Universiteit Leiden
The Netherlands

Opleiding Informatica

Optimization of Metagenomics Analysis

using Distributed Computing on LLSC

M.A. Voogt

Supervisors:
Dr. K.F.D. Rietveld & Prof.dr.ir. F.J. Verbeek

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

21/08/2018

Abstract

In metagenomics the analysis of DNA is very computationally intensive. In this thesis we will investigate using a computer cluster to speed up this process. We will start by investigating ways to parallelize typical software tools for metagenomics analysis, i.e. NCBI-BLAST, uSearch and ITSx, on the LIACS Life Science Cluster (LLSC) and then analyze their performance. With this analysis we try to find an optimum configuration in terms of nodes and processes to run these packages.

Contents

List of Figures

1 - Introduction

In the eld of metagenomics there is an increasing necessity for the fast processing of DNA data recovered from environmental samples. The amount of available data for bioinformatics is growing faster than the amount of transistors in a chip [1]. Therefore a possible solution would be to parallelize the processing of this data. A lot of these programs are already capable of using extra threads in a multicore CPU to parallelize to some degree, but a further improvements could be achieved by using a computer cluster with multiple CPUs.

1.1 Metagenomics and Fungi

Metagenomics is the analysis of genomes recovered from an environmental sample. First DNA is extracted from the sample and then sequenced. It is then cleaned and ltered to prepare it for further analysis. This ltering and cleaning step is important because not all DNA from a sequence can be used to identify the species from the sample. After ltering the data it can be compared to DNA from known species for the analysis of gene diversity or possible discovery of new species.
In this research project we will only use fungal data. Compared to other kingdoms, the fungal kingdom is less discovered and knowledge about their diversity and global distribution is still lacking due to the their sizes and locations. Therefore metagenomics is very important to the discovery of fungi and their symbiotic role in plant communities.

1.2 Research scope

For this research project we will be looking at three di erent software packages typically used in the metagenomics analysis of fungi: NCBI-BLAST, uSearch and ITSx. These software packages are used at di erent stages of the metagenomics process. ITSx is used to identify the ITS DNA regions from larger sequences, and NCBI-BLAST and uSearch can be used to compare the found DNA to already known DNA. The ITS sequences are speci c parts of DNA important for the identi cation of fungi in fungal samples [11]. We will start with enabling parallel execution of these processes, and analyzing the parallel performance of these packages on the LIACS Life Science Cluster (LLSC), a computer cluster from the Leiden Institute of Advanced Computer Science (LIACS) used for bioinformatics purposes.
In this research we will only look at DNA sequences. And therefore we will only look at the four letter DNA alphabet.

1.3 CPU versus GPU-optimization

In our initial outline for this project we intended to also include GPU performance. There exist various versions of NCBI-BLAST functions that are optimized to make use of GPU hardware. Various papers have been written on the topic [2] [3], and most of the existing software versions are written for the acceleration of protein searches [4] [5]. But as stated before we will only be looking at DNA searches and the GPU accelerated versions for those functions have not been updated for years and would not compile with the current CUDA versions.
However the fact that there are various research papers written on this topic and that the GPU accelerated versions for protein searches work would suggest that GPU acceleration is e ective, but only for those protein searches considering the current state of the other versions. This might have to do with the fact that the protein

alphabet is larger compared to the DNA alphabet, which could be more suitable for the large parallel processing power of a GPU. Therefore we dropped our initial idea of taking GPU computing into account and for this research we focus on the optimization of CPU computing.

1.4 Research Questions

The aim of this research project is to try nding an optimal way to distribute the computational load over the computer cluster. To do this we will be looking at di erent aspects: the best way to divide the load over multiple processes, a comparison of this to the built-in multithreading options, how well the computational load scales over the nodes of the cluster and how well this scales with le size. The nal goal is to nd an optimal con guration in terms of nodes and processes per node to be used to process data as fast as possible. This is important to process the vast amount of data in an e cient and time constrained manner.

2 - Materials

In this chapter we describe the hardware and software used in this research project.

2.1 LLSC

The LIACS Life Science Cluster (LLSC) is a computer cluster for bioinformatics software. It has three user nodes, 20+ compute nodes and a file server. Each compute node consists of two Intel Xeon dual- or quad-core processors with 16GB RAM. The separate file server has 36TB storage. The nodes and file server are connected using Gigabit Ethernet. The LLSC currently uses the TORQUE job scheduler [6] to allocate the computational tasks, but a future upgrade to the SLURM Workload Manager [7] has been planned. To ensure consistent performance measurements we will only be using the nodes that contain two Intel Xeon 5150 dual core processors, given that Most of the currently operational nodes in the cluster are of this type.

Figure 2.1: The LLSC

2.2 Software

In this section we will brie y describe three software packages currently in use in the metagenomics pipeline. We will be con guring these packages for use on the LLSC.

2.2.1 NCBI-BLAST

The Basic Local Alignment Search Tool (BLAST) is a heuristic algorithm used for comparing biological sequence information like DNA nucleotide and protein sequences. It compares input strings of sequences to a database and computes and returns the most similar sequences in the database. The algorithm was created in 1990 by S. Altschul et al. [13] and it has shown an indispensable tool in the analysis of genomic and protein sequences. The algorithm is useful for discovering the purpose of unknown DNA or protein sequences by comparing it to known sequences or the classi cation of DNA from di erent sources. You could basically say that it is a string comparison algorithm. The current BLAST program from the NCBI is a software suite with various functions:

- Nucleotide-nucleotide BLAST (blastn)
- Protein-protein BLAST (blastp)
- Position-Speci c Iterative BLAST (PSI-BLAST) (blastpgp)
- Nucleotide 6-frame translation-protein (blastx)
- Nucleotide 6-frame translation-nucleotide 6-frame translation (tblastx)
- Protein-nucleotide 6-frame translation (tblastn)
- Large numbers of query sequences (megablast)

As mentioned earlier, this research is focused in the analysis of genomic sequences taken from soil samples. Therefore we will only be looking at DNA sequences so we will only be using blastn. We will be using version 2.7.1 which was the latest version available at the start of this research.

2.2.2 uSearch

uSearch [15] is a software package that can perform the same analysis as NCBI-BLAST, but is advertised as and proven to be much faster than the BLAST algorithm. It can achieve this speedup compared to NCBI-BLAST by only seeking the best hits instead of all hits. In this research we will be comparing the performance to NCBI-BLAST on the computer cluster. We will be using version 10.0.240 32bit. The 32-bit version of uSearch is free, and the 64-bit version is available for purchase.

2.2.3 ITSx

ITSx [14] is a software package used to extract ITS1 and ITS2 subregions from raw ITS sequences using Hidden Markov Models. The ITS sequences are the most important part of DNA for fungal research since they are used for the identi cation of fungi and are seen as the barcode sequences for fungi [11]. Figure 2.2 shows the location of these regions in DNA. These output sequences can be used as input queries for the BLAST software. ITSx does not use a database, and is therefore lower on IO and more heavy on the computational side compared to BLAST or uSearch. This, combined with the fact that the processing of large amounts of data can take up to multiple days on a standalone workstation, makes it a very good candidate for parallelization on the LLSC. We will be using version 1.1b1.

Figure 2.2: ITS1 and ITS2 regions in DNA

2.3 data sets for Testing

To analyze the performance of the software on the LLSC, a workload is needed that is representative of queries that will be requested in the future. Therefore we used data sets containing DNA data sequences from a large fungal study.

2.3.1 BLAST and uSearch Database

As mentioned above, BLAST and uSearch use a database of sequences to which the input query sequences will be compared. The Fungi database we used is the UNITE[12] Fungal reference database from 10-10-2017. This was the newest version available when this research was started. It is a 504Mb fasta le with 777046 Fungi DNA sequences.
This fasta le needs to be compiled into a BLAST database for NCBI-BLAST, and a uSearch database for uSearch, before it can be used in the \blasting" process.

2.3.2 Query Files for BLAST and uSearch

For input, we used two query les representative of the workload that is to be run on the computer cluster. A smaller one to use for development and initial tests and a bigger one to look at performance scale with le size in comparison to the smaller one.
These les are subsets from a recent large global fungi study[8]. The raw data was retrieved from the NCBI Sequence Read Archive (SRA)[9]. After retrieval this data was cleaned for use with the BLAST and uSearch software. This study was chosen because it was a large global study containing diverse samples from many di erent environments.
In metagenomics les tend to be very big and can take many hours, or sometimes days to process, the test les used are random subsets from this cleaned data to speed up the process. The smaller one was a 134Kb fasta le with 462 sequences, the bigger one was a 215K fasta le with 537 sequences.

2.3.3 Test Files for ITSx

For ITSx we used two di erent test les as well with the same purpose of having a smaller one for initial development and testing, and a bigger one to check the performance scale over le size. These test les are subsets from the same study we mentioned earlier, but were taken from an earlier step in the cleaning process since ITSx is used in an earlier stage in the metagenomics pipeline. The small le is a 4.1M fa le with 8170 sequences. The big le is a 8.1M fa le with 19000 sequences.

2.4 Implementation Tools

To implement the job scripts we used job scripts written in bash and Python Clustershell for load distribution over the cluster. We used Python 2.7.3 with Clustershell 1.7.3[10] and GNU bash 4.2.37(1)-release. Clustershell is a Python framework used for parallel command execution on computer clusters.

3 - Methodology and Implementation

3.1 Load Distribution

All software packages used in this research have built in multithreading options to allow the use of more threads on a single machine, but are not out of the box capable of running on the multiple nodes of a computer cluster. Therefore we need to write job scripts that can equally divide workload and then run the software on the di erent nodes of the cluster. To distribute the computational load for BLAST and uSearch over a computer cluster there are three options:

- Split the input queries, distribute them over the nodes, \blast" them against the full database and append all results
- Split the database, \blast" the full set of queries against each part of the database on the nodes and merge sort all results
- Use a combination of these two methods to \blast" a part of the queries against a part of the database

Splitting the workload can be done in these ways since the queries are independent of each other. It works the same for the database since before compilation it is structured the same way as the input queries.

Figure 3.1 is a work ow diagram for processing les using the rst method. This process consists of rstly a sequential part in which the input query is split into equally sized parts. The amount of parts equals the amount of reserved nodes the amount of processes per node. Then starts the second part of the process. This part is executed in parallel on all the nodes. Firstly each process on each node copies it's own part of the query into local storage, and then runs the software on this part. The output is written to local storage as well, and if the software is nished, this output is moved onto the le server. After this the nal part of the process starts. This happens sequentially like the rst part. The outputs get appended into one large output le which is the total result of the process.
For this method a compiled version of the database is put into local storage on each node beforehand to speed up IO. This means that every time a new version of the database is released, it has to be recompiled and copied onto every node. The e ciency of this would depend on how often a new version of this database is released.

Figure 3.2 is a work ow diagram for processing queries by splitting the database instead of the query. Just like the rst option it consists of a sequential part, a parallel part and then a sequential part again. Firstly the database le is split just like the query mentioned above. Then in the parallel part the database parts are copied to the nodes and are compiled into separate databases for each process on each node. Then the query le is copied onto the nodes for each process and the software is run in parallel on them for each database. Then the outputs are moved onto the le server. Finally the outputs have to be merge sorted into one large output le. This cannot be appended because the results from each individual query have to be grouped together.
The big drawbacks from this setup are that the merging of results will take more time compared to the simple appending, and the compiling of the databases also costs time.

The original experiment would be to try the rst two options and compare them. If they are approximately equally fast the third option would be experimented with. But if one of the rst two options would be much faster when compared to the other, it would not be a good idea to combine them because the slower option would slow the faster option down.

ITSx does not use a database, so the only way do distribute the computational load in this case is to split and distribute the input sequences.

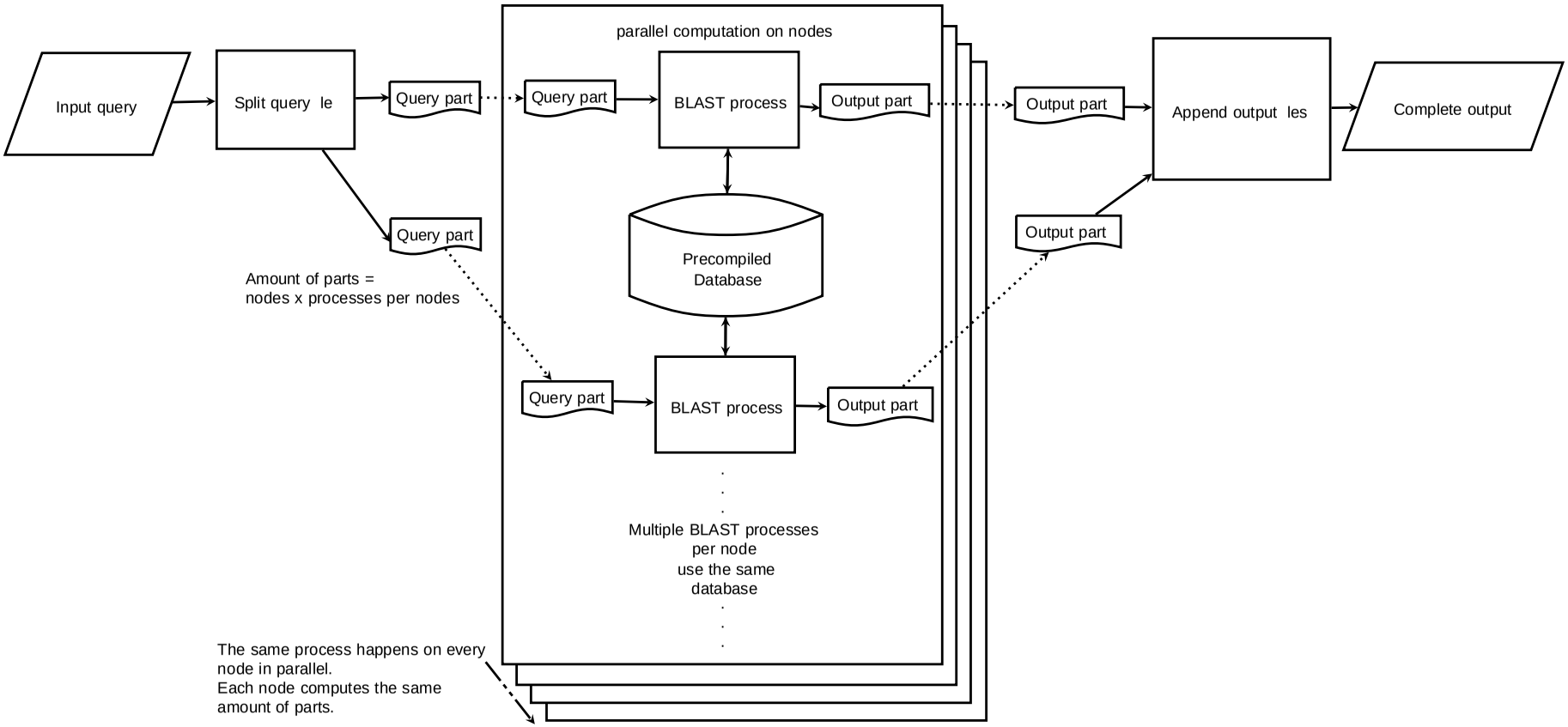

Figure 3.1: Splitting the query le for work distribution

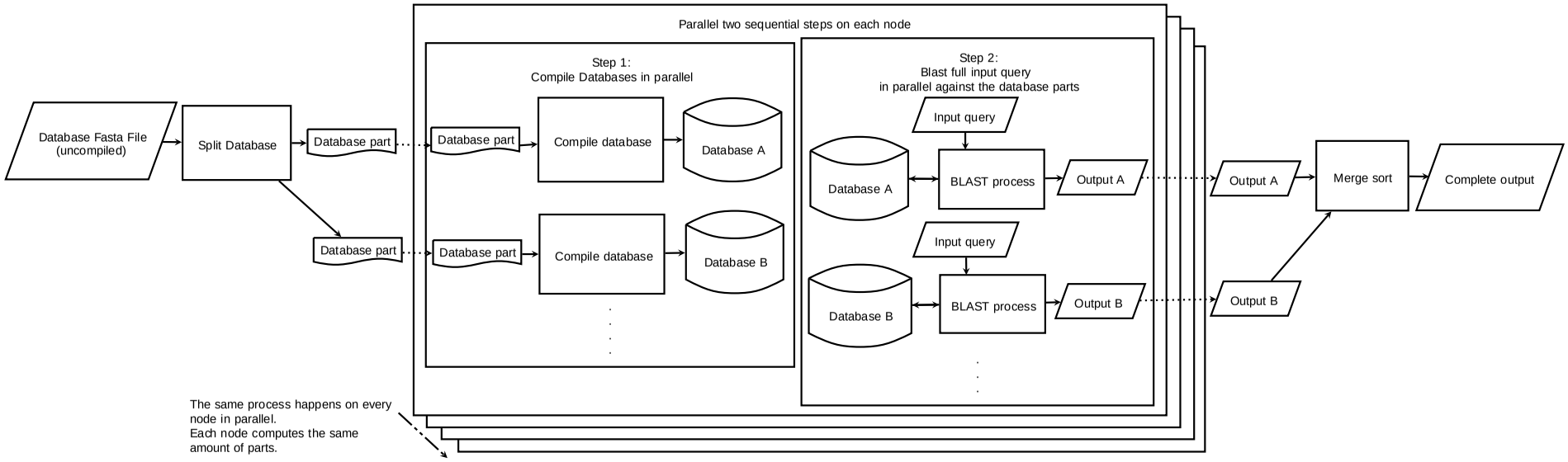

Figure 3.2: Splitting the database le for work distribution

3.2 Implementation

To implement these methods we need three things: a job script, a Python script, and a wrapper script.

The job script contains information about the amount of nodes and processes per node needed for the job. This information is used by the job scheduler to allocate nodes. The job script also contains all parts of the job that do not need to be executed in parallel, in this case the splitting of the data beforehand, and the merging of the parts of the results. It also records time stamps for each of these steps and calculates the total execution time. This is just a normal bash script with special variables containing information for the job scheduler. Appendix A contains a sample job script.

To split the files we use a separate split script which calculates the amount of lines per part and then splits the input fasta file per two lines using split. This script can be used in all situations where the input fasta file is in the standard description, sequence, description, sequence format. The input for ITSx is sadly not in this format and contains additional newlines in the sequence. To remove this we have a different separate script which uses a sed script.

After the data is split, computational processes for each of the data fragments need to be started on the allocated nodes. To do so we use a Python script written with the Clustershell Python module. This script is used to distribute the load over the allocated nodes. To do this, it uses the Clustershell library. It reads the information from the job scheduler about which nodes have been allocated for the job and starts the wrapper script for each process on each of those nodes in parallel. When all wrapper scripts finish, it prints output or errors and we return to the job script.

The wrapper script contains all parts of the job that need to be executed in parallel. In this takes care of copying the parts of the input and output and running the software. The wrapper is written in bash. The software is started with time -v for useful information about CPU-usage and execution time.

To make it easier to write these scripts, we also have a script that automatically generates these scripts. To do so it needs a name for the job, an input file, a database and the amounts of nodes and processes per node. After generating these scripts, you only need to submit the job script to the job scheduler to start it. These script generators can be used to easily integrate the process into a larger process or system.

To eliminate most read/write access time for the processes on the nodes, we have a compiled version of the database stored on the local disk in each node and the wrapper script also copies the input file onto this disk and writes output to this disk.

4 - Experiments & Results

First we will look at the baseline results. Then we will take a look at all experiments and results on the cluster from NCBI-BLAST and uSearch and compare them. After that we will look at the experiments and results for ITSx.

4.1 Experiment Setup

For all experiments we will run them three times to make sure the page caches of the nodes are lled with relevant data. These \warm" caches will speed up the computational process since there will be less cache misses. The di erence we measured between \warm" and \cold" caches can be as large as 5 seconds. For our measurements we will only be using results from runs with warm caches for consistency, because cold cache experiments may show irregular performance, and in the future there will hopefully be enough \blast" activity on the cluster to ensure that the caches are almost always warm.

All timings in the results are averages of three runs of the experiments. Times of individual processes are measured using the time command, all other times were measured using bash date. In all cases there was very little or no deviation in timing between these three runs. This is not really surprising considering all software uses deterministic algorithms. If the input is the same, then the output will be the same too, and with the same hardware the execution time is also the same. In between runs the output from the algorithms itself had to be deleted because it would take longer if an output le had to be overwritten.

4.2 Baseline Results

Our baseline experiments were running the query les on one node with one process with one thread. This should be equivalent to running the le on a single workstation with no parallelization.
We will start by comparing NCBI-BLAST and uSearch:

	NCBI-BLAST	uSearch
134Kb Test le	50s	24s
215Kb Test le	3384s	89s

Table 4.1: Baseline results for NCBI-BLAST and uSearch (n=3)

From the baseline results in Table 4.1 we can already see that uSearch is faster than NCBI-BLAST, especially with larger les. We can also see that processing times for both NCBI-BLAST and uSearch increase with le size, but they are not proportional to the le size. The larger test le is not even twice as large as the smaller one, but processing times are in both cases more than twice as big. This is especially visible with NCBI-BLAST.

The baseline results for ITSx are as follows:

In Table 4.2 we can see that processing time for ITSx probably scales proportionally with with le size.

	ITSx
4.1Mb Test le	2097s
8.1Mb Test le	4166s

Table 4.2: Baseline results for ITSx (n=3)

4.3 Load Distribution

In chapter three we described two di erent methods for load distribution. To compare these we tested them on a single node with two to four processes per node, and on eight nodes as well to see if the situation would change for larger amounts of processes.

(a) Single node (b) Eight nodes

Figure 4.1: Splitting the database (red) vs. splitting the query (blue) with 2-4 processes. (n=3)

(a) Trend line for the data from Figure 4.1a (b) Trend line for the data from Figure 4.1b

Figure 4.2: Trend lines for the data from Figure 4.1

In the results from Figures 4.1 and 4.2 and we can see that splitting the database is slower compared to splitting the queries. The computational time of just \blasting" the full query set against a part of the database was already slower compared to the total process time of the rst option. Additionally, the process time does not signi cantly decrease when the database is further split over three or four processes per node. Because this computation phase already took more time, the additional merge sort that has to be performed for this setup was never implemented. Because of this result the third option was never tried.

In conclusion: all following experiments we used splitting the input as the method to distribute the load over the nodes of the computer cluster.

4.4 NCBI-BLAST and uSearch

In this section the performance of NCBI-BLAST and uSearch will be compared. We conducted various experiments for di erent aspects of the software, cluster and data in order to determine an optimal cluster setup.

4.4.1 Performance Scale for Multithreading with parts of the data set

BLAST and uSearch have built in options for multithreading. uSearch actually automatically uses the amount of threads of the CPU it is running on, if a maximum amount of threads was not set beforehand.
In this experiment we try to nd out the e ect of using the built in multithreading on the execution time and compare it with the execution time for using multiple processes with a divided load. For the experiment we will use a node in the cluster with two Xeon x5450 quad-core CPUs. We will test the e ectiveness of 1 to 8 threads for the complete set of input sequences versus dividing the input sequences in 1 to 8 parts. We expect to nd an optimum number of threads because starting up too many threads would create too much overhead and negatively impact the process time. Furthermore we also hope to nd out if it is actually worth it to multithread, because using more threads would also mean we can use less processes on a single node since the multithreading would take threads that could also be used by more processes on a node. So we are comparing using more threads to increase computing power per process to decreasing workload per process by having more processes and therefore splitting the input data set further.
We will be using the smaller test les for this experiment. The measured times are just the processing times without the added time for the splitting of data or the merging of results.

(a) Individual data for NCBI-BLAST

(b) Individual data for uSearch

Figure 4.3: Dividing the load in 1-8 parts with one thread (red) vs. Multithreading the full load (blue) with 1-8 threads. (n=3)

Figures 4.3 and 4.4 contain the multithreading test results. We can clearly see that for uSearch at a certain point adding threads is causing more overhead which results in a higher processing time, while for NCBI-BLAST such an optimum was not yet found, but cannot be far away. We can also see that adding more threads beyond four does not result in a decrease in processing time for NCBI-BLAST. The built-in multithreading for uSearch is very good and almost equally fast as splitting the data for up to four threads. Beyond four threads, both options slow down and therefore we can see an optimum at four threads or processes per node. This is good news since most nodes in the LLSC only have four threads available.

Here we can conclude that it is more e ective to use the multiple cores on the processors in the nodes for extra processes instead of extra threads. This is something a computer cluster is very well suited for.

(a) Trend line for the data from Figure 4.3a (b) Trend line for the data from Figure 4.3b

Figure 4.4: Trend lines for the data from Figure 4.3

4.4.2 Performance Scale over Nodes and Processes

Using the load distribution method mentioned above we will try to find an optimal load distribution over the nodes in the cluster. The main questions to be answered with these experiments will be about the optimal amount of cluster nodes that should be used and the amount of processes that will run on these nodes. These experiments will be run using the smaller test files.
For these experiments we will make sure BLAST and uSearch are set to only use one thread per process, since this was proven to be faster in the experiment before.
As mentioned before we will only use the nodes of the cluster that contain two Intel Xeon 5150 dual-core CPU's. We will run experiments splitting the input file with the amounts of nodes varying from 1 to 8 and the amount of processes per node from 2 to 4. We expect some sort of optimal distribution to be found here. We will also do an extra experiment with 12 nodes to check if an optimum was found.

We expect the total process time to drop since we are decreasing the computational load per process, and these processes are running in parallel. Each process will do computations for $\frac{1}{(usednodes \quad processespernode)}$ th of the input sequences. We hope to find an optimum because if this part of the input becomes too small, it will generate more overhead than actual computation time, and will be slower in total in compared to when we use a bigger part of the data set.

From the results, the shape of the resulting Trend line will be the most interesting in order to find an optimum, instead of all the exact times.

(a) Individual data for NCBI-BLAST (b) Individual data for uSearch

Figure 4.5: Dividing the workload over 1-12 nodes with 2 (red), 3 (green) and 4 (blue) processes per node. (n=3)

(a) Trend line for the data from Figure 4.5a (b) Trend line for the data from Figure 4.5b

Figure 4.6: Trend lines for the data from Figure 4.5

In Figure 4.5 and 4.6 we see that uSearch is faster compared to NCBI-BLAST. NCBI-BLAST will always have a shorter processing time if you increase the amount of parallel processes. But for uSearch we see a strange e ect when we have more than two processes per node. All con gurations with two processes per node are faster compared to their versions with three and four processes per node. There is clearly an optimum there. In the output from the time command we can see that the amount of involuntary context switches when running more than two processes per node is higher compared to when running two processes per node.
We can also see that the di erence in time between di erent amounts of processes becomes smaller when we increase the amount of nodes, and at some points the di erence in time is almost indistinguishable. We hope that in the tests for scaling with le size we can see the di erence more clearly. The speedup when using more nodes also becomes less signi cant. In comparison to the baseline result NCBI-BLAST had a speedup of 8.33 for both running on 8 and 12 nodes with four processes per node. Adding 16 extra processes did not give us a faster process time. For uSearch the speedup when running on 8 and 12 nodes with two processes per node are respectively 4.8 and 6 with a time di erence of one second. This extra one second speedup is also insigni cant considering we are adding 8 extra processes. In conclusion we would not recommend using more than 8 nodes for this le size.

4.4.3 Explanation for uSearch behaviour

The reason uSearch slows down when running more than two processes on a single node could be because the processes are in the way of each other when doing IO. To check this hypothesis, we conducted an experiment in which we monitor disk usage. If uSearch uses the disk a lot, the multiple uSearch processes might slow each other down when reading and writing at the same time, due to reaching maximum IO bandwidth. This is because we expect that in the process there is rst a lot of IO to read the database, and then a lot less IO when the computation happens. If this IO phase would take longer in cases with more processes per node then it would mean that the processes are in the way of each other. If we do not see di erences in IO, then the explanation should be found in the computational part of the process. To monitor this we will start up IOSTAT in the main job script before the Python-script is called to monitor the IO in the disk in the node on which the job gets started. We set IOSTAT to output disk usage to a le every second, and hopefully this data can give us an answer to what is going on.

The output from IOSTAT showed us that there was no disk IO. But since uSearch still needs the database, it must be cached in RAM. To verify this we checked the amount of data in the caches on the node using free. This showed that there was data cached in RAM roughly the size of the database. To verify if it was actually the database, we forcefully emptied the RAM caches, and ran the experiment again. This time there was an average of 250MB/s read and checking the RAM again with free showed us that the same amount of data as before was cached again. All this would mean that the database is cached in RAM after running uSearch with this database, and therefore IO to disk cannot be the bottleneck that would explain the behaviour of uSearch.

If it is not IO, then the next option to explain the behaviour of uSearch could be the use of shared caches in

	DB cached in RAM	Emptied RAM cache
Mb read/s	0	250

Table 4.3: Disk monitoring results

the processors. The Intel Xeon 5150 processor has a shared L2 cache. If there are multiple processes running on the same processor they have to share this cache and can be hindering each other. To test this we ran an experiment with three processes on a single node, where we force a process to a single designated core on a separate processor. This way one processor will have one process running on it, and the other will have two processes. The two processes on a single CPU have to share cache and therefore we expect that these processes each will have approximately similar execution times, but both slower compared to the single process on the other CPU. We also ran the experiment with four processes on a single node, and compared the speed of these processes to the speed of the other two processes on a single CPU. The results were as we expected. The single process on a CPU was approximately twice as fast as the processes that had to share cache. Those processes were all equally fast. These execution times are stated in Table 4.4.

	CPU 1		CPU 2	
3 processes	2	1.05 min	1	33 s
4 processes	2	1.05 min	2	1.05 min

Table 4.4: Locking processes on two CPUs

From these small experiments we can understand that uSearch runs faster when it has more cache available and it is likely the reason why running uSearch with more than two processes per node is slower.

4.4.4 Performance Scale over file size

There are many more data sets that need to be processed and many of them are larger than the test files used for development. Therefore we also need to see how performance scales if we increase the input file size. For these experiments we will repeat the same Performance Scale over Nodes and Processes for the bigger test files and compare the results. We hope to see a linear time increase with file size. We also hope to see that since we are comparing NCBI-BLAST to uSearch, they scale with input file size in a similar way.

(a) Individual results for NCBI-BLAST (b) Individual results for uSearch

Figure 4.7: Dividing the big file over 1-12 nodes with 2 (red), 3 (green) and 4 (blue) processes per node. (n=3)

The results in Figure 4.7 and 4.8 should be compared to the results from the previous experiment. The graphs here have almost the same shape as the versions with the smaller files, but these are shifted to higher higher values on the y-axis. For the smaller files we could see that at a certain point adding more nodes with processes would not be reducing processing time by a large amount. Here, since the files are larger, additional nodes have more effect. Speedups from baseline to 8 and 12 nodes for NCBI-BLAST are respectively 25.0 and 36.8. For uSearch they are 8.9 and 11.1. While for NCBI-BLAST this increase in speedup is significant, we must not

(a) Trend line for the data from Figure 4.7a (b) Trend line for the data from Figure 4.7b

Figure 4.8: Trend lines for the data from Figure 4.7

forget that it is still around 11 times as slow as uSearch if we compare them both running on 12 nodes, but uSearch with only 2 processes per node and NCBI-BLAST 4.
From this comparison we can also clearly see that the le size scaling behaviour looks the same as what we could see from our baseline experiment. uSearch scales much better with le size compared to NCBI-BLAST. Note the scaling di erence on the y-axis between the results from NCBI-BLAST and uSearch. In conclusion we recommend using uSearch for larger les, since it is signi cantly faster compared to NCBI-BLAST. An optimal amount of nodes for uSearch would be around eight since the speedup from 8 to 12 is insigni cant for the added 8 processes, but using more could be more e cient for larger les. The optimum amount of processes per node is the amount of CPUs in the node to make sure the uSearch processes do not need to share cache.

4.5 ITSx

For ITSx the same experiments are conducted as for NCBI-BLAST and uSearch to nd an optimum in scaling. The di erence is that for most experiments we will use up to 14 nodes instead of 12 because the ITSx les are much larger and we might not nd an optimum within 12 nodes.

4.5.1 Performance Scale for Multithreading with parts of the data set

Figure 4.9b contains the results for the multithreading test for ITSx. This the same test as we did with NCBI-BLAST and uSearch on the 8 core machine.

As we can see in Figure 4.9 ITSx also processes faster with a smaller input load compared to using extra threads. And threads are also less e ective when using them on smaller input loads.

4.5.2 Performance Scale over Nodes and Processes

In Figure 4.10 we can see that ITSx bene ts a lot from using more processes and nodes. While e ectiveness goes down with more and more nodes, we would still recommend using more nodes because the input les are very large and therefore the computation time is large as well. This is also why we did not stop the experiment at eight nodes and continued up to 14. Using more than 14 nodes is not recommended at this le size since the time gain between 13 and 14 nodes is already quite small.

(a) Individual results (b) Trend line

Figure 4.9: ITSx: Dividing the load in 1-8 parts with one thread (red) vs. Multithreading the full load (blue) with 1-8 threads. (n=3)

(a) Individual results (b) Trend line

Figure 4.10: ITSx: Dividing the workload over 1-14 nodes with 2 (red), 3 (green) and 4 (blue) processes per node. (n=3)

4.5.3 Performance Scale over le size

Just like with BLAST and uSearch we should compare the results from Figure 4.11 to the baselines and the scaling results from Figure 4.10. We can see that the process time scales proportionally with le size, just as in the baseline results. The speedup from baseline to 14 nodes with 4 processes per node is 47.7 for the small le and 49.0 for the large le. These speedups are very similar and show us that larger les bene t more from large amounts of nodes. This is very important to know for ITSx since both our test les are only subsets from the data from the study. The actual les are signi cantly bigger but since ITSx scales proportionally we can predict the processing times for these.

(a) Individual results (b) Trend line

Figure 4.11: ITSx: Dividing the big le over 1-14 nodes with 2 (red), 3 (green) and 4 (blue) processes per node. (n=3)

5 - Conclusions & Discussion

5.1 Baseline Results

From these simple baseline results we can only draw one simple conclusion: in this case uSearch is faster than NCBI-BLAST and the di erence in speed becomes larger with le size. Neither of them seem to scale linearly. For ITSx we can only conclude that the scaling in process time with le size is linear or almost linear.

5.2 Workload Distribution

We tried two di erent methods of dividing the workload over multiple processes: Splitting the input query le and splitting the database. Splitting the query le was signi cantly faster. We can also guarantee that the required merge sort of the results would be more complicated and therefore slower compared to the simple appending of the results for splitting the query, even though we did not implement it.

5.3 Built-in Multithreading vs. More Processes

For NCBI-BLAST we can clearly see that the usage of the built in multithreading is less e ective compared to using the same CPU cores for extra processes. The same goes for uSearch up to four processes per node. After that the processes are slowing each other down due to having not enough cache space. In our case this works out well since most nodes on the LLSC only have four cores. We can also conclude that the multithreading functions for uSearch are much more e ective compared to those from NCBI-BLAST. We can conclude that for multithreading uSearch there is an optimum at around 4 threads. After that the overhead from generating these extra threads is only slowing down the process. For NCBI-BLAST we did not nd such an optimum but the e ectiveness of adding extra threads would goes down signi cantly per extra added thread.

For ITSx we can see that the built-in multithreading is quite good but splitting the input is still faster.

In conclusion: For all software it would be better to run separate processes on parts of the input instead of using multithreading on the LLSC. Having a process thread occupy a CPU core is less e ective compared to having another full process occupy that computational space. If the multithreading options were faster, a better option would be to use a single multi-CPU machine or running the processes multithreaded on the nodes.

5.4 Scaling over Nodes

For all three programs we can see that the e ectiveness of extra nodes and processes decreases when adding more and more. For NCBI-BLAST and uSearch we can see that there is very little di erence between 8 nodes and 12 nodes. This is because at that point the \blasting" of smaller les is only a bit faster but appending all the results takes a bit longer because there are more les to deal with. The overhead created by spawning all these extra processes could also be a reason why there is not much time gain. For ITSx we can see the same, but

the time gain in the range of 8-14 nodes is still a few seconds per step.

We can see in this test that uSearch is faster compared to NCBI-BLAST, but it runs faster with less processes per node. We advice to run it so that the individual processes on a node do not need to share CPU cache. If they do they get in the way of each other and slow each other down.

5.5 Scaling over file size

If we look at the base-case results we can see that NCBI-BLAST becomes a lot slower when we increase the file size. For a file just under twice as big, it takes just under 68 times as much processing time. This is not a constant factor when comparing the results from Figure 4.6a to those from Figure 4.8a. uSearch scales a lot better with file size. It is not a linear increase in time, but is much closer to that compared to NCBI-BLAST.

From our results we conclude that ITSx seems to scale linearly with file size. This makes it easier to predict how much time a file will take to process.

In conclusion: NCBI-BLAST was in all cases slower compared to uSearch. This was however what we predicted and what was proven in the research paper from R.C. Edgar[15]. The results from the file size scaling comparison were still very surprising.
For an optimum amount of nodes to run uSearch and NCBI-BLAST on, we would recommend around 8 nodes for these file sizes because using more than that does not result in a significantly larger speedup. But for files that are a lot bigger, more nodes should be considered. The amount of processes per node should be the amount of CPUs in the node for uSearch to make sure there is no need to share cache, and for NCBI-BLAST you can use as many processes as there are cores in the node. For ITSx it is possible to estimate the process time for your files since it scales proportionally to file size. For files larger than our biggest test file we would recommend using at least 14 nodes since larger files benefit more from the workload division. ITSx processes are long and the time gain is therefore much more needed. For ITSx you can use as many cores as there are in the node. These file size scaling tests are very important considering all used test files are only subsets from a large study. The actual files are much bigger, and will take much more time. But the processing time of these files on a cluster will be faster compared to running them on a single machine.

5.6 Future Work

For uSearch there should be more experiments done to explain the behaviour for multiple processes on a single node. Clearly the amount of cache available for each process has a big influence on performance. Comparing performance on different CPUs with different cache sizes should give a better idea how much faster uSearch could go if it had access to more cache. A full comparison to the 64-bit version of uSearch should also be made. Various other data sets should also be tested. We used only fungal DNA data, but tests could be done for other species, or for Proteins instead of DNA since uSearch and NCBI-BLAST can do much more.
Furthermore a new comparison to GPU optimized versions of these software packages should be made. Recently GPU speeds and capacity have been growing faster compared to CPU performance. Even though the smaller DNA alphabet might not be optimal for a GPU, a comparison between a computer cluster and a GPU using protein data could be interesting.

6 - Acknowledgements

I would like to thank my supervisors Kristian Rietveld and Fons Verbeek for their helpful advice and critiques, Irene Martorelli for providing me all the data and information about the whole metagenomics processes and Leon Helwerda for his help with trying to get the GPU versions running.

Bibliography

[1] A.J. Butte (2001) Challenges in bioinformatics: infrastructure, models and analytics. Trends Biotechnol., 19, 159{160.

[2] S. Suzuki, M. Kakuta, T. Ishida, Y. Akiyama, (2016) GPU-acceleration of sequence homology searches with database subsequence clustering. PLoS ONE 11(8): e0157338. doi:10.1371/journal.pone.0157338

[3] K. Zhao, X. Chu, G-BLASTN: Accelerating nucleotide alignment by graphics processors, Bioinformatics 30 (2014) 1384{1391

[4] S. Suzuki, T. Ishida, K. Kurokawa, Y. Akiyama (2012) GHOSTM: A GPU- accelerated homology search tool for metagenomics. PLoS ONE 7(5): e36060. doi:10.1371/journal.pone.0036060

[5] P.D. Vouzis, N.V. Sahinidis, GPU-BLAST: Using graphics processors to accelerate protein sequence alignment, Bioinformatics 27 (2011) 182{188

[6] http://www.adaptivecomputing.com/products/torque/

[7] https://slurm.schedmd.com/

[8] Fungal biogeography. Global diversity and geography of soil fungi. Tedersoo L, Bahram M, Põlme S et al., (2014) https://doi.org/10.1126/science.1256688

[9] https://www.ncbi.nlm.nih.gov/sra
https://trace.ncbi.nlm.nih.gov/Traces/sra/sra.cgi?study=SRP043706
https://www.ncbi.nlm.nih.gov//bioproject/PRJNA252425

[10] https://clustershell.readthedocs.io/en/latest/

[11] Conrad L. Schoch, Keith A. Seifert, Sabine Huhndorf, Vincent Robert, John L. Spouge, C. Andre Levesque, Wen Chen, Fungal Barcoding Consortium, Nuclear ribosomal internal transcribed spacer (ITS) region as a universal DNA barcode marker for Fungi, Proceedings of the National Academy of Sciences Apr 2012, 109 (16) 6241-6246; DOI: 10.1073/pnas.1117018109

[12] Kõljalg, Urmas, Nilsson, R. Henrik, Abarenkov, Kessy, Tedersoo, Leho, Taylor, Andy F. S., Bahram, Mohammad, . . . Larsson, Karl-Henrik, (2013), Towards a uni ed paradigm for sequence-based identi cation of fungi. Molecular Ecology, 22(21), 5271-5277.

[13] S.F. Altschul et al., Basic local alignment search tool, J. Mol. Biol. 215 (1990) 403{410

[14] Johan Bengtsson-Palme, Vilmar Veldre, Martin Ryberg, Martin Hartmann, Sara Branco, Zheng Wang, Anna Godhe, Yann Bertrand, Pierre De Wit, Marisol Sanchez, Ingo Ebersberger, Kemal Sanli, Filipe de Souza, Erik Kristiansson, Kessy Abarenkov, K. Martin Eriksson, R. Henrik Nilsson, ITSx: Improved software detection and extraction of ITS1 and ITS2 from ribosomal ITS sequences of fungi and other eukaryotes for use in environmental sequencing, Methods in Ecology and Evolution, 4: 914{919, 2013, DOI: 10.1111/2041-210X.12073

[15] R.C. Edgar, Search and clustering orders of magnitude faster than BLAST, Bioinformatics 26 (2010) 2460{2461

[16] S. Suzuki, M. Kakuta, T. Ishida, Y. Akiyama, (2014) GHOSTX: An improved sequence homology search algorithm using a query su x array and a database su x array. PLoS ONE 9(8): e103833. doi:10.1371/journal.pone.0103833

Appendices

A - Sample Job Script

```
#!/bin/sh
#PBS -l nodes=6:cpu-5150:ppn=4
#Special variables for the job scheduler
#This script reserves 6 nodes with 4 processes per node
#It only reserves nodes with 5150 CPUs
#PBS -N blast6-4_blast
#Job output name

starttime="$(date '+%s')"
#Register start time
#Timestamps are taken for different phases of the process
cd "$PBS_O_WORKDIR"

sss="$(date '+%s')"
./sss.sh /scratch/mvoogt/Test_ITS2.fasta 24
#Call to the split script
sse="$(date '+%s')"
echo "$(date) SSS IS DONE: $((sse - sss)) seconds taken."

. /opt/vast/python2env/bin/activate
python blast6-4_blastsnake.py $
#Call to the python script

cd /scratch/mvoogt/results

cs="$(date '+%s')"
cat blast6-4_results.out > blast6-4_results.out
ce="$(date '+%s')"
echo "$(date) CAT IS DONE: $((ce - cs)) seconds taken."
#Go to the results directory and append all results
rms="$(date '+%s')"
rm /scratch/mvoogt/Test_ITS2.fasta_
rm blast6-4_results_
rme="$(date '+%s')"
echo "$(date) RM IS DONE: $((rme - rms)) seconds taken."
#Remove the result parts
endtime="$(date '+%s')"
echo "$(date) TOTAL JOB IS DONE: $((endtime - starttime)) seconds taken."
#Calculate total time
exit 0
```