

CAB 401 – High Performance and Parallel Computing

Report

By Jordi Smit - 10264139

Table of contents

1 Application overview	3
1.1 What does this application do?	3
1.2 Application components	3
1.2.1 Sequential	3
1.2.2 BLOSUM62	4
1.2.3 Sigma70Consensus	4
1.2.4 Sigma70Definition	4
1.2.5 GenbankRecord	4
1.2.6 SmithWatermanGotoh	4
1.2.7 NucleotideSequence	4
1.2.8 BioPatterns	4
2 Analysis	5
2.1 Dependencies	5
2.1.1 Parse Reference Genes method	5
2.1.2 Parse method	5
2.1.3 Homologous method	5
2.1.4 Get Upstream Region method	6
2.1.5 Predict Promoter method	6
2.1.6 The main loops	6
2.2 Heavy work profiling	7
2.3 Potential Parallelism	7
2.3.1 Parallelize loop 1	8
2.3.2 Parallelize loop 2	8
2.3.3 Parallelize loop 3	8
2.3.4 Rewrite code	9
3 Approach	9
3.1 Handling performance barriers	9
3.1.1 Load imbalance	9
3.1.2 Stateful functions	10
3.1.3 Combining the final results	10
3.2 Parallel code solutions	10
3.2.1 Streaming solution	11
3.2.2 Executor Service solution	11
4 Result	12
4.1 Profiling of work	12
4.1.1 Streaming solution	12
4.1.2 ExecutorService solution	13
4.2 Timing	14
4.3 Speedup	15

4.4 Verifying the results	16
5 Conclusion	16
6 Reflection	16
Bibliografie	18
Appendix I Speed up graphs	19
Appendix II High level UML including methods	21
Appendix III Running application	22
Appendix IV Test results	24
Appendix V Stateful function in library method	25
Appendix VI The main loops of the application.	26
Appendix VII Description of used tools	27
Hardware	27
Compilers	27
Software	27
Parallel streams	27
Executor service	28
ThreadLocal variable	28
Synchronized methods and locks	28
JUnit 5 testing framework	28
Maven	28
Appendix VIII Rewritten sequential version	29
Appendix IX Parallel source code ExecutorService	30
Appendix X Parallel source code collecting ExecutorService version	31

1 Application overview

1.1 What does this application do?

This application takes in a list of reference genes and multiple gene bank files as input. It then starts off by parsing these files and storing them in memory as objects. The application then continues by going through every combination of reference gene and gene from the gene bank in order to find matches between them.

However, checking each possible combination of genes for matches is rather costly, especially since a lot of these combinations won't have any matches. So in order to minimize the amount of work, the application uses the homology property to pre filter the gene combination. Homology is the existence of shared ancestry between a pair of structures, or genes, in different taxa ("Homology (biology) - Wikipedia," n.d.). The most common example of homology are the similarities between the forelimbs in animals, such as dogs, birds and whales. If the two genes share a common ancestor it's also very likely that these gene will have at least some matches.

This application tests gene combinations for homology using the Smith–Waterman algorithm. This algorithm compares gene segments of all possible lengths and optimizes the similarity measure using dynamic programming which makes it much faster than the brute force approach ("Smith–Waterman algorithm - Wikipedia," n.d.).

The remaining gene combinations will be matched based and the results of these matches will be stored in the output of the application.

1.2 Application components

As can be seen in the UML diagram of this application (Appendix II), it mainly uses 8 classes namely: Sequential (1.2.1), BLOSUM62 (1.2.2), Sigma70Consensus (1.2.3), Sigma70Definition (1.2.4), GenbankRecord (1.2.5), SmithWatermanGotoh (1.2.6), NucleotideSequence (1.2.7) and BioPatterns (1.28).

1.2.1 Sequential

The sequential class is the main class of the application. This can also be seen in the UML diagram, since all the dependencies originate from this class (Figure 1.2.a).

This class starts off by loading the BLOSUM62 data, sigma70_pattern data, the reference genes data and the genbank data from file. It then continues by looping through every possible combination of reference gene and gene in the gene banks and checks each combination for homology. This class uses the methods from the SmithWatermanGotoh class and the data from the BLOSUM62 class in order to determine if a combination of genes are homologous. If the gene combination is found to be homologous this class will match the upstream region of these genes with the sigma70_pattern data and save these matches in the consensus. When this class has gone through each gene combination it will output all the matches it has stored in the consensus.

1.2.2 BLOSUM62

The BLOSUM62 class is a data class. This class is responsible for storing all the raw values for each type of acid. This class is only used to retrieve this data.

1.2.3 Sigma70Consensus

The Sigma70Consensus class is responsible for storing the matches that have been found. The sequential class has a Sigma70Consensus instance for each reference gene in which it stores the matches that have been found for this specific gene. The sequential class also has another Sigma70Consensus instance in which it stores all matches that have been found.

1.2.4 Sigma70Definition

The Sigma70Definition is also a data class. This class contains the pattern the application is looking for in the upstream regions. This class only has one method which creates a new copy of this pattern.

1.2.5 GenbankRecord

The GenbankRecord is a data class that is being used to store the data from the genbank files. This class only has a single method which is used to parse the genbank files. Besides these method, this class also has pointers to the stored genes and the nucleotide sequence.

1.2.6 SmithWatermanGotoh

The Smith Waterman Gotoh class contains an implementation of the Smith Waterman algorithm. This algorithm is being used by the sequential class in order to determine if two genes are Homologous using the genes and BLOSUM_62 data as input.

1.2.7 NucleotideSequence

The NucleotideSequence is a data class that contains the nucleotide information of the genbank. This class is only used to pass data around.

1.2.8 BioPatterns

The BioPattern class only has 1 method. This method is used by the sequential class to find a match between the gene and the sigma70_pattern.

2 Analysis

I started my analysis by examining the main dependencies (2.1) and profiling the workload (2.2). Then I continued my analysis by searching for areas with potential for parallelism (2.3).

2.1 Dependencies

In the program there are 6 areas of interest namely: the ParseReferenceGenes method (2.1.1), the Parse method (2.1.2), the Homologous method (2.1.3), the GetUpstreamRegion method (2.1.4), the Predict Promoter method (2.1.5) and the main loops (2.1.6)

2.1.1 Parse Reference Genes method

The ParseReferenceGenes method is an IO function that reads the list text file from disk. This file contains the genes we want to compare with the gene banks. This function must be evaluated before the program can enter the main loops, since the result of this function is used in these loops. This method changes the state of the instance variable called consensus. Which means that this method introduces a data dependency.

2.1.2 Parse method

The Parse method is an IO function that searches the provided gene bank directory for gene bank files and parses them. This method is a pure function, as it does not modify the state of any instance variables. This method returns a list of gene banks that are used in the main loop. So this function must be evaluated before the program can enter the main loop.

2.1.3 Homologous method

The Homologous method is used to determine if the gene of interest and the gene from the gene bank are homologous. This method uses the Smith Waterman Gotoh algorithm to determine if the two genes are homologous. The Homologous method provides the Smith Waterman Gotoh algorithm with two deep copies of the gene as input (Figure 2.1.c), which means that there is only a input dependency with no possibility of changing the state using on references.

The algorithm also takes in a reference to the BLOSUM_64 instance variable as input. The BLOSUM_64 variable is a matrix value object which contains the raw score values. Based on deeper inspection in the Smith Waterman Gotoh algorithm we see that there are only input dependencies for this value object.

Finally the algorithm also takes in two floating point variables as input. These variables are local and are only being used once, so these variables do not have any dependencies with the main program.

Which means that the Homologous method is a pure function. This will proof to be quite important since most of the work is being performed within this function (2.1).

```

private boolean Homologous(PeptideSequence A, PeptideSequence B) {
    return SmithWatermanGotoh
        .align(
            new Sequence(A.toString()),
            new Sequence(B.toString()),
            BLOSUM_62,
            openGapPenalty: 10f,
            extendGapPenalty: 0.5f
        ).calculateScore() ≥ 60;
}

```

Figure 2.1.3.a The Homologous method

2.1.4 Get Upstream Region method

This method has input dependencies with the complement instance variable reference, nucleotide sequence input reference and the gene input reference. This method does not have any writing dependencies. So this method can be evaluated in parallel as long as the state of these references will not be changed by any other thread.

2.1.5 Predict Promoter method

The Predict Promoter method uses the sigma70_pattern instance variable as reference input for the get Best Match method (Figure 3.1.2.a). Which is an input dependency.

```

private Match PredictPromoter(NucleotideSequence upStreamRegion) {
    return BioPatterns.getBestMatch(sigma70_pattern, upStreamRegion.toString());
}

```

Figure 2.1.5.a Sigma70_pattern is class variable.

The match method uses the sigma70_pattern reference inside its function. Based on deeper examination of the match method we see that this method holds temporary instance state. The index variable that is being used in the while loop is an instance variable, which means that there is an output dependency inside this method (Appendix V).

So this means that even though we only have an input dependency of the sigma70_pattern reference we cannot run this method on multiple threads due to the temporary state during its evaluation.

This dependency can easily be removed by changing the instance variable into a local variable. However since this method is part of an external library, we will be unable to change this code.

2.1.6 The main loops

For each gene of interest the program loops through all the genes in the gene banks (Appendix VI). We have discussed the methods used inside these loops in the previous paragraphs and we concluded that none of these methods change the state of the gene objects. Which means that there are only input dependencies for these loop variables, so we can go through these loops in any order and also potentially in parallel.

However, the main problem happens in the Predict Promoter method as this function cannot safely be evaluated in parallel due to its internal state (1.2.5).

There are also control dependencies inside the innermost loops due to the two if statements of which we must be aware.

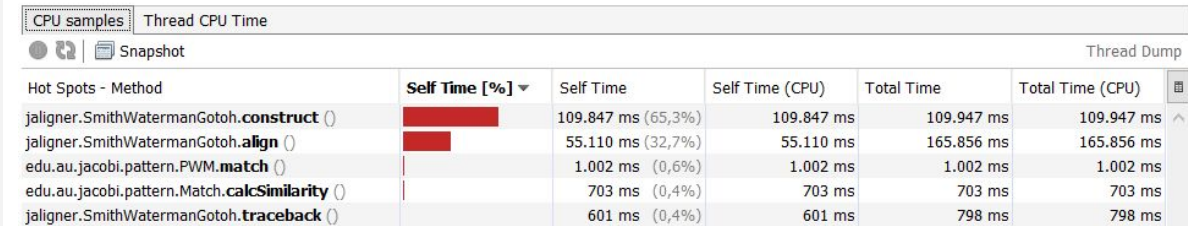
The innermost loop also adds the results incrementally to the consensus instance variable. The add match method updates the predictions and gapTotal instance variables of the consensus object, which introduces two flow dependencies (Figure 1.2.6.a). Which means that we cannot update the consensus in parallel.

```
public void addMatch(Match match) {
    predictions += 1;
    gapTotal += match.getSubMatch(1).calcLength();
    add_10Sequence(match.getSubMatch(2).letters().getBytes());
    add_35Sequence(match.getSubMatch(0).letters().getBytes());
}
```

Figure 2.1.6.a Flow dependence in output combination.

2.2 Heavy work profiling

When we run the application with a profiler, in this case visualVM (VisualVM: Home), we see that the application spends roughly 65% in the construct method and 32% in the align method (Figure 2.2.a). Both of these methods are part of the Smith Waterman Gotoh algorithm. This algorithm is only being used in the Homologous method (1.2.3), which dependencies would potentially allow it to be evaluated in parallel. So based on the dependencies and the amount of work we have found the ideal part to parallelize.



Hot Spots - Method	Self Time [%]	Self Time	Self Time (CPU)	Total Time	Total Time (CPU)
jaligner.SmithWatermanGotoh.construct ()	65,3%	109.847 ms	109.847 ms	109.947 ms	109.947 ms
jaligner.SmithWatermanGotoh.align ()	32,7%	55.110 ms	55.110 ms	165.856 ms	165.856 ms
edu.au.jacobi.pattern.PWM.match ()	0,6%	1.002 ms	1.002 ms	1.002 ms	1.002 ms
edu.au.jacobi.pattern.Match.calcSimilarity ()	0,4%	703 ms	703 ms	703 ms	703 ms
jaligner.SmithWatermanGotoh.traceback ()	0,4%	601 ms	601 ms	798 ms	798 ms

Figure 2.2.a The profiler's sampling results

2.3 Potential Parallelism

We want to find a way to paralyze either the loop 1 (2.3.1) in Appendix VI , loop 2 (2.3.2) or loop 3 (2.3.3), since we concluded that the main amount of work happens inside these loops (2.2). As previously discussed in chapter 1 we can iterate through the genes in parallel and evaluate the Homologous and getUpstreamRegion method in parallel. However, the main problems are in the PredictPromoter and consensus update, as these methods cannot be evaluated in parallel in their current form. The PredictPromoter problem can most likely be solved by either an Threadlocal variable or through locks/synchronization. However the consensus update problem can only be solved through locks/synchronization. (In this paragraph we will focus on the amount of parallel tasks we can achieve by paralyzing specific parts of the code, for more information about how to handle these problems see chapter 3).

Besides simply paralyzing these loops it might be more useful to rewrite their code to expose more parallelism (2.2.4).

2.3.1 Parallelize loop 1

The amount of tasks we can evaluate in parallel in this solution depends on the number of gene bank files. Currently there are only 4 gene bank files, so at most we can evaluate 4 tasks in parallel. The different amount of genes varies per gene bank file, as does the amount of work per gene, so we will most likely end up with an unbalanced workload distribution. Which will introduce a performance barrier through idle threads.

Besides these problems we also have to deal with the synchronization of the PredictPromoter method and consensus update, which will introduce a waiting time performance barrier.

This solution might have been more useful if we had more gene banks to evaluate, since this would allow us to run more than 4 parallel tasks. However this would not solve the unbalanced workload distribution problem.

2.3.2 Parallelize loop 2

The amount of parallel tasks we can create for this option depends on the number of reference genes. Currently there are 8 reference genes, so we can only create at most 8 parallel tasks. The simplest option to run these loops in parallel would be to rewrite this loop directly into a parallel form. However this would result in a fork and join for each gene bank, which would introduce a performance barrier.

A better option would be to swap the evaluation order of loop 1 and loop 2. As we discussed in chapter 1 we are allowed to do this based on the dependencies of the loop variables. All though it might be wise to perform the IO of the gene banks only once before we start the iterating part by storing it as a variable. Repeating the IO part multiple times would introduce an unnecessary performance penalty.

This solution would solve the workload unbalance problem caused by the varying amount of genes per gene bank. However the workload also depends on the length of the reference gene, so even though this solution would be more balanced than the previous solution it would still have a workload unbalance.

Just like the previous solution, this solution also has to deal with the synchronization of the PredictPromoter method and consensus update, which will also introduce a waiting time performance barrier.

2.3.3 Parallelize loop 3

We could also transform the innermost loop into a parallel form. With this option we finally have a large amount of tasks we can run in parallel. With the data that is currently being provided, we have on average 4143 tasks we can do in parallel.

The main advantage of this solution is that the parallelism is scalable, since on average we can run at most 4143 tasks in parallel. Another advantage of this approach is that we have far more options to find an evenly distributed workload. However we still have to be careful of how we distribute the work. The amount of work depends on the length of each gene and they still vary in length.

However the disadvantage of this approach is that we have to perform a fork and join for each combination of reference gene and gene bank. This will result in a lot of overhead.

Just like the previous solutions, this solution also has to deal with the synchronization of the PredictPromoter method and consensus update, which will also introduce a waiting time performance barrier.

2.3.4 Rewrite code

The problem with the previous approaches was that we had either very little parallel tasks or we had a lot of fork join operations. Either way the main amount of works still happens in the Homologous method. This method requires a record, referenceGene and gene as input. By first collecting all these combinations and only then performing the work we would be able to expose a large amount of parallel tasks while only requiring a single fork join.

With the provided example data we have $16575 = 4084 + 4128 + 4146 + 4217$ genes in the gene banks and 8 references genes. Thus in total we would have to collect $16575 * 8 = 132600$ combinations.

The main advantage of this approach is that its parallelism scales independent of the amount of input data. However we must keep in mind that by using this approach we will reach the out of memory exception much faster when the amount of input data would increase. But based on the amount of example data we will most likely not reach this point in the near future. The other advantage is that we have far more options to divide the workload evenly just like the loop 3 solution (2.3.3), while only requiring a single join fork.

Just like the previous solutions, this solution also has to deal with the synchronization of the PredictPromoter method and consensus update, which will also introduce a waiting time performance barrier.

3 Approach

3.1 Handling performance barriers

We already discussed the potential parallel solutions in chapter 2. However each of these solutions will encounter the same performance barriers namely: load imbalance (3.1.1), stateful functions (3.1.2) and combining the final results (3.1.3).

3.1.1 Load imbalance

The largest amount of work happens in the Homologous method. The load imbalance problem with this function is that its workload varies depending on the length of the input gene. Which makes it very hard to distribute the work evenly. We can take two possible approaches with this performance barrier. Firstly we can assume (or hope) that the gene length will be almost equal. With this assumption it is best to divide the data into chunks using the streaming framework and evaluate each chunk in parallel as this will minimize the synchronization overhead.

However if the gene lengths vary greatly we will end up with a large workload imbalance. In this case it will be better to use the executorservice, since this will divide the work very well over the worker thread pool. However using the executorservice will introduce a large amount of synchronization overhead.

The main problem with the workload imbalance is that it will be a trade of between idle threads (3.1.1) and synchronization overhead (3.1.2). Both solutions are equally good however which approach will prove to be the best for a specific data set truly depends on the data set.

In order to find the best possible solution, I implemented both approaches and discuss them in paragraaf 3.2.

3.1.2 Stateful functions

As previously discussed in chapter 2, the PredictPromoter method calls a temporary stateful method on the instance variable `sigma70_pattern`. Since we do not write to this variable (except for the temporary state) we can replace this variable using a thread local variable. This would solve the temporary state problem since each thread would have its own copy of the variable. Within the thread the `sigma70_pattern` will only be called in a sequential manner so the temporary state will no longer be an issue.

3.1.3 Combining the final results

As previously discussed in chapter 2, the consensus update has a flow dependency. We can handle this dependency in 2 possible ways. Firstly we wrap the consensus update in a synchronized block. This will ensure that only a single thread can execute this code, which will keep the flow dependency intact. However using locks will create additional overhead, since the threads must wait until the resources become available.

An alternative approach would be to collect all the updating tasks and perform them in a sequential manner. This would minimize the thread synchronization waiting time overhead as the threads can simply continue working. However there will still be a synchronization overhead cost at the end, since we still have to collect the results from all the threads. Which trade off will prove to be the best will depend on the input data. So I implemented both version in order to measure which version performs the best (3.2).

3.2 Parallel code solutions

I decided to go for solution 2.3.2 since this is the only solution that allows for scalable parallelism. So before we can add parallelism to the application, we have to rewrite the code in the sequential version in order to expose this parallelism. I made 3 changes to the sequential version, in order to achieve the goals of 2.3.3. Firstly, I extracted the IO function of the gene banks from the original loops (1 in Appendix VIII) and stored the results into a variable. Secondly, I collected all the gene comparison tasks in to a list (2 in Appendix VIII). Finally, we iterate through all the gene comparison tasks and evaluate them (3 in figure Appendix VIII). With this new version we haven't introduced any parallelism yet, so the evaluation of the tasks will still happen in sequence. However this version will make it much easier to achieve scalable parallelism. In 3.2.1 we will discuss how we can introduce scalable parallelism to this version using the streaming framework, while in 3.2.2 we will discuss how we can also achieve this using the ExecutorService framework.

The source code of all the solutions can be found in the `qut.parallel_versions` package.

3.2.1 Streaming solution

In this solution we assume that the workload can be evenly distributed over multiple chunks. The Homologous work is responsible for the largest amount of work, so let's start by transforming this work into a parallel form. This is also an easy place to start since the Homologous method doesn't have any dependencies we have to worry about. So we don't have to introduce any complicated overhead such as locks or thread local variables. Using the streaming framework we can run the Homologous work in parallel by replacing the loop 3 in Appendix VIII with the code in figure 3.2.1.a. This code starts by creating a parallel stream using the `parallelStream` method. The code then continues by filtering the streams using the Homologous method in parallel. Finally the code collects the filtered results into a list (using a fork join operation that has been abstracted away by the framework) and performs the remaining code in sequence. This simple change already results into a speed up of 3.67 on 8 cores, which is already a huge improvement.

```
dataContainers.parallelStream()
    .filter(dataContainer → Homologous(dataContainer.getGene().sequence, dataContainer.getReferenceGene().sequence))
    .collect(Collectors.toList())
    .forEach(dataContainer → {
        NucleotideSequence upStreamRegion = GetUpstreamRegion(dataContainer.getNucleotides(), dataContainer.getGene());
        Match prediction = PredictPromoter(upStreamRegion);

        if (prediction ≠ null) {
            consensus.get(dataContainer.getName()).addMatch(prediction);
            consensus.get("all").addMatch(prediction);
        }
    });
```

Figure 3.2.1.a Perform the filtering in parallel.

By transforming the `sigma70_pattern` instance variable into a thread local variable (3.2.2), we will be able to also perform the `GetUpstreamRegion` method, `PredictPromoter` method and is prediction not null check in parallel. As discussed before the consensus update must be evaluated in sequence (3.1.3). So this will result in to the code in figure 3.2.1.b. This code improves the speed up to 3,838 on 8 cores which is better than the original streaming version.

```
dataContainers.parallelStream()
    .filter(dataContainer → Homologous(dataContainer.getGene().sequence, dataContainer.getReferenceGene().sequence))
    .map(dataContainer → {
        NucleotideSequence upStreamRegion = GetUpstreamRegion(dataContainer.getNucleotides(), dataContainer.getGene());
        return new PredictionContainer(PredictPromoter(upStreamRegion), dataContainer.getName());
    })
    .filter(predictionContainer → predictionContainer.prediction ≠ null)
    .collect(Collectors.toList())
    .forEach(predictionContainer → {
        consensus.get(predictionContainer.name).addMatch(predictionContainer.prediction);
        consensus.get("all").addMatch(predictionContainer.prediction);
    });
```

Figure 3.2.1.b Performing additional work in parallel.

3.2.2 Executor Service solution

In this solution we assume that the workload cannot be evenly divided into chunks, so we try to achieve a balanced workload using scheduling. In this solution we replace the loop at 3 in Appendix VIII with the code in figure 3.2.2.a. This code starts by creating a worker thread pool and then uses an `ExecutorService` to schedule the tasks on the workers.

```

for (GenbankRecord record : records) {
    for (Gene referenceGene : referenceGenes) {
        System.out.println(referenceGene.name);
        for (Gene gene : record.genes) {
            dataContainers.add(new DataContainer(gene, referenceGene, record.nucleotides, referenceGene.name));
        }
    }
}

ExecutorService executorService = Executors.newFixedThreadPool( nThreads: 8);
executorService.invokeAll(dataContainers);
executorService.shutdown();

```

Figure 3.2.2.a Starting up the ExecutorService

The ExecutorService requires its input to implement the Callable interface. This interface specifies the code that will be run in parallel (Appendix IX). The consensus cannot be updated by multiple threads at the same time. So in order to prevent this from happening we require each thread to acquire a lock before they are allowed to update the consensus.

This code is able to achieve a speed up of 4,184 on 8 cores.

The locks in the previous solution might require some threads to wait in idle mode before they can continue. We might be able to achieve a higher speedup if we extract the consensus update and perform them all at the end in sequence. We can do this by replacing the loop at 3 in Appendix VIII with the code in figure 3.2.2.b. All the tasks will be performed in parallel except the consensus update (Appendix X). And only after all these tasks have been performed we will collect them to perform the consensus update. This solution improves the speed up a little bit to 4,232.

```

ExecutorService executorService = Executors.newFixedThreadPool( nThreads: 8);
List<Future<PredictionContainer>> results = executorService.invokeAll(dataContainers);

for (Future<PredictionContainer> future : results) {
    PredictionContainer predictionContainer = future.get();
    if (predictionContainer != null) {
        consensus.get(predictionContainer.getName()).addMatch(predictionContainer.getPrediction());
        consensus.get("all").addMatch(predictionContainer.getPrediction());
    }
}

```

Figure 3.2.2.b Collecting ExecutorService solution

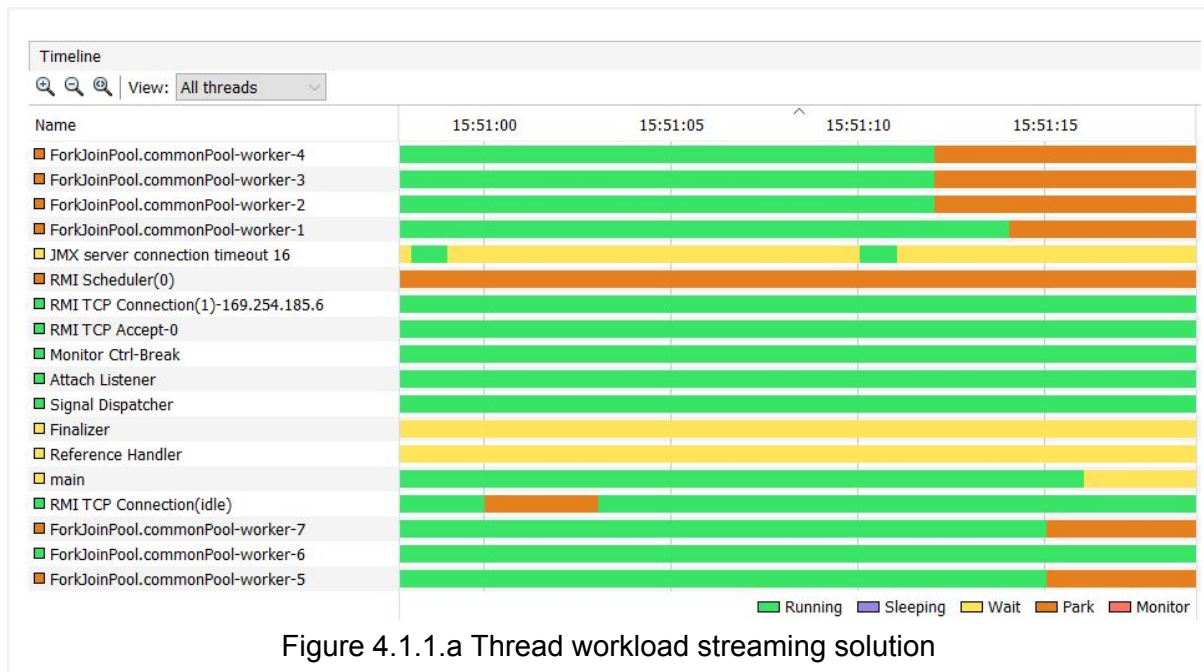
4 Result

In total I implemented 3 different parallel versions. I profiled the work (4.1) foreach version and also measured the execution time (4.2) and speedup (4.3) foreach version on different amount of cores.

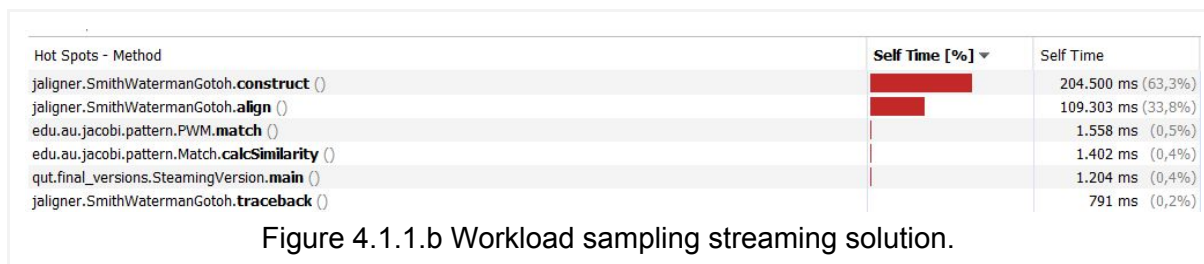
4.1 Profiling of work

4.1.1 Streaming solution

As we expected the chunk workload distribution approach results into an unbalanced workload distribution. Multiple workers are being parked while they wait until the last thread is finished (The orange bars in figure 4.1.1.a). This waiting time is the reason why this version is slower than the ExecutorService solution which has more overhead.



Even in this parallel version, the main amount of work still happens in the Smith Waterman Gotoh algorithm (Figure 4.1.1.b).

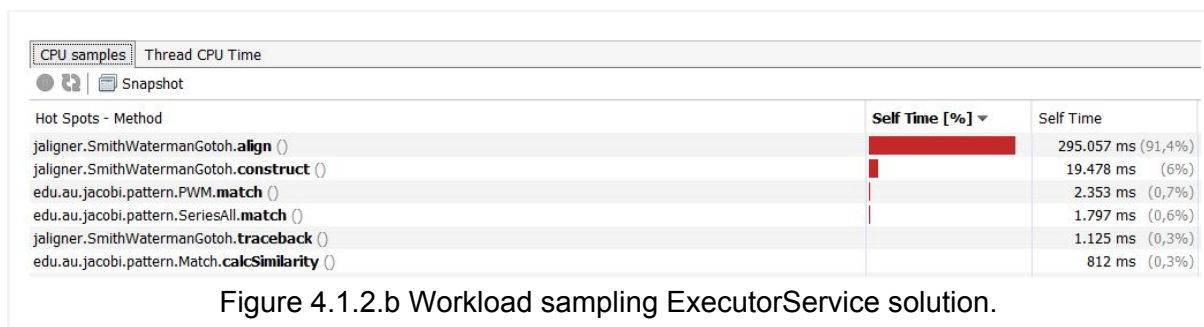


4.1.2 ExecutorService solution

As we expected ExecutorService approach into an very well balanced workload distribution. All the workers are being used until the end of the program (The green bars in figure 4.1.2.a).



The workload sampling is very similar to the streaming solution. The main amount of work still happens in the Smith Waterman Gotoh algorithm (Figure 4.1.2.b).



4.2 Timing

I run and timed the sequential version 10 times and on average the sequential version takes about 172,69 seconds to execute. I then runned and timed each of the different parallel versions 5 times on different amount of cores. This resulted into the measurements in the table below.

Number of threads	Steaming version in seconds	Executorservice synconzided version in seconds	Executorservice collector version in seconds
1	182,53	179,88	173,77
2	94,95	90,92	88,53
3	68,50	66,60	64,49
4	58,58	57,36	56,09
5	53,89	52,40	51,00
6	50,37	47,31	46,14
7	47,94	43,51	42,81
8	44,99	41,28	40,81

4.3 Speedup

Based on the measurements from paragraph 4.2 we can calculate the speedup for each version. This resulted into the values in the table below. The speed up graphs based on this table can be found in appendix I.

Number of threads	Steaming version speed up	Executorservice synchronized version speed up	Executorservice collector version speed up
1	0,946	0,960	0,994
2	1,819	1,899	1,951
3	2,521	2,593	2,678
4	2,948	3,010	3,079
5	3,204	3,296	3,386
6	3,429	3,650	3,743
7	3,602	3,969	4,034
8	3,838	4,184	4,232

4.4 Verifying the results

We can not simply use the new parallel program without first verifying if its output is exactly the same as the original program. In order to ensure this I created an automated test that compares the output of the parallel version to the original output of the sequential version (Appendix IV). The output results of the original version are read from a file to ensure that we cannot accidentally change these results.

This test case has been used on each parallel version and each version has passed it (Appendix IV).

5 Conclusion

The main difficulty in transforming this application into a parallel application is the balancing of the workload distribution. We can divided all tasks into parallel batches as we did the streaming approach, however we will run into preformance penties due to idle threads if workload of the input data is unbalanced.

The alternative approach is to minimize the workload imbalance by scheduling the tasks based on synchronized queue as we did with the executorservice approach. However with this approach will run into preformance penties due to the synchronizion overhead.

So eventually the choice between the approaches becomes a trade of between preformance penties due to idle threads or synchronizion overhead. Which approaches is the best truly depends on the input data. The stream approach will most likely outperform the executorservice approach on decently balanced input data, while executorservice approach will outperform the streaming version on very unbalanced data.

The input data that has been provided is very unbalanced, so that is why the executorservice currently out performs the streaming version. This imbalance in the workload result into a lot of additional overhead and is the reason why it not possible to reach near linear speed up.

6 Reflection

How successful was my attempt?

Sadly, I was unable to achieve near linear speed up due to the high synchronization overhead costs caused by the load imbalance problems. After a lot load balancing optimizations I was able to reach a speedup of 4,232 on 8 virtual cores.

What could I you have done differently?

Sadly, I was unable to achieve near linear speed up due to the high synchronization overhead costs caused by the load imbalance problems. I might have been able to achieve near linear speed up if I would have used GPU programming. However, I wasted a lot of time searching for a non default project but in the end I was unable to find an interesting one. If I had used this time more wisely I might have been able to learn myself how to do GPU programming in java. Currently the end of the semester is approaching very fast and a lot of deadlines are coming up. So this didn't leave me with enough time to learn GPU programming and that is why I stayed with the safe approach of CPU programming.

What have I learnt?

When I started with this project, I thought parallel programming was very easy. You just create a parallel stream and you add some synchronization and you are done. However with this simplistic approach you will not be able to reach a near linear speedup. This project clearly showed me how important it is to think about the dataflow, workflow and workload distribution of your application, since these determine the maximum amount of speedup you can achieve.

Before I did this project, I also thought that parallel streams dequeued their work from a large queue. However I now know that streams distribute their work over workers using chunks. By distributing the work into chunks they minimize the data movement, which significantly reduces the overhead.

The final thing I learned during this project might sound silly but before this project I did not know what a profiler was. During this project I worked a lot with Java's VisualVM profiler and I now know how useful a profiler can be. In future projects I will definitely use it again.

Bibliografie

Homology (biology) - Wikipedia. (n.d.). Retrieved October 21, 2018, from

[https://en.wikipedia.org/wiki/Homology_\(biology\)](https://en.wikipedia.org/wiki/Homology_(biology))

javac. (2016, January 8). Retrieved October 16, 2018, from

<https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javac.html>

Java compiler reordering. (n.d.). Retrieved October 10, 2018, from

<https://stackoverflow.com/questions/31015169/java-compiler-reordering>

Java SE Development Kit 8 - Downloads. (n.d.). Retrieved October 16, 2018, from

<https://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>

JUnit 5. (n.d.). Retrieved October 16, 2018, from <https://junit.org/junit5/>

Kishore, P. (n.d.). Compiler Reordering: final and volatile. Retrieved October 10, 2018, from

<http://brainatjava.blogspot.com/2015/10/compiler-reordering.html>

Maven – Introduction. (n.d.). Retrieved October 16, 2018, from

<https://maven.apache.org/what-is-maven.html>

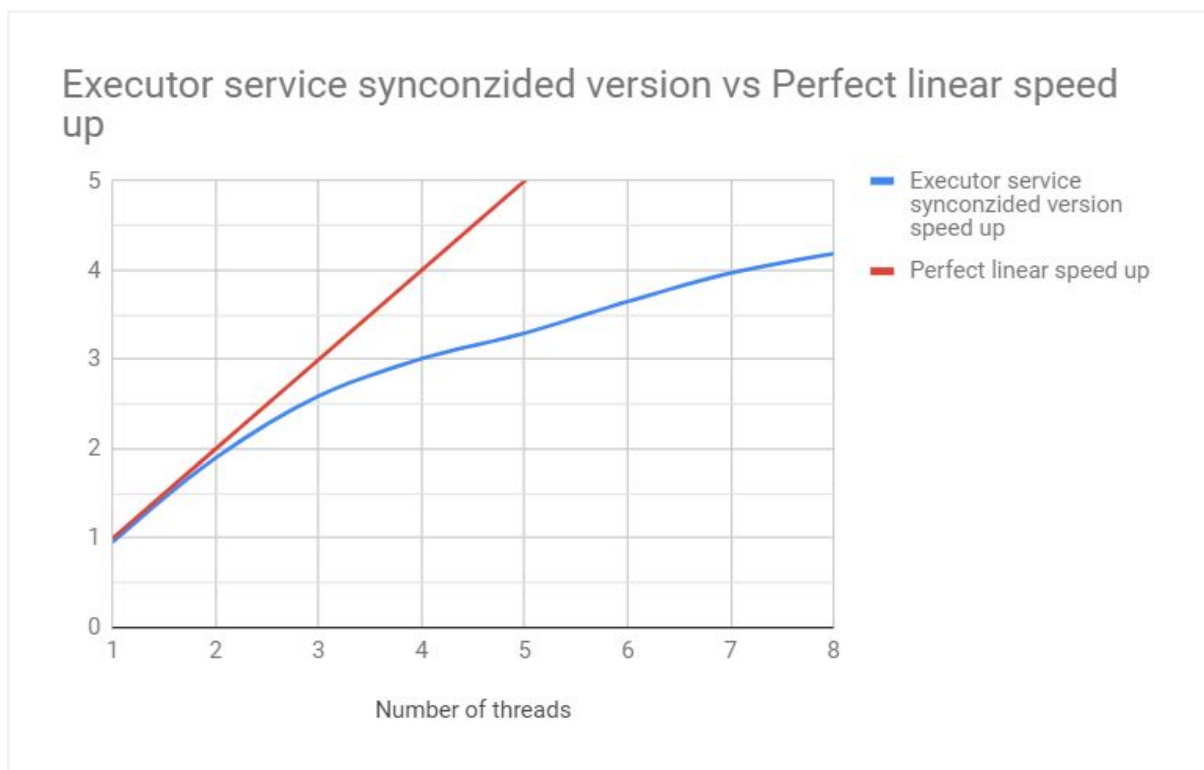
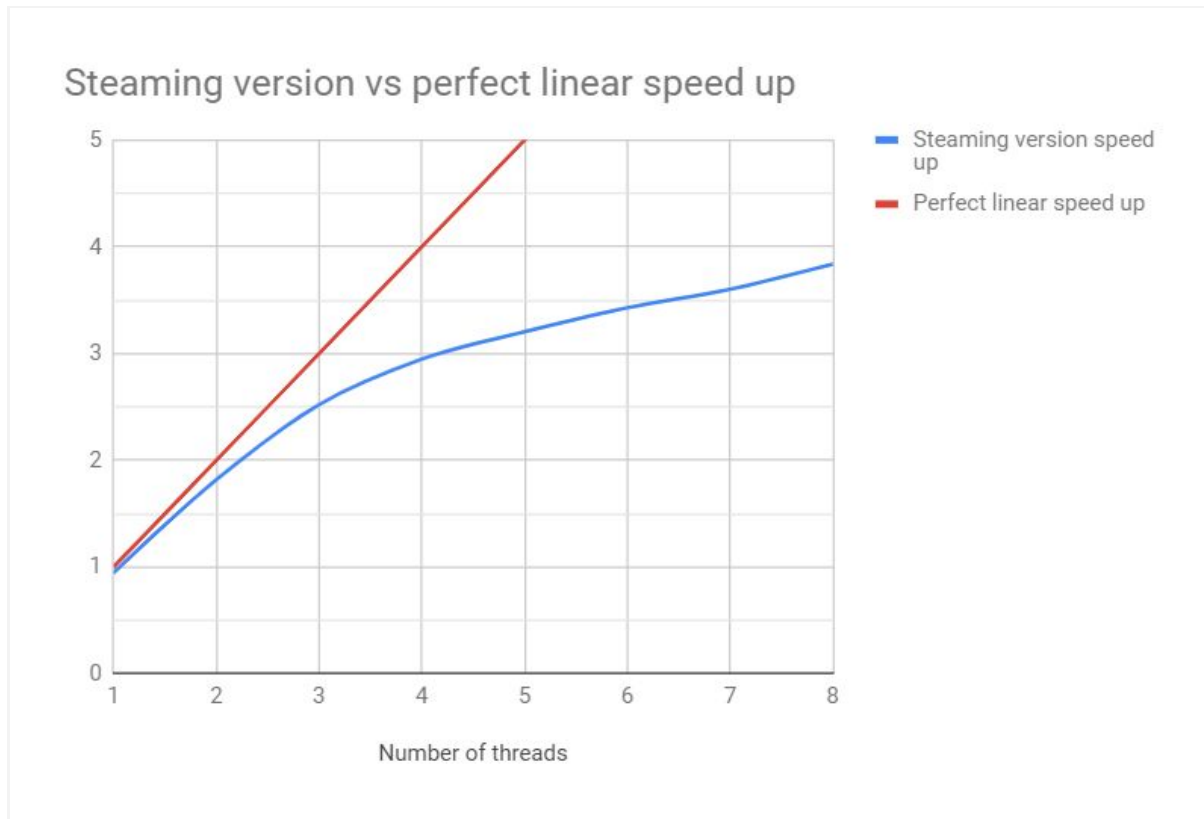
Smith–Waterman algorithm - Wikipedia. (n.d.). Retrieved October 21, 2018, from

https://en.wikipedia.org/wiki/Smith%E2%80%93Waterman_algorithm

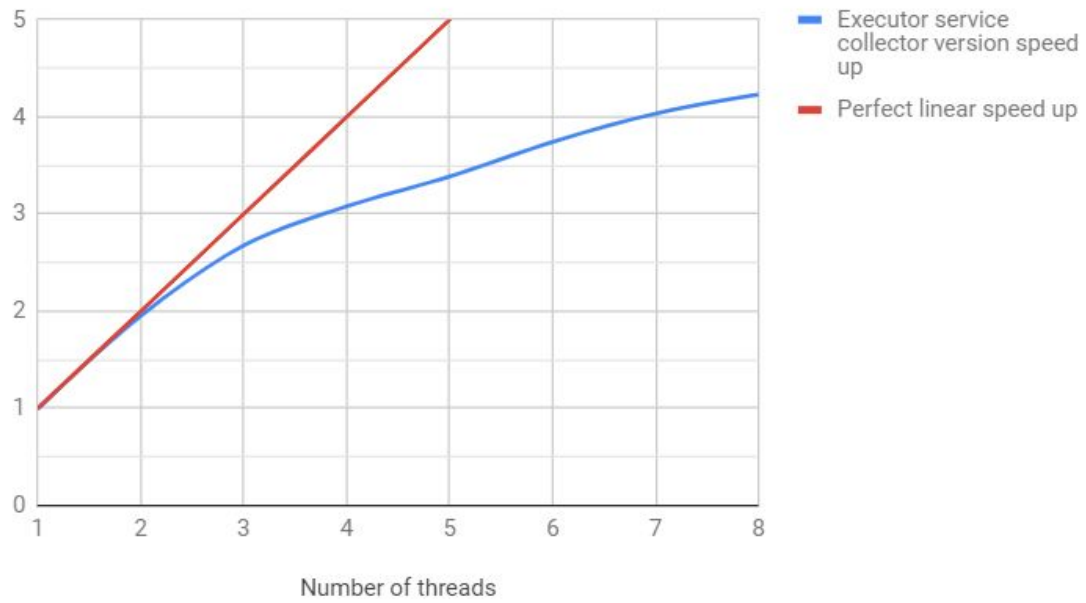
Travis CI - Test and Deploy Your Code with Confidence. (n.d.). Retrieved October 16, 2018,

from <https://travis-ci.org/>

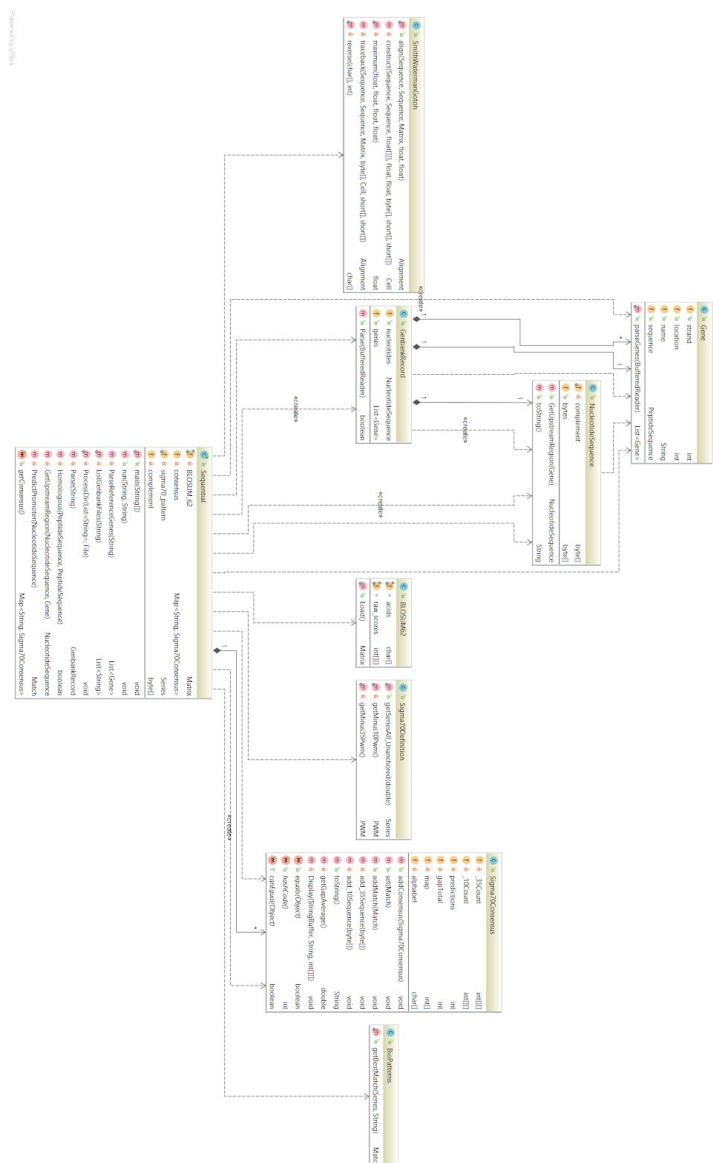
Appendix I Speed up graphs



Executor service collector version vs en Perfect linear speed up



22



Appendix III Running application

This application is build using maven. So in to run this application it required to have installed on your computer. (If you decided to run it in an IDE you might need the lombok plugin.)

Step 1

Navigate to projects directory in your terminal (the folder with the pom.xml file).

```
J@DESKTOP-5F0BQM3 MINGW64 ~/Documents/GitHub/CAB401/project (master)
$ ls
consensus.json      jsonHelperTest.json  referenceGenes.json   src/                uml.png
ecoli/              lib/                  referenceGenes.list   target/             uml.uml
expected_output/    pom.xml               result_1536298355852.txt testLists/          uml_complete.png
```

Step 2

Compile the project by running the following command: “mvn compile”

```
J@DESKTOP-5F0BQM3 MINGW64 ~/Documents/GitHub/CAB401/project (master)
$ mvn compile
[INFO] Scanning for projects...
[INFO]
[INFO] -----< CAB401:project >-----
[INFO] Building project 1.0-SNAPSHOT
[INFO] -----[ jar ]-----
[INFO]
[INFO] --- maven-install-plugin:2.5.1:install-file (install-jar-lib) @ project ---
[INFO] pom.xml not found in jacobi.jar
[INFO] Installing C:\Users\J\Documents\GitHub\CAB401\project\lib\jacobi.jar to C:\Users\J\.m2\repository\edu\au\jacobi\jacobi\1.0\jacobi-1.0.jar
[INFO] Installing C:\Users\J\AppData\Local\Temp\mvninstall4450613684901646343.pom to C:\Users\J\.m2\repository\edu\au\jacobi\jacobi\1.0\jacobi-1.0.pom
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ project ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 1 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ project ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] BUILD SUCCESS
[INFO]
[INFO] Total time: 1.440 s
[INFO] Finished at: 2018-10-21T21:12:18+10:00
[INFO]
```

Step 3

Start the Sequential version using the command:

“mvn exec:java -Dexec.mainClass="qut.Sequential"”

Start the BasicSteamingVersion using the command:

“mvn exec:java -Dexec.mainClass="qut.parallel_versions.BasicSteamingVersion"”

Start the ExecutorServiceCollectorVersion using the command:

"mvn exec:java -Dexec.mainClass="qut.parallel_versions.ExecutorServiceCollectorVersion"

Start the ExecutorServiceSynconzidedVersion using the command:

"mvn exec:java

-Dexec.mainClass="qut.parallel_versions.ExecutorServiceSynconzidedVersion"

Start the SteamingVersion using the command:

"mvn exec:java -Dexec.mainClass="qut.parallel_versions.SteamingVersion"

```
J@DESKTOP-5FOBQM3 MINGW64 ~/Documents/GitHub/CAB401/project (master)
$ mvn exec:java -Dexec.mainClass="qut.parallel_versions.ExecutorServiceCollectorVersion"
[INFO] Scanning for projects...
[INFO]
[INFO] < CAB401:project >-----
[INFO] Building project 1.0-SNAPSHOT
[INFO] -----[ jar ]-----
[INFO]
[INFO] --- exec-maven-plugin:1.6.0:java (default-cli) @ project ---
nhaA
yaaY
carA
caiF
caiD
fixA
fixB
folA
nhaA
yaaY
carA
caiF
caiD
fixA
fixB
folA
nhaA
yaaY
carA
caiF
caiD
fixA
fixB
folA
nhaA
yaaY
carA
caiF
caiD
fixA
fixB
all Consensus: -35: T T G A C A gap: 17,6 -10: T A T A A T (5430 matches)
fixB Consensus: -35: T T G A C A gap: 17,7 -10: T A T A A T (965 matches)
carA Consensus: -35: T T G A C A gap: 17,7 -10: T A T A A T (1079 matches)
fixA Consensus: -35: T T G A C A gap: 17,6 -10: T A T A A T (896 matches)
caiF Consensus: -35: T T C A A A gap: 18,0 -10: T A T A A T (11 matches)
caiD Consensus: -35: T T G A C A gap: 17,6 -10: T A T A A T (550 matches)
yaaY Consensus: -35: T T G T C G gap: 18,0 -10: T A T A C T (4 matches)
nhaA Consensus: -35: T T G A C A gap: 17,6 -10: T A T A A T (1879 matches)
folA Consensus: -35: T T G A C A gap: 17,5 -10: T A T A A T (46 matches)
[INFO]
[INFO] BUILD SUCCESS
[INFO]
```


Appendix IV Test results

```
@Test
public void _slowSystemTest() {
    String dir = "Ecoli";
    String fileName = "referenceGenes.list";
    String filePath = TestResultsGenerator.formatOutputPath(fileName, dir);
    Map<String, Sigma70Consensus> expected = readExpectedFile(filePath);

    Map<String, Sigma70Consensus> result = calcConsensus(filePath, dir);

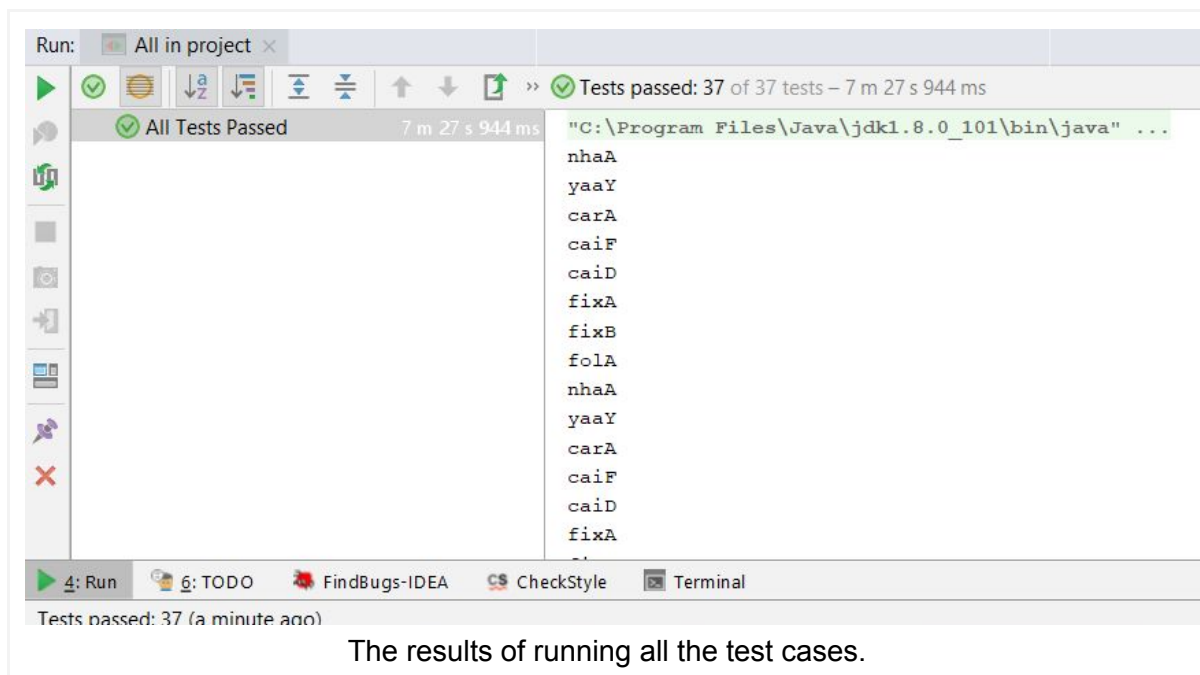
    assertThat(result).isEqualTo(expected);
}

@sneakyThrows
private Map<String, Sigma70Consensus> calcConsensus(String filePath, String dir) {
    ISequential ISequential = getISequential();

    ISequential.run(filePath, dir);

    return ISequential.getConsensus();
}
```

The test case that verifies that the program's output remain unchanged.



The results of running all the test cases.

Appendix V Stateful function in library method

```
public Match match(Sequence sequence, int position) {
    Match match = getMatch();
    Match nextMatch = null;
    int patternNumber = patternList.size();

    while(index ≥ 0) { // index of the current pattern
        IPattern pattern = get(index); // get the current pattern

        position = index > 0 ? match.getSubMatch(index-1).end()+1 : position;
        nextMatch = pattern.match(sequence, position);

        if(nextMatch == null) { // no other next match
            while(--index ≥ 0 && get(index).getIncrement() > 0)
                ;
            continue;
        }
        pattern.getMatch().set(nextMatch);

        index++;
        if(index == patternNumber) {
            while(--index ≥ 0 && get(index).getIncrement() > 0)
                ;
            increment = 0;

            match.calcSimilarity();
            if(match.similarity() < getThreshold())
                return(null);

            match.calcStartEnd();
            match.setStrand(sequence.strand());
            match.setSequence(sequence);

            return(match);
        }
    }
    increment = super.getIncrement();
    index = 0;
    return(null);
}
```

Figure 1.2.5.b Stateful function in library method. Index is instance variable.

Appendix VI The main loops of the application.

```
1 for (String filename : ListGenbankFiles(dir)) {  
    System.out.println(filename);  
    GenbankRecord record = Parse(filename);  
  
    2 for (Gene referenceGene : referenceGenes) {  
        System.out.println(referenceGene.name);  
  
        3 for (Gene gene : record.genes) {  
            if (Homologous(gene.sequence, referenceGene.sequence)) {  
                NucleotideSequence upStreamRegion = GetUpstreamRegion(record.nucleotides, gene);  
                Match prediction = PredictPromoter(upStreamRegion);  
  
                if (prediction != null) {  
                    consensus.get(referenceGene.name).addMatch(prediction);  
                    consensus.get("all").addMatch(prediction);  
                }  
            }  
        }  
    }  
}
```

The main loops of the application.

Appendix VII Description of used tools

In the different approaches we used 3 types of tools namely hardware related tools, compiler related tools and software related tools.

Hardware

For this project I used a HP Zbook 15G2 to test the speedup of my code. This computer has an Intel Core i7-4710MQ CPU with a clock speed of 2,5GHz. This CPU has 4 physical cores and 8 virtual cores.

Compilers

For this project I used the default java compiler called Javac ("javac," 2016). Java uses its compiler in two stages. Firstly it compiles the text files (.java files) into java bytecode (.class files). In this stage the compiler cannot change any in-thread execution orders. However in this stage the compiler can still omit some statements if it thinks they are redundant. In the second stage, which will be used as a just in time compiler, it will compile the java bytecode into binary machine code. This compiler can change the in-thread execution order. ("Java compiler reordering," n.d.).

We must be aware of these potential changes in a multithreaded environment, since it can affect the execution results. Java developers can specify which part of the code are the compiler is not allowed to change using synchronized blocks and the final keyword for variables (Kishore, n.d.).

The compiler does not introduce parallel execution by itself.

Software

For this project I used Oracle's Java version 1.8 ("Java SE Development Kit 8 - Downloads," n.d.). Java 1.8 provided me with parallel computing features such as Parallel streams, Executor services, ThreadLocal variables and synchronized methods/locks. Besides these default features I also used the third party library JUnit 5 to validate if the parallel versions kept returning the same result as the original version ("JUnit 5," n.d.). I also used Maven as automated building tool ("Maven – Introduction," n.d.).

Parallel streams

The parallel streaming framework is a new feature of java 1.8. This framework splits the work into chunks and divides them over multiple threads. The main advantage of using the streaming framework is that it reuses its worker threads. This means that we have to pay the thread creation penalty only once.

A disadvantage of using this feature is that it requires a balanced workload distribution because of its chunk distribution approach. If the workload is not balanced all the other threads will be waiting in idle mode until the last thread is finished.

Executor service

The executor service is a parallel task scheduling framework. It creates a pool of worker threads and puts all the work tasks into a synchronized queue. Each time a worker finishes its task it will dequeue the next task from the synchronized queue. This will continue until all the work has been done.

The main advantage of this approach is that it will result into a very well balanced workload distribution, since a worker thread will only go idle if there are no more tasks.

The main disadvantage is the huge overhead caused by the synchronized queue. This synchronization is necessary to ensure that the work will only be done once, however it also creates a performance barrier due to the large amount of needed communication.

ThreadLocal variable

By using a thread local variable we can ensure that each thread will have its own copy of the variable. This will help to prevent race conditions since none of the variables can be accessed at the same time. This makes it very useful for variables that hold temporary state and need to be shared by multiple threads.

A disadvantage of thread local variables is that they can only be used as reading variables.

Synchronized methods and locks

Synchronization and locks help to prevent race conditions by only allowing a single thread to enter its block of code. The main advantage of using synchronization and locks is that it allows for both reading and writing. However the main disadvantage is that it introduces additional overhead since threads might have wait before they can enter the code block.

JUnit 5 testing framework

JUnit is an automated testing framework. Before I paralyze the application, I started by creating multiple test cases based on the output of the original sequential version. I serialized these outputs into JSON files to make sure that the test outputs remained the same. I then added an automated testing trigger to my Github repo using Travis ("Travis CI - Test and Deploy Your Code with Confidence," n.d.). This made sure that every code change during the parallelization process did not change the output of the program.

Maven

I used Maven as an automated building tool. Maven handles the dependency management and building processes. Making it much easier to share a Java project over different devices, since the installation of the project now becomes automated.

Appendix VIII Rewritten sequential version

```
1 for (String filename : ListGenbankFiles(dir)) {
    records.add(Parse(filename));
}

2 for (GenbankRecord record : records) {
    for (Gene referenceGene : referenceGenes) {
        System.out.println(referenceGene.name);
        for (Gene gene : record.genes) {
            dataContainers.add(new DataContainer(gene, referenceGene, record.nucleotides, referenceGene.name));
        }
    }
}

3 for (DataContainer dataContainer : dataContainers) {
    if (Homologous(dataContainer.getGene().sequence, dataContainer.getReferenceGene().sequence)) {
        NucleotideSequence upStreamRegion = GetUpstreamRegion(dataContainer.getNucleotides(), dataContainer.getGene());
        Match prediction = PredictPromoter(upStreamRegion);

        if (prediction != null) {
            consensus.get(dataContainer.getName()).addMatch(prediction);
            consensus.get("all").addMatch(prediction);
        }
    }
}
```

Rewritten sequential version to create more potential parallel tasks.

- 1) Extracts the IO part from the original loops;
- 2) Collect all the task in a list;
- 3) Loops through the list and performs the task (still in sequences);

Appendix IX Parallel source code ExecutorService

```
private static final Object LOCK = new Object();

@Getter
@RequiredArgsConstructor
private class DataContainer implements Callable<Void> {
    private final Gene gene;
    private final Gene referenceGene;
    private final NucleotideSequence nucleotides;
    private final String name;

    @Override
    public Void call() throws Exception {
        if (Homologous(gene.sequence, referenceGene.sequence)) {
            NucleotideSequence upStreamRegion = GetUpstreamRegion(nucleotides, gene);
            Match prediction = PredictPromoter(upStreamRegion);

            if (prediction != null) {
                synchronized (LOCK) {
                    consensus.get(name).addMatch(prediction);
                    consensus.get("all").addMatch(prediction);
                }
            }
        }
        return null;
    }
}
```

Parallel part of the code for ExecutorService.

Appendix X Parallel source code collecting ExecutorService version

```
@Getter
@RequiredArgsConstructor
private class DataContainer implements Callable<PredictionContainer> {
    private final Gene gene;
    private final Gene referenceGene;
    private final NucleotideSequence nucleotides;
    private final String name;

    @Override
    public PredictionContainer call() throws Exception {
        if (Homologous(gene.sequence, referenceGene.sequence)) {
            NucleotideSequence upStreamRegion = GetUpstreamRegion(nucleotides, gene);
            Match prediction = PredictPromoter(upStreamRegion);

            if (prediction != null) {
                return new PredictionContainer(prediction, name);
            }
        }
        return null;
    }
}

@Getter
@RequiredArgsConstructor
private static class PredictionContainer {
    private final Match prediction;
    private final String name;
}
```

Parallel code collecting ExecutorService solution