CAB401   
  
PARALLELIZATION PROJECT REPORT

Dac Duy Anh Nguyen n10603280

Promoter

Table of Contents

[1. Original sequential application 4](#_Toc86519479)

[1.1 What it does 4](#_Toc86519480)

[1.2 How it works 4](#_Toc86519481)

[1.2.1 Important packages 4](#_Toc86519482)

[1.2.2 Sequential 5](#_Toc86519483)

[2. Potential parallelism analysis 6](#_Toc86519484)

[2.1 Profiler analysis 6](#_Toc86519485)

[2.2 Dependencies: 7](#_Toc86519486)

[3. Tools and techniques utilized 8](#_Toc86519487)

[3.1 Hardware specs: 8](#_Toc86519488)

[3.2 Software: 8](#_Toc86519489)

[3.3 Techniques: 8](#_Toc86519490)

[3.3.1 Explicit Threading (in ExplicitThreading.java) 9](#_Toc86519491)

[3.3.2 Parallel stream (in Parallel.java) 11](#_Toc86519492)

[3.3.3 Executor Service (in Parallel.java) 13](#_Toc86519493)

[3.4 Testing 14](#_Toc86519494)

[4. Outcomes 15](#_Toc86519495)

[4.1 ExplicitThreading 15](#_Toc86519496)

[4.2 Parallel stream 15](#_Toc86519497)

[4.3 Executor Service 15](#_Toc86519498)

[5. Difficulties 15](#_Toc86519499)

[5.1 Explicit Threading 15](#_Toc86519500)

[5.2 Parallel Stream 16](#_Toc86519501)

[6. Reflection 16](#_Toc86519502)

[7. References 16](#_Toc86519503)

[8. Appendixes 17](#_Toc86519504)

[8.1 Instructions to run app with each technique 17](#_Toc86519505)

[8.2 Others 17](#_Toc86519506)

# 1. Original sequential application

## 1.1 What it does

The application is utilized to identify different promoters – whose genes are listed in reference list – from the Ecoli bacteria genes.

For more information regarding the biology in this app, referenceGenes.list contains gene name and its gene sequence - the objectives this app needs to find in the bacteria genes. In the provided program, there are 8 reference genes in total. Each letter in the sequence corresponds to an amino acid – a basic building block of a protein sequence.

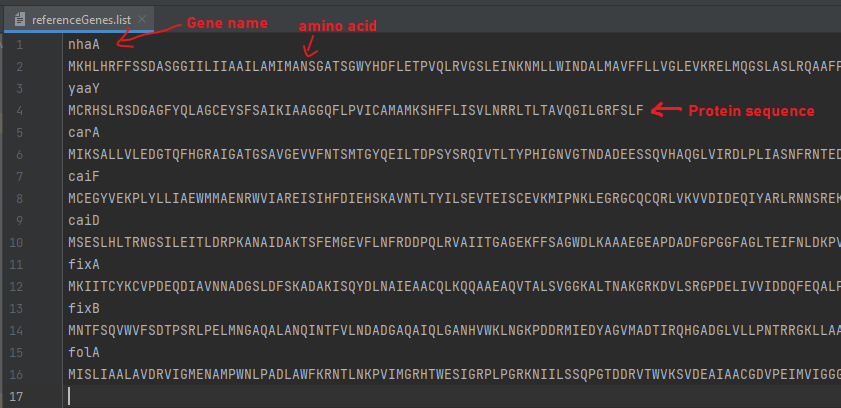


Figure 1 Capture of the referenceGens.list file

In each gene bank file (with .gbk tail), ORIGIN section is the DNA of a bacteria organism. The DNA can contain millions of DNA molecules represented by 4 letter a-t-g-c and triplet of DNA molecules encodes 1 protein. There are 20 different amino acids built from 64 possible combinations of a triplet, and they are the basic building block of proteins. Some sections of DNA don’t correspond directly to proteins so in between proteins, there are upstream region called promoters which are the objects we need to identify.

## 1.2 How it works

### 1.2.1 Important packages

Description and analysis of some important packages:

* SmithWatermanGotoh: contains utility classes that uses Smith-Waterman algorithm with Gotoh’s improvement for biological local pairwise sequence alignment. Note that classes in this package requires an intensive computation power to determine the similar regions between two strings of sequences.
* Sigma70Consensus: initializes the map and prediction of a sigma factor 70 consensus and contains utility functions to record results.
* BioPatterns: contains getBestMatch function which is responsible for searching the best match of a pattern in a nucleotide sequence.
* GenBankRecord: its instance contains a list of gene names, their strand, location and gene sequences and a nucleotide sequence. This package also has Parse method used to extract the mentioned data from a file.

### 1.2.2 Sequential

Through a brief analysis in run() function, the program first loads Ecoli DNA and reference genes using ParseReferenceGenes function.

After that, in the **first** **For** **loop** (number 1), the program iterates through each Ecoli DNA file to retrieve its gene record using Parse function.

Another **inner For loop** (number 2) iterates through the reference genes to compare it with each Ecoli gene in its **inner For loop** (number 3).

The comparation process is handled by Homologous function to determine if 2 genes serve the same purpose. In this case, they are an Ecoli gene from Ecoli records (from For loop 3) and another gene from reference genes (from For loop 2).

Inside the Homologous function, gene sequences are passed to align function of the SmithWatermanGotoh package.

If the Homologous function return true, the program proceeds to extract upstream region of the Ecoli gene by GetUpStreamRegion function and use it to predict if that Ecoli gene is a promoter. If true, the results are recorded in ‘consensus’ variable by adding 1 to the counter of that promoter type and the total matches.

# 2. Potential parallelism analysis

## 2.1 Profiler analysis

From the initial analysis of the given sequential program, Homologous function dominantly consumes 99.7% of the CPU computation. More specifically, it’s the align function in SmithWatermanGotoh class that get called by the Homologous function.

Text

Description automatically generated with medium confidence

Graphical user interface, application

Description automatically generated

Figure 2 CPU hotspot from JProfiler

Since SmithWatermanGotoh function is a complicated algorithm and required a deep understanding of problem to rewrite for better performance, the work of this report mainly focuses to parallelize the app and increase performance at other parts where possible.

More importantly, this app uses 3-nested for loop which is being executed sequentially and includes many computations, read and write operations inside the for loops. As a consequence, the computing time takes a substantial amount of time and could be reduced if applying parallelization.

## 2.2 Dependencies:

sigma70\_pattern is assigned value from Sigma70Definition.getSeriesAll\_Unanchored(). But the method itself getSeriesAll\_Unanchored() is also static which will cause data race when there are multiple threads trying to access sigma70\_pattern. So to parallelize, I made it a ThreadLocal variable to store data individually for each thread (except for Explicit Threading technique).

Control dependency: the condition statement in ‘if’ affects whether the code in it executes or not. However, the properties in Gene class are not static meaning values of its instance are separated from each thread so it can be safe to implement parallelization for this part.

2 data dependencies are also identified in the most inner section of the For loops. Specifically, they are of True Dependence type in which one statement reads a value written by an earlier statement. In this case, they are upStreamRegion and prediction.

Text

Description automatically generated

Figure 3 Control (red) and data dependencies (orange)

# 3. Tools and techniques utilized

## 3.1 Hardware specs:

* CPU: Intel Core i7-8565U @1.8GHz
* 4 cores
* 8 virtual cores
* 8GB RAM
* Cache
  + L1 Data 4x32KB, 8-way
  + L1 Instruction 4x32KB, 8-way
  + L2 4x256KB, 4-way
  + L3 8MB, 16-way

## 3.2 Software:

* Integrated development environment: IntelliJ
* Java 15
* Run on Window Operating System
* Profiler tool: Jprofiler

## 3.3 Techniques:

In this project, I restructured the For loops code block to parallelize (except for Explicit threading technique). I made For loop 2 wrap For loop 1 and 3. Now, the order of the 3 For loops looks like this:

2: referenceGene -> 1: filename -> 3: gene

Text

Description automatically generated

Figure For loops initial order

Text

Description automatically generated

Figure For loops order changed

The aim of this transformation is to increase spatial locality by reducing gap time between accessing ‘record’ in For loop 1 and 3 and therefore, increase the number of cache hits for ‘record’ when transiting from For loop 1 to 3

While this puts referenceGene further from the inner part (*if (Homologous(…){ …}*), the high frequency in calling the variable will keep its memory location stored in cache.

There are 3 parallelization approaches I have implemented in this project:

* Explicit Threading
* Parallel Stream
* Executor Service

### 3.3.1 Explicit Threading (in ExplicitThreading.java)

This is the first method I attempted for this application. From the sequential code, I thought the application can run faster by parallelizing and letting each thread handles one single input file instead of having them queue and running one by one. Therefore, I estimated this method can reduce the execution time by approximately 4 times (with 4 threads running parallelly to process 4 given Ecoli text files at once).

To apply explicit threading method, I created a new class called ExplicitThreading so that it can spawn a new thread of Sequential class by a separated For loop. To distribute the workload, that For loop also pre-processes ListGenBankFiles and assign each file for each thread. With this change, the Sequential class now only handle one file in each thread and 4 of them can run simultaneously.

With this approach, there are several changes in the original code structure. In order to prepare and extract the necessary data and add them as parameters to each Sequential call, 2 functions, ProcessDir and ListGenBankFiles, from Sequential class were moved down to my newly created class.

Text

Description automatically generated

Figure 6 Newly created ExplicitThreading Class

New private variables and constructors were also added in Sequential to keep exclusive data from ExplicitThreading for each particular thread.

Text

Description automatically generated

Figure 7 New private variables and constructors

As for critical part with most write operations, I used ReentrantLock to only allow 1 thread to access to it in any given time.

Text

Description automatically generated

Figure 8 Critical part is protected with a lock

### 3.3.2 Parallel stream (in Parallel.java)

Parallel Stream is an API that allows us to create multiple parallel streams, perform operations on them parallelly and as a result, utilize multiple processor cores. This technique especially comes in handy when the source of a stream is a Collection or an Array. In this case, we use parallelStream() method.

#### 3.3.2.1 Parallel stream 3rd For loop ( runParallelStream3rd() )

As analyzed from the Jprofiler hotspot (Figure 2 CPU hotspot from JProfiler), 99.7% of execution time is spent on Homologous function so I decided to just parallelize the Homologous part using parallel stream.

#### 3.3.2.2 Parallel stream with pre-process ( runParallelStreamWithPrep() )

As analyzed from the Jprofiler hotspot (Figure 2 CPU hotspot from JProfiler), 99.7% of execution time is spent on Homologous function so I decided to just parallelize the Homologous part using parallel stream. In addition, loops distribution is applied, which turns a block of 3 For loops into one block of 3 For loops to pre-process files, store outcomes and another For loop to actually run Homologous check and promoter prediction.

In order to increase efficiency in storing data in the pre-process phase and accessing it in the second For loop, I created a new Class called TaskHandler which takes each combination of reference gene, gene and gen bank record and then store it as an item in a List for later use.

Text

Description automatically generated

Figure 9 TaskHandler class

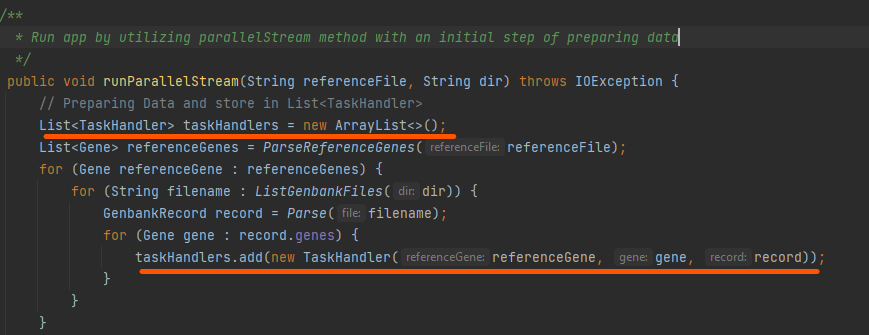


Figure 10 Pre-process code with List<TaskHandler> taskHandlers to store data

Then, parallel stream can easily be implemented on a List as shown in Figure 11 Apply parallelStream() on the hotspot block. Here, the tasks in List<TaskHandler> will be assigned to a thread and processed as soon as one becomes free. ‘filter’ expression of parallelStream() is also utilized instead of if condition to check whether if 2 gene sequences are Homologous or not.

Text

Description automatically generated

Figure 11 Apply parallelStream() on the hotspot block

Note that the codes involve consensus, the hotspot of many write statements during the process, are now placed in a synchronized function to ensure only one thread can write on its value in any given moment and therefore, a correct final result.

Graphical user interface, text

Description automatically generated

Figure 12 synchronized addConsensus() function

### 3.3.3 Executor Service (in Parallel.java/runExecutorService() )

Executor Service is a JDK API that enables app to create a pool of threads and provides an API to assign tasks to it as threads become free. Number of threads is assigned when declaring an ExecutorService object. In this solution, tasks are objects of a class – RunnableTask – implementing Runnable Interface and each of which will keep a combination of referenceGene, gene, and record. Following the same approach with Parallel stream technique, thread pool will only be used for running Homologous code.

Text

Description automatically generated

Figure 13 RunnableTask class

In Executor service, Future represents pending completion of a task. To assign tasks to a pool, these objects are first submitted to an executorService object, and they will then be added to a List<Future>.

Text

Description automatically generated

Figure 14 Pre-process data: collecting List<Future>

After having completed to getting all the futureTasks in List<Future>, the program iterates though each item in List<Future> and execute them asynchronously using the pool of threads we have declared in executorService.

## 3.4 Testing

In my UnitTests (CompareTests.java), I declare a defaultConsensus variable of type HashMap<String, String> to store default result in the original Sequential run with given default input dataset. This helps speed up the testing time when comparing the final result of new parallelization techniques with Sequential code’s.

Text

Description automatically generated

Figure 15 Declare default consensus function

Tests in this app includes testing all 3 techniques.

All the tests are conducted by going through each entry of consensus and comparing the 2 values of each entry’s key (Figure 15 A test unit example)

# 4. Outcomes

First of all, all the parallelization techniques have achieved correct outcomes from the same input dataset. Below are the speedups of all techniques (on average of 10 runs each) mentioned in this report with best sequential time = 200s:

Table

Description automatically generated

Figure 16 Speedup of the parallelization techniques

Chart, line chart

Description automatically generated

Figure 17 Speedup chart

CPU utilization analysis 2 PS techniques

Overall, Parallel Stream and Executor Service technique achieved significantly better performance than Explicit Threading and the CPU is utilized at high percentage - 90% on average - during their execution periods.

2 ss of CPU, 12 threads by ES and PS

Parallel stream and Executor service technique clearly has more advantages than Explicit Threading in this app since it automates the process of distributing tasks between threads and then combine the results after finishing the task.

Explicit threading can spawn more threads as we get inside the program with more For loops to increase performance, but at the same time it also gets more complicated with data and control dependencies.

Change loop order, increase locality but have not been tested run compare performance / execution time

## 4.1 ExplicitThreading

## 4.2 Parallel stream

## 4.3 Executor Service

# 5. Difficulties

## 5.1 Explicit Threading

The app can save some computation power if ParseReferenceGene is moved down to my new class ExplicitThreading. Because then, the referenceFile can now be processed once in ExplicitThreading and used multiple times by adding the variable as a parameter in every thread call instead of compute it every time a new Thread is call.

However, ParseReferenceGene depends on ‘consensus’ and functions, which makes the program much more complex. While attempting to do this, I got a data race error and did not manage to fix it.

Exception in thread "0" java.lang.NullPointerException: Cannot invoke "qut.Sigma70Consensus.addMatch(edu.au.jacobi.pattern.Match)" because the return value of "java.util.HashMap.get(Object)" is null

at qut.Sequential\_for\_explicit\_threading.run(ExplicitThreading.java:93)

So, this further development is stopped.

## 5.2 Parallel Stream

**Inefficient performance while parallelize every For Loop:**

My first approach to this Parallel Stream technique was to apply parallelStream() to every List instead of iterating through every item in List in every For loop without the pre-processing step.

However, later I found out that applying parallelStream() to every For loop does not increase performance as the app still spawns a pre-determined number of threads. So, this may be counterproductive because the app now has more overheads to run parallelStream() 3 times which brings no benefit in this case. The execution time is **62 seconds for 8 threads or 61 seconds for 12 threads**, which is much greater than the proposed Parallel Stream technique above that used parallelStream() once.

Text

Description automatically generated

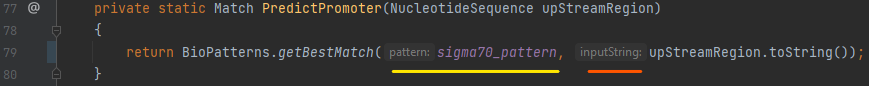
Figure 18 Initial approach for parallel stream: using parallelStream() 3 times

**Fixing data race while parallelStream() the app.**

While parallelizing the app with parallel stream, my app ran into an IndexOutOfBound error. So I speculated this is a data race issue. I followed the error report and found the key cause of this error which is in PredictPromoter function.

A screenshot of a computer

Description automatically generated with medium confidence



Data race in sigma70\_pattern between difference threads has cause IndexOutOfBound error. It could not be upStreamRegion.toString() because it is a String type which is not a popular cause of IndexOutOfBound error. And sigma70\_pattern is assigned values from a static function -Sigma70Definition.getSeriesAll\_Unanchored(0.7) - at the beginning of the program so its properties contain a lot of Integer and Array as we can see in the below code snippet in Sigma70Definition.java.

Text

Description automatically generated

Figure 19 getSeriesAll\_Unanchored() function

After doing some research on the internet, I managed to solve it by making sigma70\_pattern a ThreadLocal<Series> variable because ThreadLocal is a special construct allows to store data that will be accessible only by the thread running it.

## 5.3 Executor Service

# 6. Reflection

**To parallelize an app well, we must have a deep understanding of the app** such as its data structure, purposes of the app in general, functions, classes, noticing small details like data type as well as knowing various parallelization techniques to use the most suitable one. For example, ThreadLocal protects data race between threads, but we do not apply it to all variables, but only to the exposed ones.

**Complicated program should often be used with implicit threading instead of explicit threading**. The reason for this is that threads can run and finished in different time, and we have little control over this. Therefore, it gets more complicated to organize and add tasks to threads so as to make the most of all CPU cores power. On the other hand, implicit threading solves this problem by automating task distribution for us.

**Finding the most optimized parallelization techniques requires much hand-on work**. There is no superior parallelization method that fits all cases or has the best performance. And it may need change to the original program to further optimize the runtime. For instance, changing program architecture or algorithm. Therefore, the performance of each technique varies in each program, so it needs trial and error to find the best parallelization technique.

# 7. References

<https://www.baeldung.com/java-executor-service-tutorial>

<https://www.baeldung.com/java-when-to-use-parallel-stream>

<https://docs.oracle.com/javase/tutorial/collections/streams/parallelism.html>

<https://www.tabnine.com/code/java/methods/java.util.Collection/parallelStream>

<https://passel2.unl.edu/view/lesson/3ccee8500ac8/6#:~:text=A%20triplet%20code%20could%20make,of%20all%2020%20amino%20acids>.

<https://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html#:~:text=A%20class%20that%20implements%20Runnable,and%20no%20other%20Thread%20methods>.

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/ExecutorService.html>

<https://www.vogella.com/tutorials/JavaConcurrency/article.html>

# 8. Appendixes

## 8.1 Instructions to run app with each technique

## 8.2 Others

Text

Description automatically generated

Figure 20 A unit test

parallelStream 111 50s – 76s

A picture containing chart

Description automatically generated

Diagram, timeline

Description automatically generated

101 57s 78s 79s

Diagram

Description automatically generated

Chart

Description automatically generated with medium confidence

011 72 less overhead?

Diagram

Description automatically generated

A screenshot of a computer

Description automatically generated with medium confidence

001 77

Diagram

Description automatically generated

Chart

Description automatically generated

110 83

Thread waiting for other thread to finish

Timeline

Description automatically generated

A picture containing chart

Description automatically generated

100 108

Too many thread waiting

Timeline

Description automatically generated with medium confidence

Timeline

Description automatically generated

010 101s

Only create 5 threads

Timeline

Description automatically generated A picture containing timeline

Description automatically generated

Sheep1 62 77 78 85 50 60

Sheep2 52

Sheep2 PS 12 threads average 63s

Me 8 threads average 65s

Me 12 threads average 70s

Me 8 threads without specify average 84s