle an experiment even close to the production scale. However, it would be interesting if the architecture would support that many machines.

6 Conclusion

In this project, we implemented a distributed system for DNA sequence alignment using the Smith-Waterman algorithm. We wrote a memory-efficient implementation of the

algorithm and used thread-level parallelization to improve the efficiency. We separated the system into components and analysed and tested the scalability and performance bottlenecks of each component with different workloads and parameters. A key factor was the creation of the intelligent work scheduler, which schedules the work efficiently and enhances the system's robustness to heterogeneous worker nodes.

We found the main bottleneck to be the Coordinator and the communication methods employed. These elements hindered the system's overall effectiveness, making it far from production-ready. As a result, even with homogeneous workers, the system exhibited suboptimal parallel scalability. However, we have suggested many potential fixes to address these issues. These include adopting a programming language better suited for the Coordinator, implementing data streaming, and fully integrating the work-stealing method on the worker level.

In conclusion, while the project faced challenges, our findings demonstrate the feasibility of distributed DNA sequence alignment. With the suggested improvements, we are confident the system has the potential for more optimal scalability and efficiency.

7 Reproducibility

This project's source code is publicly available online[5], which allows others to rerun the experiments and allows them to reproduce the results as required by **NFR3**.

Furthermore, detailed experiment setups and results (including the types of sequences used and software versions), and the plotting code and plots have been archived and are publicly available in a separate repository[6].

7.1 System Experiments on DAS5

Before our system can run on DAS5, the environment has to be set up. This includes setting up a Python environment, setting up Go, and cross-compiling the rust binary used inside the worker. All details regarding this setup process (including software versions) are detailed in the *DAS5.md* file available in the main repository.

Running experiments on the cluster entails starting a Coordinator node and (multiple) worker node(s) and "executing" a job through the CLI. This process has been automated through a series of scripts.

The main script (*/utils/run_das5_experiments.py*) allows one to define the number of worker nodes, the number of experiment iterations, and the CLI parameters (including the sequences to use). Executing the script will automatically run the defined experiment configuration step by step and place the results into a parsable JSON result file.

The *top* program was used to track the system utilization (see */utils/master.sh*).

7.2 Worker Experiments

7.2.1 Without work stealing. The experiment without work stealing can be reproduced by running the worker in the main repository, *go run cmd/worker/main.go [server-url] [num-cores]* and modifying the *num-cores* parameter. The worker will only run the experiments with the supplied number of cores. A valid Coordinator has to be running on the supplied *server-url* and a job has to be submitted through the CLI.

7.2.2 With work stealing. The work-stealing method can be run via the *cargo run –release –bin fasta_parser.rs* command, where the first two arguments denote the query and database filenames, respectively, and the third supplied argument is the number of threads to use.

7.3 Algorithm

7.3.1 SIMD lane scalability. The results of the SIMD lane scalability experiment mentioned in section 4.3.1 can be reproduced by running *cargo run –release –bin simd_lanes* on the Rust experiment project, located in the *algorithm-experiments* directory. This produced the artefacts used in *plotting/algorithm/simd_lanes.py* which can generate figure 11. The results in figure 11 and the underlying artefacts were produced on an Amazon AWS EC2 C5.large machine.

7.3.2 Target Scalability. The algorithm experiments mentioned in section 4.3.2 can be trivially reproduced by running *cargo run –release –bin target_scalability* on the Rust experiment project, located in the *algorithm-experiments* directory. This produces the artefacts used in *plotting/algorithm/target_scalability.py* which can generate figure 12.

A Competition Results

Table 1 presents the competition results. The mean and median *elapsed_time* and 1 standard deviation (all in milliseconds; rounded to 2 decimals) are included.

As described in section 4.1 the *elapsed_time* metric is the time between the starting of a job through the CLI and the retrieval of the alignment job's results (as the desired measurement for the competition). In other words, this is the time a user (e.g., a scientist) has to wait. Just like in section 4.1.1, 10 measurements were collected per configuration.

For the individual measurements and Python script (for mean, median, and std.dev.) check out the competition directory in the experiments' repository (section 7). Have a look at the *README.md* file there for more information regarding the contents of the subdirectories.

B Time Sheets

See table 2 for the team's time spent on this lab assignment.

[5]Project source code: github.com/Noorts/DLSA

[6]Project experiment setup and results archive: github.com/Noorts/DLSA-Experiments

Table 1. Competition Results (values in milliseconds)

Dataset	# Workers	Mean	Median	Std.dev.
XL	1	136049.8	136504.0	9316.51
	2	119693.1	120644.0	7258.93
	4	116062.7	115934.0	7796.47
	8	109493.4	110579.0	5204.03
L	1	5655.9	5671.0	135.14
	2	5472.4	5511.5	202.71
	4	5479.9	5545.5	265.33
	8	5521.5	5566.5	171.13
M	1	707.1	630.0	122.01
	2	635.2	634.0	14.58
	4	675.9	654.5	71.36
	8	774.0	699.5	132.55
S	1	308.6	277.0	84.28
	2	273.4	274.0	2.06
	4	300.2	272.0	84.95
	8	273.7	273.0	2.45

References

[1] S. Ranganathan, K. Nakai, and C. Schonbach, *Encyclopedia of bioinformatics and computational biology: ABC of bioinformatics*. Elsevier, 2018.

[2] J. Slack, "Chapter 7 - molecular biology of the cell," in *Principles of Tissue Engineering (Fourth Edition)*, fourth edition ed., R. Lanza, R. Langer, and J. Vacanti, Eds. Boston: Academic Press, 2014, pp. 127–145. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780123983589000070

[3] P. Romero, "Bioinformatics: Sequence and Genome Analysis," *Briefings in Bioinformatics*, vol. 5, no. 4, pp. 393–396, 12 2004. [Online]. Available: https://doi.org/10.1093/bib/5.4.393-a

[4] V. O. Polyanovsky, M. A. Roytberg, and V. G. Tumanyan, "Comparative analysis of the quality of a global algorithm and a local algorithm for alignment of two sequences," *Algorithms for molecular biology*, vol. 6, no. 1, pp. 1–12, 2011.

[5] H. R. Shakir, "A color-image encryption scheme using a 2d chaotic system and dna coding," *Advances in Multimedia*, vol. 2019, pp. 1–13, 2019.

[6] T. Smith and M. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195–197, 1981. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0022283681900875

[7] L. Bergroth, H. Hakonen, and T. Raita, "A survey of longest common subsequence algorithms," in *Proceedings Seventh International Symposium on String Processing and Information Retrieval. SPIRE 2000*, 2000, pp. 39–48.

[8] O. Gotoh, "An improved algorithm for matching biological sequences," *Journal of Molecular Biology*, vol. 162, no. 3, pp. 705–708, 1982. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0022283682903989

[9] A. Wozniak, "Using video-oriented instructions to speed up sequence comparison," *Bioinformatics*, vol. 13, no. 2, pp. 145–150, 04 1997. [Online]. Available: https://doi.org/10.1093/bioinformatics/13.2.145

[10] T. Rognes and E. Seeberg, "Six-fold speed-up of Smith–Waterman sequence database searches using parallel processing on common microprocessors ," *Bioinformatics*, vol. 16, no. 8, pp. 699–706, 08 2000. [Online]. Available: https://doi.org/10.1093/bioinformatics/16.8.699

[11] M. Farrar, "Striped Smith–Waterman speeds database searches six times over other SIMD implementations," *Bioinformatics*, vol. 23, no. 2, pp. 156–161, 11 2006. [Online]. Available: https://doi.org/10.1093/bioinformatics/btl582

[12] B. Alpern, L. Carter, and K. Su Gatlin, "Microparallelism and high-performance protein matching," in *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '95. New York, NY, USA: Association for Computing Machinery, 1995, p. 24–es. [Online]. Available: https://doi.org/10.1145/224170.224222

[13] C. P. E. M. H. J. R. K. Rahn R, Budach S, "eneric accelerated sequence alignment in seqan using vectorization and multi-threading." *G Bioinformatics*, 2018.

[14] S. M. Larson, C. D. Snow, M. Shirts, and V. S. Pande, "Folding@ home and genome@ home: Using distributed computing to tackle previously intractable problems in computational biology," *arXiv preprint arXiv:0901.0866*, 2009.

[15] V. A. Voelz, V. S. Pande, and G. R. Bowman, "Folding@ home: achievements from over twenty years of citizen science herald the exascale era," *Biophysical Journal*, 2023.

[16] F. Hufsky, K. Lamkiewicz, A. Almeida, A. Aouacheria, C. Arighi, A. Bateman, J. Baumbach, N. Beerenwinkel, C. Brandt, M. Cacciabue *et al.*, "Computational strategies to combat covid-19: useful tools to accelerate sars-cov-2 and coronavirus research," *Briefings in bioinformatics*, vol. 22, no. 2, pp. 642–663, 2021.

[17] C. Wang, Z. Liu, Z. Chen, X. Huang, M. Xu, T. He, and Z. Zhang, "The establishment of reference sequence for sars-cov-2 and variation analysis," *Journal of medical virology*, vol. 92, no. 6, pp. 667–674, 2020.

[18] C. Yin, "Genotyping coronavirus sars-cov-2: methods and implications," *Genomics*, vol. 112, no. 5, pp. 3588–3596, 2020.

[19] A. Haileamlak, "Pandemics will be more frequent," *Ethiopian Journal of Health Sciences*, vol. 32, no. 2, p. 228, 2022.

[20] Z. Xia, Y. Cui, A. Zhang, T. Tang, L. Peng, C. Huang, C. Yang, and X. Liao, "A review of parallel implementations for the smith–waterman algorithm," *Interdisciplinary Sciences: Computational Life Sciences*, pp. 1–14, 2021.

[21] F. Zhang, X.-Z. Qiao, and Z.-Y. Liu, "A parallel smith-waterman algorithm based on divide and conquer," in *Fifth International Conference on Algorithms and Architectures for Parallel Processing, 2002. Proceedings.*, 2002, pp. 162–169.

[22] A. Khajeh-Saeed, S. Poole, and J. Blair Perot, "Acceleration of the smith–waterman algorithm using single and multiple graphics processors," *Journal of Computational Physics*, vol. 229, no. 11, pp. 4247–4258, 2010. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0021999110000823

[23] P. Zhang, G. Tan, and G. R. Gao, "Implementation of the smith-waterman algorithm on a reconfigurable supercomputing platform," in *Proceedings of the 1st International Workshop on High-Performance Reconfigurable Computing Technology and Applications: Held in Conjunction with SC07*, ser. HPRCTA '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 39–48. [Online]. Available: https://doi-org.vu-nl.idm.oclc.org/10.1145/1328554.1328565

[24] B. Xu, C. Li, H. Zhuang, J. Wang, Q. Wang, and X. Zhou, "Efficient distributed smith-waterman algorithm based on apache spark," in *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, 2017, pp. 608–615.

[25] D. Szajda, M. Pohl, J. Owen, B. G. Lawson, and V. Richmond, "Toward a practical data privacy scheme for a distributed implementation of the smith-waterman genome sequence comparison algorithm." in *NDSS*, 2006.

[26] Go, "The go programming language specification," Aug 2023. [Online]. Available: https://go.dev/ref/spec

[27] H. Bal, D. Epema, C. de Laat, R. van Nieuwpoort, J. Romein, F. Seinstra, C. Snoek, and H. Wijshoff, "A medium-scale distributed system for

Person	Total Time	Think Time	Dev Time	Xp Time	Analysis Time	Write Time	Wasted Time
Niclas	140	20	75	0	0	35	10
Paul	103	20	40	0	0	35	8
Haraldur	129	30	50	8	5	26	10
Simon	149	64	34	25	5	20	1
Daniël	180	20	100	20	15	15	0
Enrico	90	20	55	0	0	10	5

Table 2. Time Allocation for Assignment Completion

computer science research: Infrastructure for the long term," *Computer*, vol. 49, no. 05, pp. 54–63, may 2016.

[28] DAS, 2012. [Online]. Available: https://www.cs.vu.nl/das5/clusters.shtml

[29] "Rayon." [Online]. Available: https://github.com/rayon-rs/rayon

[30] Python, "Page," Dec 2020. [Online]. Available: https://wiki.python.org/moin/GlobalInterpreterLock

[31] Go. [Online]. Available: https://github.com/golang/go/blob/6db1102605f227093ea95538f0fe9e46022ad7ea/src/net/http/server.go#L3285

[32] "Distributed systems lecutre notes 2019-2020," 2019.