

CAB401

PARALLELIZATION PROJECT REPORT

Dac Duy Anh Nguyen n10603280

Promoter

Table of Contents

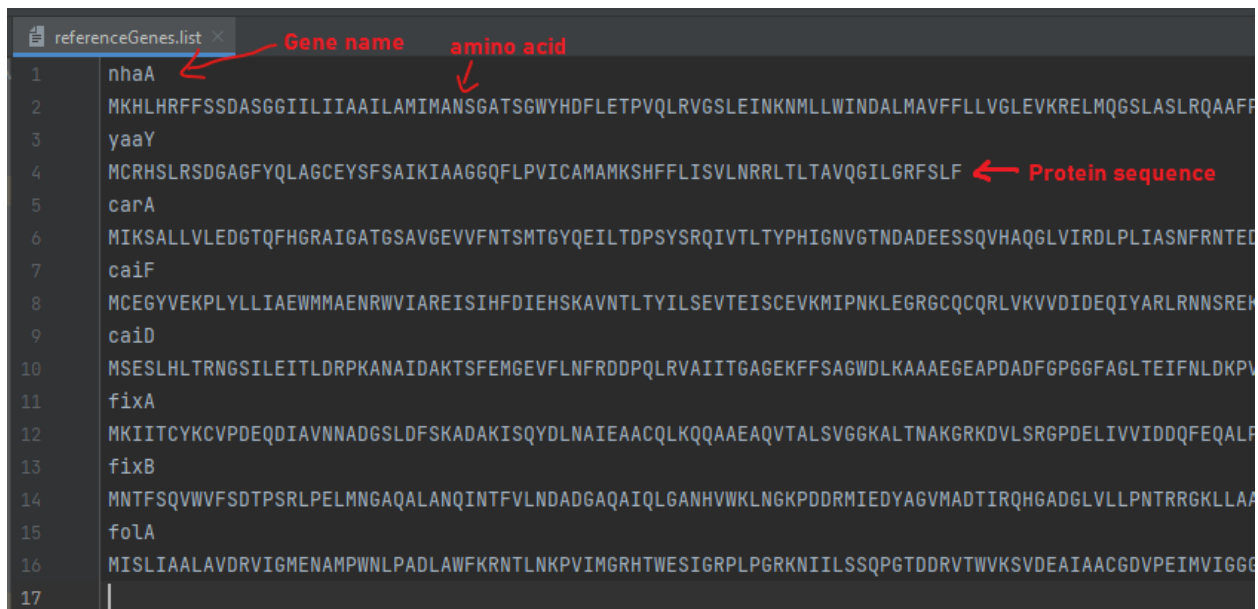
1. Original sequential application	3
1.1 What it does	3
1.2 How it works	3
1.2.1 Important packages	3
1.2.2 Sequential	4
2. Potential parallelism analysis	5
2.1 Profiler analysis	5
2.2 Dependencies	6
3. Tools and techniques utilized	7
3.1 Hardware specs	7
3.2 Software	7
3.3 Techniques	7
3.3.1 Explicit Threading (in ExplicitThreading.java)	8
3.3.2 Parallel stream (in Parallel.java)	10
3.3.3 Executor Service (Parallel.java - runExecutorService)	13
3.4 Testing (CompareTests.java)	14
4. Outcomes	14
5. Limitations	17
6. Difficulties	17
6.1 Explicit Threading	17
6.2 Parallel Stream	18
6.3 Executor Service	20
7. Reflection	20
8. References	21
9. Appendixes	22
9.1 Instructions to run app with each technique	22
9.2 Others	22

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.



	Gene name	amino acid
1	nhaA	MKHLHRRFFSSDASGGIILIIAAILAMIMANS
2		GATSGWYHDFLETPVQLRVGSLEINKNMLLWINDALMAVFFLLVGLVVKRELMQGS
3	yaaY	SLASLRQAFF
4	MCRHSLRSDGAGFYQLAGCEYSFSAIKIAAGGQFLPVICAMAMKSHFFLISVLNRR	TLTAVQGILGRFSLF ← Protein sequence
5	carA	
6	MIKSALLVLEDGTQFHGRAIGATGS	AVGEVFNSTMTGYQEILTDPSYSRQIVTLTYPHIGNVGTNDADEESSQVHAQGLVIRDLPLIASNFRNTE
7	caiF	
8	MCEGYVEKPLYLLIAEWMAENRWVIAREISIHFDIEHSAVNTLT	YILSEVTEISCEVKMIPNKLEGRGCQCQRLVKVVDIDEQIYARLRNNSREH
9	caiD	
10	MSESLHLTRNGSILEITLDRPKANAIDAKTSFEMGEVFLNFRDDPQLRVAIITGAGEKFFSAGWDLKAAAE	GEAPDADFPGPGFAGLTEIFNLDPV
11	fixA	
12	MKIITCYKCVPEQDIAVNNADGSLDFSKADAKISQYDLNAIEAACQLKQAAEAQVTALSVGGKALTNAKGRKDVLSRGPDELIVVIDDQFEQALF	
13	fixB	
14	MNTFSQVWVSDTPSRLPELMNGAALANQINTFVLDADGAQAIQLGANHVWKLNGKPD	DRMIEDYAGVMADTIRQHGDGLVLLPNTRRGKLLAA
15	folA	
16	MISLIAALAVDRVIGMENAMPWNLPADLAWFKRNTLNKPVIMGRHTWESIGRPLPGRKNIILSSQPGTDDRVTWVKS	VDIAACGDVPEIMVIGG
17		

Figure 1 referenceGenes.list

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 (Lee & Hain, 2021). 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 comparison 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.

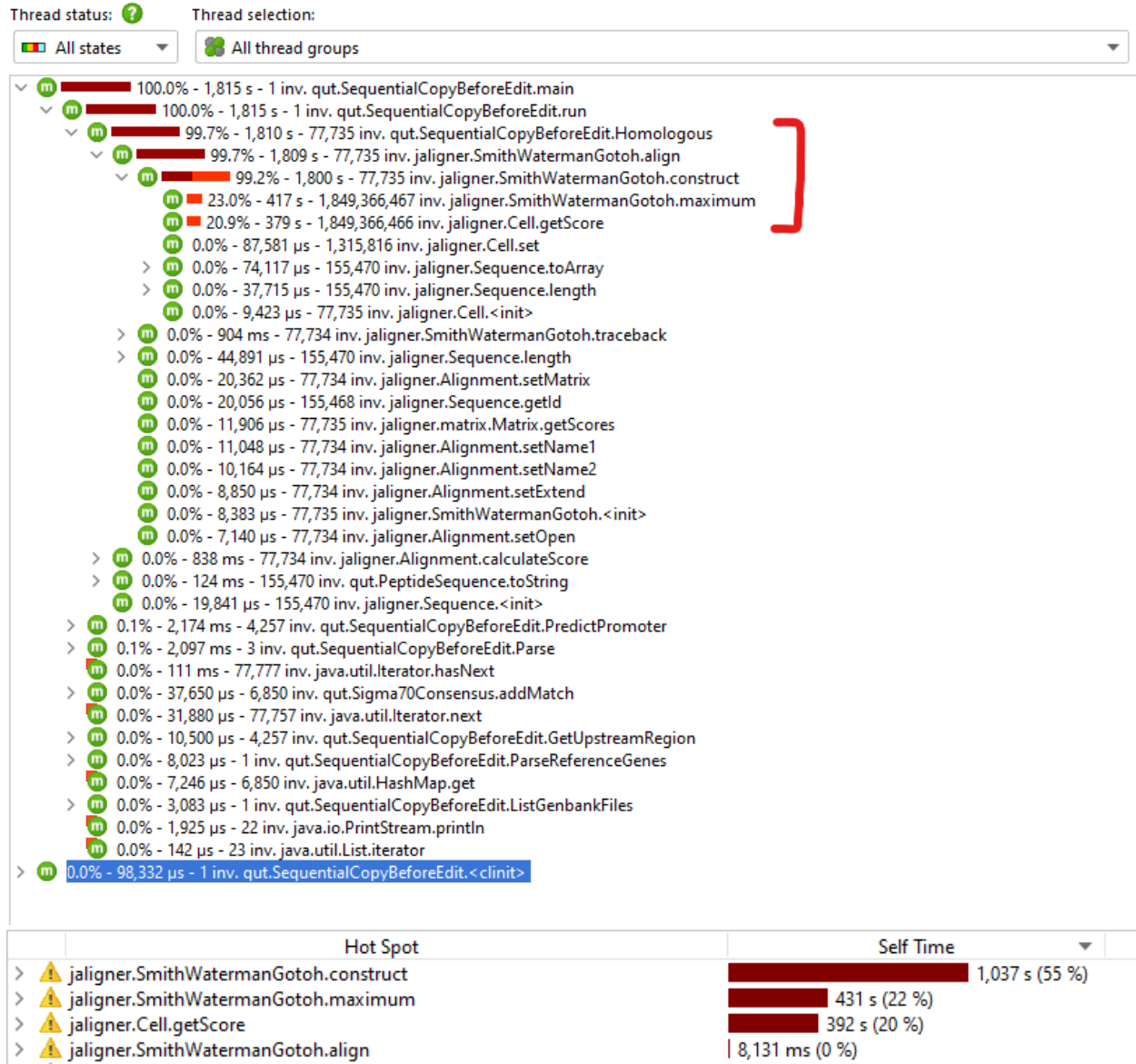


Figure 2 CPU hotspot from JProfiler

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

More importantly, this app uses 3-nested for loop which is being executed sequentially and contains both heavy (Homologous function) and light operations (extracting genes, gen bank records), 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 we apply parallelizing and restructuring.

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 (orange underlined): 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 (yellow underlined) 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.

```
110 // Initial order
111 for (String filename : ListGenbankFiles( dir, dir)) {
112     System.out.println(filename);
113     GenbankRecord record = Parse( file: filename);
114     for (Gene referenceGene : referenceGenes) {
115         System.out.println(referenceGene.name);
116         for (Gene gene : record.genes) {
117             if (Homologous( A: gene.sequence, B: referenceGene.sequence)) {
118                 NucleotideSequence upStreamRegion = GetUpstreamRegion(
119                     dna: record.nucleotides, gene: gene);
120                 Match prediction = PredictPromoter( upStreamRegion: upStreamRegion);
121                 if (prediction != null) {
122                     consensus.get(referenceGene.name).addMatch( match: prediction);
123                     consensus.get("all").addMatch( match: prediction);
124                 }
125             }
126         }
127     }
128 }
```

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

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

```
248 //      Order changed
249 for (Gene referenceGene : referenceGenes) {
250     System.out.println(referenceGene.name);
251     for (String filename : ListGenbankFiles( dir: dir)) {
252         System.out.println(filename);
253         GenbankRecord record = null;
254         try { record = Parse( file: filename);
255         } catch (IOException e) { e.printStackTrace(); }
256         for (Gene gene : record.genes) {
257             if (Homologous( A: gene.sequence, B: referenceGene.sequence)) {
258                 NucleotideSequence upStreamRegion = GetUpstreamRegion( dna: record.nucleotides, gene: gene);
259                 Match prediction = PredictPromoter( upStreamRegion: upStreamRegion);
260                 if (prediction != null) {
261                     consensus.get(referenceGene.name).addMatch( match: prediction);
262                     consensus.get("all").addMatch( match: prediction);
263                 }
264             }
265         }
266     }
267 }
```

Figure 4 For loops order changed (Parallel.java)

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.


```

129 ▶ class ExplicitThreading {
130     private static final HashMap<String, Sigma70Consensus> consensus = new HashMap<>();
131     @ private static void ProcessDir(List<String> list, File dir)
132     { ... }
140     @ private static List<String> ListGenbankFiles(String dir)
141     { ... }
146 ▶ public static void main(String[] args) throws InterruptedException {
147
148     long startTime = System.currentTimeMillis();
149     List<Thread> threads = new ArrayList<>();
150     List<String> listGenBankFiles = ListGenbankFiles( dir: "src/Ecoli");
151     // Spawn a thread for each file
152     for ( int i=0; i < listGenBankFiles.size(); i++) {
153         String ecoliFilename = listGenBankFiles.get(i);
154         Runnable task = new Sequential_for_explicit_threading(
155             ecoliFilename: ecoliFilename, referenceFile: "src/referenceGenes.list");
156         Thread thread = new Thread( target: task); Changed params
157         thread.setName(String.valueOf( i: i));
158         thread.start();
159         threads.add(thread);
160     }
161     for (Thread thread : threads) thread.join();
162
163     long timeElapsed = System.currentTimeMillis() - startTime;
164     System.out.println("Execution time: " + timeElapsed/1000 + " s");
165 }

```

Figure 5 Newly created ExplicitThreading Class (ExplicitThreading.java)

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

```

17 // ===== New =====
18 private final String ecoliFilename;
19 private static String referenceFile;
20 static ReentrantLock lock1 = new ReentrantLock();
21
22 public Sequential_for_explicit_threading(String ecoliFilename, String referenceFile) {
23     this.ecoliFilename = ecoliFilename;
24     this.referenceFile = referenceFile;
25 }
26 // =====

```

Figure 6 New private variables and constructors (ExplicitThreading.java)

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

```

for (Gene referenceGene : referenceGenes) {
    System.out.println(referenceGene.name);
    for (Gene gene : record.genes) {
        if (Homologous(A: gene.sequence, B: referenceGene.sequence)) {
            lock1.lock();
            NucleotideSequence upStreamRegion = GetUpstreamRegion( dna: record.nucleotides, gene: gene);
            Match prediction = PredictPromoter( upStreamRegion: upStreamRegion);
            if (prediction != null) {
                consensus.get(referenceGene.name).addMatch( match: prediction);
                consensus.get("all").addMatch( match: prediction);
            }
            lock1.unlock();
        }
    }
}

```

Figure 7 Critical part is protected with a lock (ExplicitThreading.java)

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 (baeldung, 2021).

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.

```

115 for( Gene referenceGene :referenceGenes) {
116     System.out.println(referenceGene.name);
117     for (String filename : ListGenbankFiles( dir: dir)) {
118         System.out.println(filename);
119         GenbankRecord record = null;
120         try { record = Parse( file: filename); }
121         catch (IOException e) { e.printStackTrace(); }
122         List<Gene> genes = record.genes;
123         GenbankRecord finalRecord = record;
124         genes.parallelStream().forEach( action: gene -> {
125             if (Homologous( A: gene.sequence, B: referenceGene.sequence)) {
126                 NucleotideSequence upStreamRegion = GetUpstreamRegion(
127                     dna: finalRecord.nucleotides, gene: gene);
128                 Match prediction = PredictPromoter( upStreamRegion: upStreamRegion);
129                 if (prediction != null)
130                     addConsensus( name: referenceGene.name, prediction: prediction);
131             }
132         });
133     }
134 }

```

Figure 8 parallelStream() 3rd For loop

Note that the code block processing consensus value, the hotspot of many write statements during the process is 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.

```

101 public synchronized void addConsensus(String name, Match prediction) {
102     consensus.get(name).addMatch( match: prediction);
103     consensus.get("all").addMatch( match: prediction);
104 }

```

Figure 9 synchronized addConsensus() function

3.3.2.2 Parallel stream with pre-processing - runParallelStreamWithPrep

In addition to the approach in 3.3.2.1 Parallel stream 3rd For loop - runParallelStream3rd, **Divide and Conquer** technique is applied, which turns a block of 3 For loops into 2 sub-problems: one consists of block of 3 For loops to pre-process files, store outcomes (Figure 11 Pre-process code with List<TaskHandler> taskHandlers to store data) and utilizing ParallelStream to run Homologous check and promoter prediction (Figure 12 Apply parallelStream() on taskHandlers).

In order to increase efficiency in storing data in the pre-process phase and accessing it in the second sub-problem, I created a new Class called TaskHandler (Figure 10 TaskHandler class (TaskHandler.java)) 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.

```

1  package qut;
2
3  public class TaskHandler {
4      private final Gene referenceGene;
5      private final Gene gene;
6      private final GenbankRecord record;
7
8      public TaskHandler(Gene referenceGene, Gene gene, GenbankRecord record) {
9          this.referenceGene = referenceGene;
10         this.gene = gene;
11         this.record = record;
12     }
13
14     public Gene getReferenceGene() { return referenceGene; }
17     public Gene getGene() { return gene; }
20     public GenbankRecord getRecord() { return record; }
23 }

```

Figure 10 TaskHandler class (TaskHandler.java)

```

147 public void runParallelStreamWithPrep(String referenceFile, String dir, int threadNum) throws IOException
148 // Preparing Data and store in List<TaskHandler>
149 List<TaskHandler> taskHandlers = new ArrayList<>();
150 List<Gene> referenceGenes = ParseReferenceGenes( referenceFile: referenceFile);
151 for (Gene referenceGene : referenceGenes) {
152     for (String filename : ListGenbankFiles( dir: dir)) {
153         GenbankRecord record = Parse( file: filename);
154         for (Gene gene : record.genes) {
155             taskHandlers.add(new TaskHandler( referenceGene: referenceGene, gene: gene, record: record));
156         }
157     }
158 }

```

Figure 11 Pre-process code with List<TaskHandler> taskHandlers to store data

Then, parallel stream can easily be implemented on a List as shown in Figure 12 Apply parallelStream() on taskHandlers. 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.

```

163 System.out.println("Now run on " + threadNum + " threads.");
164 System.setProperty("java.util.concurrent.ForkJoinPool.common.parallelism", Integer.toString( threadNum));
165 taskHandlers.parallelStream()
166     .filter( predicate: task -> Homologous( A: task.getGene().sequence, B: task.getReferenceGene().sequence))
167     .forEach( action: task -> {
168         NucleotideSequence upStreamRegion = GetUpstreamRegion( dna: task.getRecord().nucleotides, gene: task.getGene());
169         Match prediction = PredictPromoter( upStreamRegion: upStreamRegion);
170         if (prediction != null) addConsensus( name: task.getReferenceGene().name, prediction: prediction);
171     });

```

Figure 12 Apply parallelStream() on taskHandlers

3.3.3 Executor Service (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 with pre-processing technique, thread pool will only be used for running Homologous code.

```
217 public class RunnableTask implements Runnable {
218     private final Gene referenceGene;
219     private final Gene gene;
220     private final GenbankRecord record;
221
222     public RunnableTask(Gene referenceGene, Gene gene, GenbankRecord record) {
223         this.referenceGene = referenceGene;
224         this.gene = gene;
225         this.record = record;
226     }
227     @Override
228     public void run() {
229         if (Homologous(A: gene.sequence, B: referenceGene.sequence)) {
230             NucleotideSequence upStreamRegion = GetUpstreamRegion(dna: record.nucleotides, gene: gene)
231             Match prediction = PredictPromoter(upStreamRegion: upStreamRegion);
232             if (prediction != null) {
233                 addConsensus(name: referenceGene.name, prediction: prediction);
234             }
235         }
236     }
237 }
```

Figure 13 RunnableTask class

In Executor service, Future represents pending completion of a task (Oracle, 2021). 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>.

```
193 for (Gene referenceGene : referenceGenes) {
194     for (String filename : ListGenbankFiles(dir: dir)) {
195         GenbankRecord record = Parse(file: filename);
196         for (Gene gene : record.genes) {
197             Future futureTask = executorService.submit(
198                 task: new RunnableTask(
199                     referenceGene: referenceGene,
200                     gene: gene, record: record));
201             futureTasks.add(futureTask);
202         }
203     }
204 }
```

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 (CompareTests.java)

In my Tests, 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 assuming input dataset is not changed. Nevertheless, tests that compares consensus result from an actual Sequential run is also covered.

```
14     private HashMap<String, String> defaultConsensus;  
15  
16     @BeforeEach  
17     void declareDefaultConsensus() {  
18         defaultConsensus = new HashMap<>();  
19         defaultConsensus.put("all", " Consensus: -35: T T G A C A gap: 17.6 -10: T A T A A T (5430 matches)");  
20         defaultConsensus.put("fix8", " Consensus: -35: T T G A C A gap: 17.7 -10: T A T A A T (965 matches)");  
21         defaultConsensus.put("carA", " Consensus: -35: T T G A C A gap: 17.7 -10: T A T A A T (1079 matches)");  
22         defaultConsensus.put("fixA", " Consensus: -35: T T G A C A gap: 17.6 -10: T A T A A T (896 matches)");  
23         defaultConsensus.put("caiF", " Consensus: -35: T T C A A A gap: 18.0 -10: T A T A A T (11 matches)");  
24         defaultConsensus.put("caiD", " Consensus: -35: T T G A C A gap: 17.6 -10: T A T A A T (550 matches)");  
25         defaultConsensus.put("yaaY", " Consensus: -35: T T G T C G gap: 18.0 -10: T A T A C T (4 matches)");  
26         defaultConsensus.put("nhaA", " Consensus: -35: T T G A C A gap: 17.6 -10: T A T A A T (1879 matches)");  
27         defaultConsensus.put("foLA", " Consensus: -35: T T G A C A gap: 17.5 -10: T A T A A T (46 matches)");  
28     }
```

Figure 15 Declare default consensus function

Tests includes testing all 3 techniques. All the tests are conducted by going through each entry of consensus and comparing the 2 values correlating with the same entry key.

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 of 200 seconds:

Number of cores	Linear Speedup	Explicit Threading Speedup	Parallel Stream (3rd For loop) Speedup	Parallel Stream (with pre-processing) Speedup	Executor Service Speedup
1	1	1.00	1.74	1.61	1.05
2	2		2.13	1.57	1.56
3	3		2.67	2.25	2.27
4	4	1.89	2.53	2.41	2.63
5	5		2.63	2.86	3.57
6	6		2.86	2.82	3.03
7	7		3.13	2.67	3.39
8	8		3.85	3.45	3.85

Figure 16 Speedup of the parallelization techniques

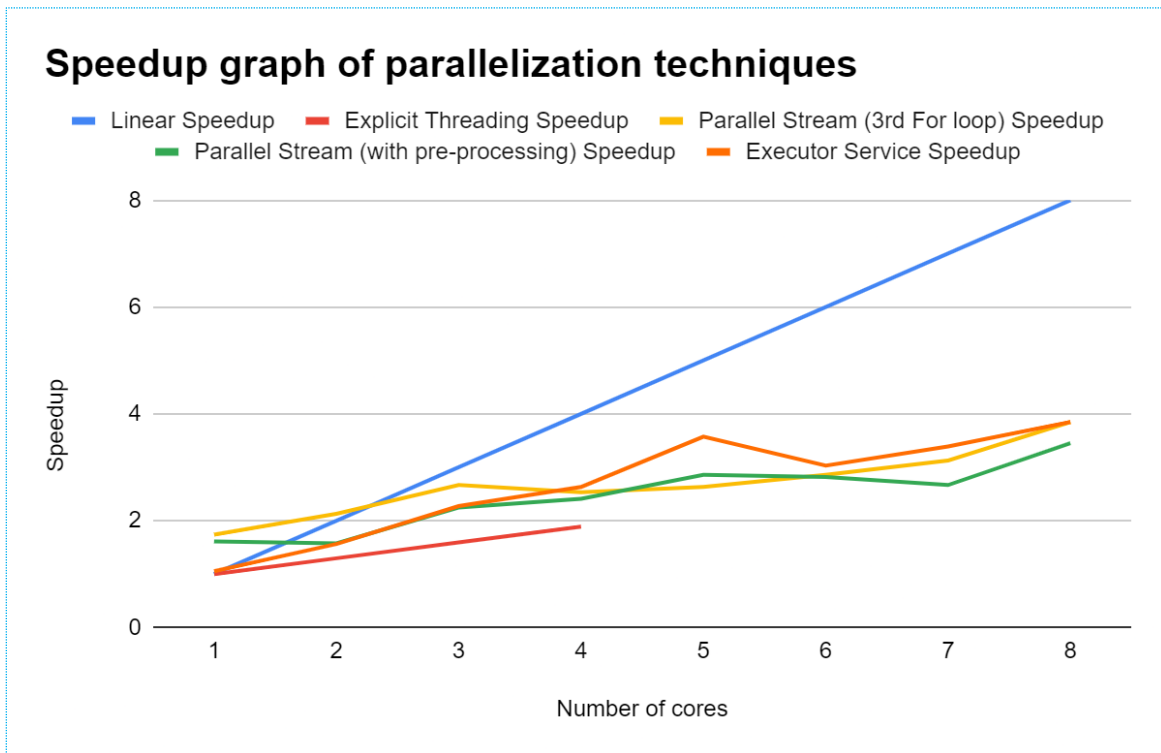


Figure 17 Speedup chart

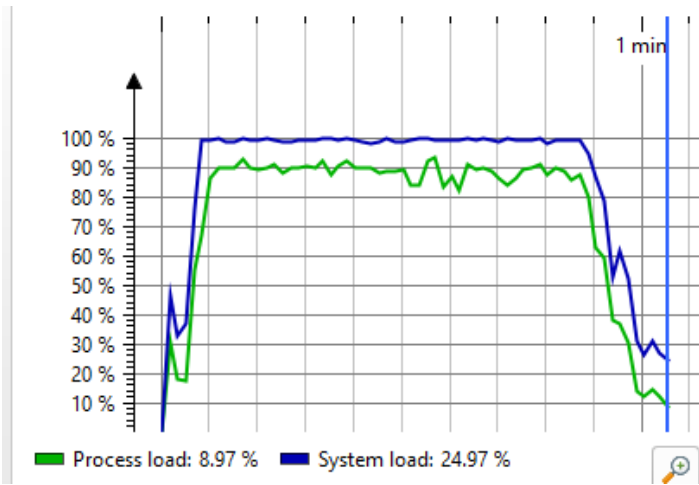


Figure 18 CPU utilization of ParallelStream with pre-processing running 12 threads

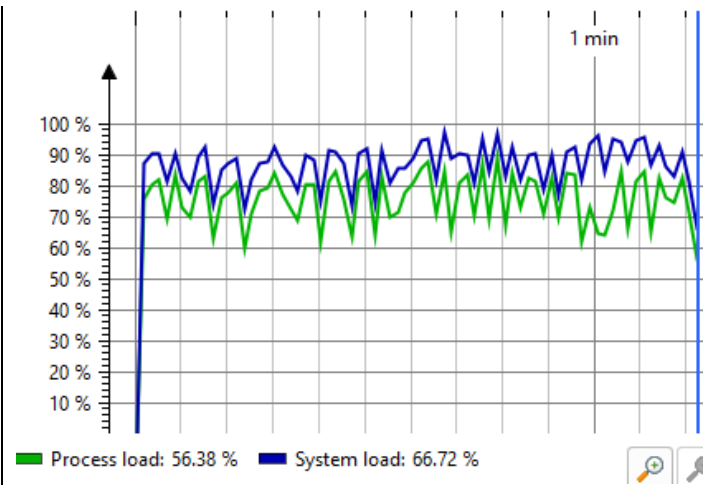


Figure 19 CPU utilization of ParallelStream 3rd For loop running 12 threads

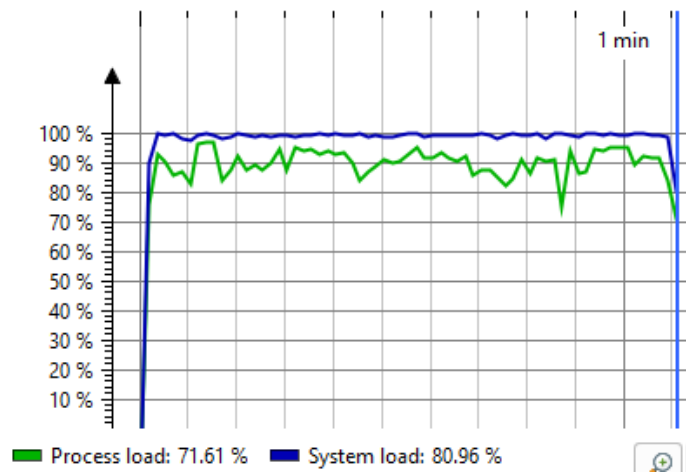


Figure 20 CPU utilization of Executor Service running 12 threads

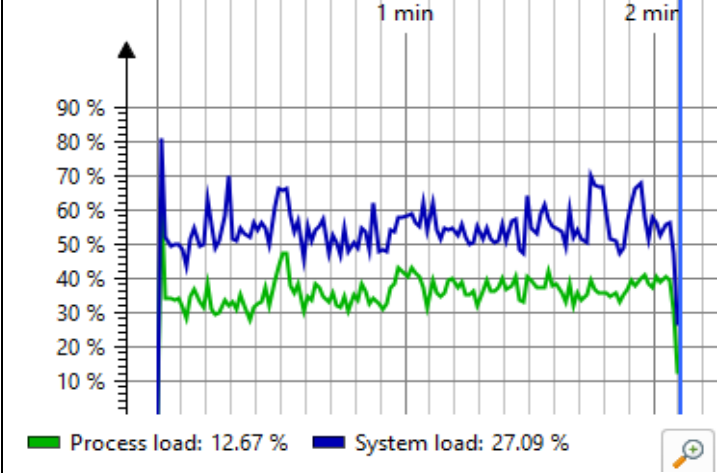


Figure 21 CPU utilization of Explicit Threading running 4 threads

Overall, from the above figures, Parallel Stream with pre-processing and Executor Service techniques achieved significantly better performance than Explicit Threading since CPU process load is utilized consistently at high percentage (>98% on average) during their execution periods. The reason for this is because data has been pre-processed, which enables constant input of data to the CPU cores, for checking Homologous to be specific.

However, at 12 threads, their speedup lines still leave a big gap with linear speedup line because the cost of overheads of creating threads and neckline issue in recording 'consensus'. The more threads there are, the more overheads of creating threads and time for threads to wait and write to 'consensus'.

Another interesting point is that, with no pre-process data, the CPU utilization seen in Explicit Threading and Parallel Stream 3rd For loop fluctuates strongly in a range of up to 20% due to the fact that program having to switching back and forth repeatedly between processing gene, referenceGene, record (which does not require much CPU power but inadvertently creates neckline) and checking Homologous (which requires a lot of CPU power). This can be observed more clearly when we compare Threads' states in Parallel Stream with pre-processing and Parallel Stream 3rd For loop examples. Despite that, Parallel Stream 3rd For loop achieved a slightly better speedup performance than Parallel Stream with pre-processing

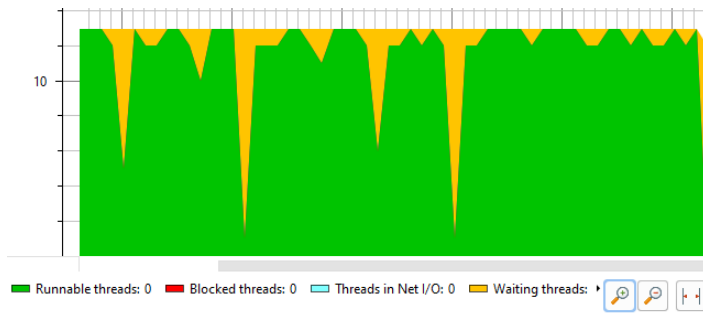


Figure 22 Threads state in PS 3rd For loop

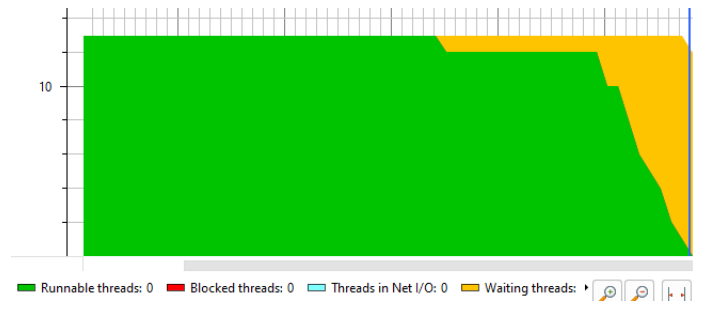


Figure 23 Threads state in PS with pre-processing

During the execution time, threads in Parallel Stream 3rd For loop had to turn to Waiting state many times to wait for data preparation (gene, referenceGene, record) while they run continuously in Parallel Stream with pre-processing until running out of tasks

With 8 cores, both Parallel stream and Executor service techniques has achieved similar results and reached approximately 3.5-4 times speedup.

To conclude, Parallel stream and Executor service techniques clearly have 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. While Explicit threading is capable of spawning more threads as we get inside the program to increase performance, it also gets more complicated with data and control dependencies at the same time.

5. Limitations

Changing For loop order is expected to increase locality and thus performance but this has not been tested and compared to unchanged version in Parallel Stream and Executor Service.

In each Homologous check, recording consensus result could add to an empty slot in an array and then be summed up after finishing the promoter identification phase. This would remove the neckline issue and increase performance of the app. So, this is a potential improvement for the app which I hope to discover.

6. Difficulties

6.1 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.

6.2 Parallel Stream

Inefficient performance while parallelStream() 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. So after discovering this, I only used parallelStream once in each run.

```
221 referenceGenes.parallelStream().forEach( action: referenceGene -> {
222     System.out.println(referenceGene.name);
223     List<String> filenames = ListGenbankFiles( dir: dir);
224     filenames.parallelStream().forEach( action: filename -> {
225         System.out.println(filename);
226         GenbankRecord record = null;
227         try { record = Parse( file: filename);
228         } catch (IOException e) { e.printStackTrace(); }
229         List<Gene> genes = record.genes;
230         GenbankRecord finalRecord = record;
231         genes.parallelStream().forEach( action: gene -> {
232             if (Homologous( A: gene.sequence, B: referenceGene.sequence)) {
233                 NucleotideSequence upStreamRegion = GetUpstreamRegion(
234                     dna: finalRecord.nucleotides, gene: gene);
235                 Match prediction = PredictPromoter( upStreamRegion: upStreamRegion);
236                 if (prediction != null) {
237                     consensus.get(referenceGene.name).addMatch( match: prediction);
238                     consensus.get("all").addMatch( match: prediction);
239                 }
240             }
241         }
242     }
243 }
```

Figure 24 Initial approach for parallel stream: using parallelStream() 3 times

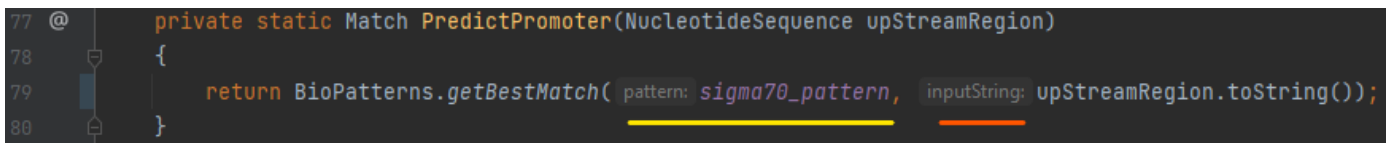
Fixing data race while parallelStream() the app.

While parallelizing the app with parallel stream, my app ran into an IndexOutOfBoundsException 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.



```
Parallel x
Exception in thread "main" java.util.concurrent.ExecutionException: java.lang.IndexOutOfBoundsException: Index 3 out of bounds for length 3
    at qut.Parallel.runExecutorService(Parallel.java:181)
    at qut.Parallel.main(Parallel.java:285)
Caused by: java.lang.IndexOutOfBoundsException: Index 3 out of bounds for length 3 <3 internal lines>
    at java.base/java.util.Objects.checkIndex(Objects.java:359)
    at java.base/java.util.ArrayList.get(ArrayList.java:427)
    at edu.au.jacobi.pattern.Series.get(Series.java:51)
    at edu.au.jacobi.pattern.SeriesAll.match(SeriesAll.java:55)
    at edu.au.jacobi.sequence.Sequence.searchBest(Sequence.java:647)
    at qut.BioPatterns.getBestMatch(BioPatterns.java:11)
    at qut.Parallel.PredictPromoter(Parallel.java:79)
    at qut.Parallel$RunnableTask.run(Parallel.java:203) <5 internal lines>
```

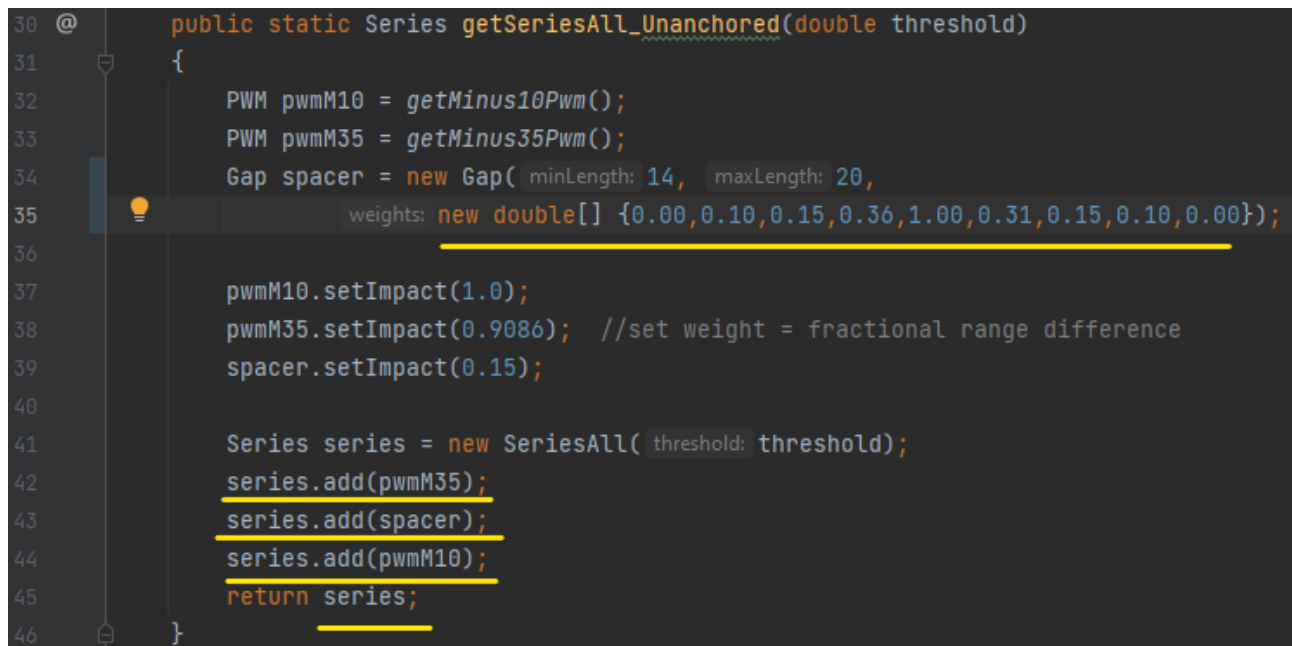
Figure 25 Console screenshot with error



```
77 @ private static Match PredictPromoter(NucleotideSequence upStreamRegion)
78 {
79     return BioPatterns.getBestMatch( pattern: sigma70_pattern, inputString: upStreamRegion.toString());
80 }
```

Figure 26 Identify error cause

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.



```
30 @ public static Series getSeriesAll_Unanchored(double threshold)
31 {
32     PWM pwmM10 = getMinus10Pwm();
33     PWM pwmM35 = getMinus35Pwm();
34     Gap spacer = new Gap( minLength: 14, maxLength: 20,
35     weights: new double[] {0.00,0.10,0.15,0.36,1.00,0.31,0.15,0.10,0.00});
36
37     pwmM10.setImpact(1.0);
38     pwmM35.setImpact(0.9086); //set weight = fractional range difference
39     spacer.setImpact(0.15);
40
41     Series series = new SeriesAll( threshold: threshold);
42     series.add(pwmM35);
43     series.add(spacer);
44     series.add(pwmM10);
45     return series;
46 }
```

Figure 27 A look into getSeriesAll_Unanchored() function

After doing some research, 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 (baeldung, 2021).

6.3 Executor Service

The same issue with sigma70_pattern was encountered in implementing Executor Service and easily resolved.

7. 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. The original program also needs change 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.

8. References

- Baeldung. (2021). *A Guide to the Java ExecutorService*. Baeldung. <https://www.baeldung.com/java-executor-service-tutorial>
- Baeldung. (2021). *An Introduction to ThreadLocal in Java*. Baeldung. <https://www.baeldung.com/java-threadlocal>
- Lee, D., & Hain, P. (2021). *Why a Triplet Code?* Plant & Soil Sciences eLibrary. <https://passel2.unl.edu/view/lesson/3ccee8500ac8/6#:~:text=A%20triplet%20code%20could%20make,of%20all%20%20amino%20acids>.
- Oracle. (2020). *Interface Runnable*. Java Platform Standard Ed.7. <https://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html#:~:text=A%20class%20that%20implements%20Runnable,and%20no%20other%20Thread%20methods>.
- Oracle. (2021). *Interface ExecutorService*. Java SE 11 & JDK 11. <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/ExecutorService.html>
- Strmecki, D. (2021). *When to Use a Parallel Stream in Java*. Baeldung. <https://www.baeldung.com/java-when-to-use-parallel-stream>
- Tabnine. (2021). *How to use parallelStream method in java.util.Collection*. Tabnine. <https://www.tabnine.com/code/java/methods/java.util.Collection/parallelStream>
- The Java Tutorials. (2021). *Parallelism*. Oracle Java Documentation. <https://docs.oracle.com/javase/tutorial/collections/streams/parallelism.html>
- Vogel, L. (2016). *Java concurrency (multi-threading) – Tutorial*. Vogella. <https://www.vogella.com/tutorials/JavaConcurrency/article.html>

9. Appendixes

9.1 Instructions to run app with each technique

Step 1: Open Zip file and open the folder with IntelliJ

Step 2: Open src > qut

Step 3: Now we are in qut directory:

- To use Sequential code, run Sequential.java
- To use Explicit Threading Technique, run ExplicitThreading class in ExplicitThreading.java
- To use Parallel Stream 3rd For loop / Parallel Stream with pre-processing / Executor Service, change 'choice' value with number 1 / 2 / 3 accordingly at line 271 (and threadNum parameter to run on [threadNum] threads) in Parallel.java and run main function.

```
public static void main(String[] args) throws IOException, ExecutionException, InterruptedException {
    long startTime = System.currentTimeMillis();
    int choice = 2;    // Choose base on the case below
    switch (choice) {
        case 1:
            new Parallel().runParallelStream3rd( referenceFile: "referenceGenes.list", dir: "src/Ecoli", threadNum: 8);
        case 2:
            new Parallel().runParallelStreamWithPrep( referenceFile: "referenceGenes.list", dir: "src/Ecoli", threadNum: 8);
        case 3:
            new Parallel().runExecutorService( referenceFile: "referenceGenes.list", dir: "src/Ecoli", threadNum: 8);
        case 4:
            run( referenceFile: "src/referenceGenes.list", dir: "src/Ecoli");
    }
    long durations = System.currentTimeMillis() - startTime;
    System.out.println("Execution time is " + durations/1000 + " s");
}
```

Step 4: To test the result, you similarly change 'choice' value and run the test you want. Tests provided are:

- Compare a Parallel technique's consensus with consensus from analyzing default dataset
- Compare ExplicitThreading technique's consensus with consensus from analyzing default dataset
- Compare a Parallel technique's consensus with Sequential code's
- Compare a Parallel technique's consensus with Sequential code's

9.2 Others

9.2.1 Test script

```
class CompareResult {

    private HashMap<String, String> defaultConsensus;

    @BeforeEach
    void declareDefaultConsensus() {
        defaultConsensus = new HashMap<>();
        defaultConsensus.put("all", " Consensus: -35: T T G A C A gap: 17.6 -10: T A T A A T (5430 matches)");
        defaultConsensus.put("fixB", " Consensus: -35: T T G A C A gap: 17.7 -10: T A T A A T (965 matches)");
        defaultConsensus.put("carA", " Consensus: -35: T T G A C A gap: 17.7 -10: T A T A A T (965 matches)");
    }
}
```

```

A T (1079 matches)");
    defaultConsensus.put("fixA", " Consensus: -35: T T G A C A gap: 17.6 -10: T A T A
A T (896 matches)");
    defaultConsensus.put("caiF", " Consensus: -35: T T C A A A gap: 18.0 -10: T A T A
A T (11 matches)");
    defaultConsensus.put("caiD", " Consensus: -35: T T G A C A gap: 17.6 -10: T A T A
A T (550 matches)");
    defaultConsensus.put("yaaY", " Consensus: -35: T T G T C G gap: 18.0 -10: T A T A
C T (4 matches)");
    defaultConsensus.put("nhaA", " Consensus: -35: T T G A C A gap: 17.6 -10: T A T A
A T (1879 matches)");
    defaultConsensus.put("folA", " Consensus: -35: T T G A C A gap: 17.5 -10: T A T A
A T (46 matches)");
}

/**
 * Compare a Parallel technique's consensus with consensus from analyzing default
dataset
 */
@org.junit.jupiter.api.Test
void parallel_n_defaultResult() throws IOException, ExecutionException,
InterruptedException {
    Parallel.main(new String[0]);
    for (Map.Entry<String, Sigma70Consensus> entry :
Parallel.getConsensus().entrySet()) {
        assertEquals(defaultConsensus.get(entry.getKey()),
entry.getValue().toString());
    }
}

/**
 * Compare a Parallel technique's consensus with consensus from analyzing default
dataset
 */
@org.junit.jupiter.api.Test
void explicit_n_defaultResult() throws InterruptedException {
    ExplicitThreading.main(null);
    for (Map.Entry<String, Sigma70Consensus> entry :
ExplicitThreading.getConsensus().entrySet()) {
        assertEquals(defaultConsensus.get(entry.getKey()),
entry.getValue().toString());
    }
}

/**
 * Compare a Parallel technique's consensus with Sequential code's
 * Note: This test should only be run when input data set has been changed
 * as Sequential takes much time to run
 */
@org.junit.jupiter.api.Test
void parallel_n_sequential() throws IOException, ExecutionException,
InterruptedException {
    Parallel.main(null);
    Sequential.main(null);
    HashMap<String, Sigma70Consensus> sequentialConsensus =
Sequential.getConsensus();
    for (Map.Entry<String, Sigma70Consensus> entry :

```

```

Parallel.getConsensus().entrySet()) {
    assertEquals(sequentialConsensus.get(entry.getKey()).toString(),
entry.getValue().toString());
}

}

/**
 * Compare a Parallel technique's consensus with Sequential code's
 * Note: This test should only be run when input data set has been changed
 * as Sequential takes much time to run
 */
@org.junit.jupiter.api.Test
void explicitThreading_n_sequential() throws IOException, InterruptedException {
    ExplicitThreading.main(null);
    Sequential.main(null);
    HashMap<String, Sigma70Consensus> sequentialConsensus =
Sequential.getConsensus();
    for (Map.Entry<String, Sigma70Consensus> entry :
ExplicitThreading.getConsensus().entrySet()) {
        assertEquals(sequentialConsensus.get(entry.getKey()).toString(),
entry.getValue().toString());    }    }    }

```