Promoter Parallelisation Report

Liam Dale | n9741283

19/10/2020

CAB401 High Performance and Parallel Computing

ABSTRACT – This report details the parallelisation and optimisation of a sequential, computationally intensive bioinformatics program. To garner performance insights an initial sequential profile is taken. The methods with the highest processor views are parallelised in the first two iterations and then finally optimised in the third. The final version can run almost the entire application concurrently across sixteen threads to achieve a speedup of almost twelve. The speedup across a span of threads is graphed and shown to be sublinear, but scalable. A discussion is presented on the results and learning outcomes.

Contents

Introduction

This application aims to predict the sigma70 promoter structure within Ecoli bacteria gene sequences. Promoters are found in the upstream region before a gene, which they control the activation or repression of. They may be dissected into two functional consensus sequences described by the most common nucleotides thirty-five and ten positions upstream [1]. This structure is shown in Figure 1. The exact sequences and their gap is not fixed, so prediction involves pattern matching against expected values. This operation is only performed for the bacteria genes which are relevant. The overall process is computationally intensive and solved sequentially so a better implementation is needed.

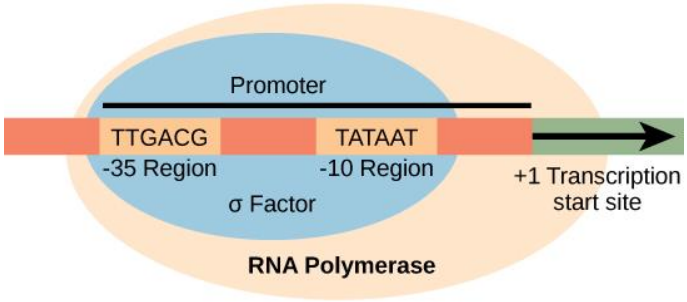

Figure 1 Gene Promoter Structure [1]

This report details the speedup process of the base promoter application. It begins by explaining the sequential application and opportunities for optimisation. In iterative implementations these opportunities are exploited to obtain superior speedup. The details for replication, issues and results for each of these improvements are discussed. In the final section the entire process is reflected on.

Hardware

Analyses and operations documented within this report are performed on a custom computer build. The processor is an AMD Ryzen 3700X, having eight physical and sixteen logical cores at 3.6 GHz base. There are 16GB of DDR4 RAM at 3200MHz. Results given are averaged over five iterations, with a minimal amount of background processes running to curtail system interference.

Sequential Application

Running the application requires a directory with genbank files (./Ecoli/) to describe the Ecoli, and a list of reference genes(./referenceGenes.list) which are of interest. These are parsed to get the genes and related nucleotide information. Iteratively Ecoli genes are filtered by homologous reference and have their upstream region checked for a consensus sequence. This process is displayed in the succeeding diagram. The output is displayed in the table overleaf.

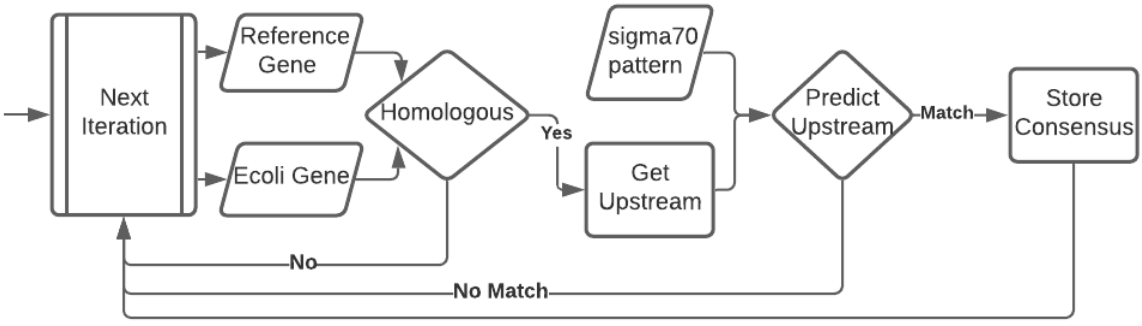

Figure 2 Process Flow

Table 1 Sequential Results

Reference	Consensus			Matches
	Sequence -35	Gap	Sequence -10	
all	T T G A C A	17.6	T A T A A T	5430
fixB	T T G A C A	17.7	T A T A A T	965
carA	T T G A C A	17.7	T A T A A T	1079
fixA	T T G A C A	17.6	T A T A A T	896
caiF	T T C A A A	18.0	T A T A A T	11
caiD	T T G A C A	17.6	T A T A A T	550
yaaY	T T G T C G	18.0	T A T A C T	4
nhaA	T T G A C A	17.6	T A T A A T	1879
folA	T T G A C A	17.5	T A T A A T	46

Sequential Profile

Application profiling is crucial to garner performance insights and inform the optimisation process. The succeeding figure displays an overview of telemetries captured using JProfiler. As expected for a sequential application there is only a single thread used. This greatly stunts performance, limiting the CPU usage to 6.25% for most of the runtime.

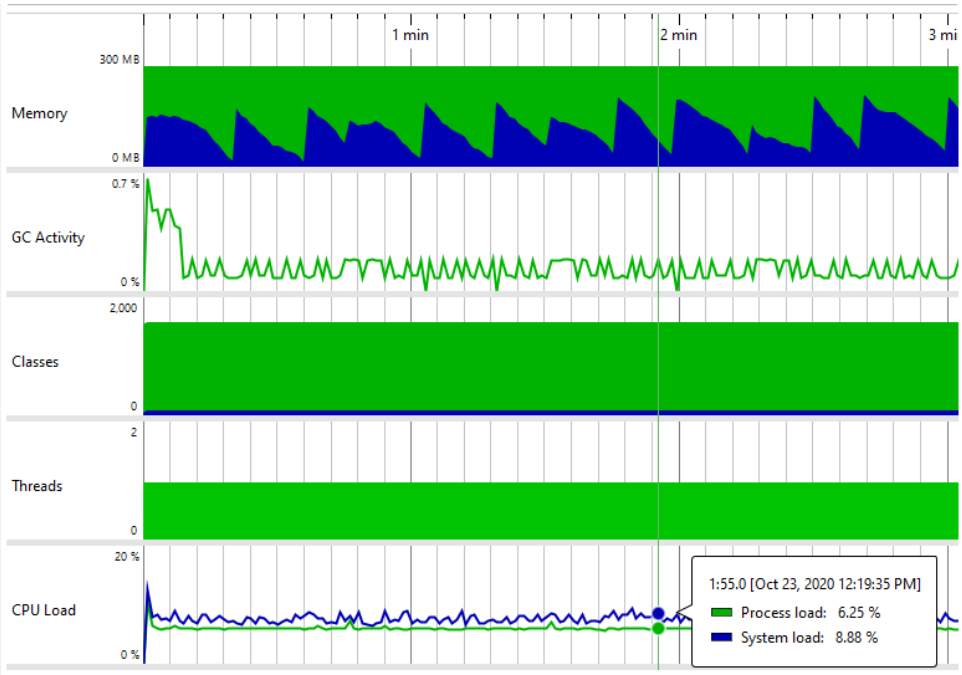

Figure 3 Sequential Profile Overview

The CPU views call tree details that 98% of the CPU time is taken within the Homologous function which checks similarity between genes with the Smith-Waterman algorithm. As such this is where the efforts to optimise should be directed. Failing to improve or parallelise this will inhibit meaningful performance increases. The second most expensive method is PredictPromoter which takes a meagre 1.6% of the CPU runtime.

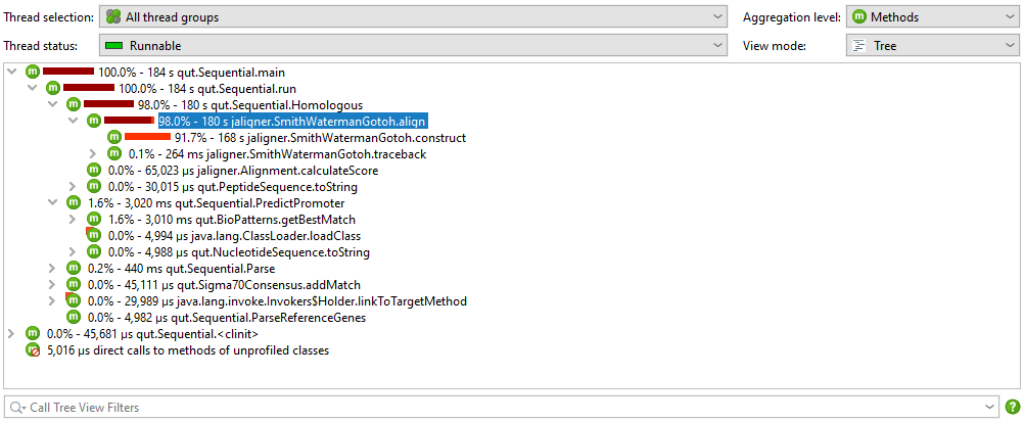

Figure 4 Sequential CPU View Call Tree

The built jar application runs in 177.51 seconds - this will be considered as the fastest sequential application for future purposes as meaningful improvements are unlikely. The Smith-Waterman algorithm is notoriously complex, and has already been optimised. Trying to reduce calls to this by restructuring and having a non-null prediction as a predicate increases runtime. This leaves concurrency as the best avenue to better performance.

Parallelisation Opportunities

All the work is contained within the body of the sequential run method. This is displayed in the following code snippet. The costly checks for homogeneity (line 06) rely solely on the immutable reference (line 04) and Ecoli (line 05) gene sequences, making them independent. As such there is an overt, readily exposed opportunity for parallelisation.

The condition statement body may also be parallelised, although this is not as impactful as suggested by profiling. The upstream region extracting (line 07) again works on fixed data. The succeeding prediction (line 08) and consensus adding (lines 10-11) are a bit harder as they access shared sigma70_pattern and consensus objects.

This is not an exhaustive list of opportunities, there are others such as parallel file parsing not noted. These other changes however are disregarded due to their infinitesimal processor expenditure.

```
01  List<Gene> referenceGenes = ParseReferenceGenes(referenceFile);
    // Loop 1. Iterate over every Ecoli file
02  for (String filename : ListGenbankFiles(dir)){
03     GenbankRecord record = Parse(filename);
       // Loop 2. Iterate over every Bacteria gene
04     for (Gene referenceGene : referenceGenes){
          // Loop 3. Iterate over every Ecoli gene
05        for (Gene : record.genes){
             // Most of the work happens here!
06           if (Homologous(gene.sequence, referenceGene.sequence))
07              NucleotideSequence upStreamRegion =
                      GetUpstreamRegion(record.nucleotides, gene);
08              Match prediction = PredictPromoter(upStreamRegion);
09              if (prediction != null) {
10                 consensus.get(referenceGene.name).addMatch(prediction);
11                 consensus.get("all").addMatch(prediction);
12              }
13           }
14        }
15     }
16  }
```

Genbank Loop
Bacertia Loop
Ecoli Loop

Code Snip 1 Sequential Run

With tens of thousands of iterations the run method provides opportunity for parallelism at many different granularity levels. Any deeper will not be explored as these individually take milliseconds and hence it does not make much sense to split them up. Attempting to parallelise Smith Waterman itself will spawn hundreds of thousands of threads and add plenty of management overhead.

Parallel Application

This section outlines the iterative improvement process. At each stage implementations are verified by comparing sequential and parallel consensus objects in string form. This is a simplistic approach which focusses on the final outcome, not worrying about anything in-between.

Threads and Mapping Computation

The Callable interface represents an asynchronous task, executable concurrently on a separate thread. It is very similar to the age-old Runnable interface, however, may return a result and throw exceptions. Implementation requires overriding the call() method which invokes the task [2]. A generalised implementation for this application is shown in the following code snippet.

```
01  private class GeneThreadFine<T> implements Callable<T> {
02     private Gene gene;
03     private Gene referenceGene;
04     private NucleotideSequence nucleotides;
       // Constructor
05     public GeneThreadFine(Gene gene, Gene referenceGene, NucleotideSequence nucleotides)
06        this.gene = gene;
07        this.referenceGene = referenceGene;
08        this.nucleotides = nucleotides;
09     }
       // Do asynchronous work here!
10     @Override
11     public T call() { ... }
12  }
```

Code Snip 2 General Callable Implementation

The ExecutorService interface provides a high-level way to manage these asynchronous Callable tasks and allocate them among a pool of threads. This avoids the need for manual mapping and load balancing [3]. The relevant methods are described as follows.

1. *Executors.newFixedThreadPool(int nThreads)* creates an ExecutorService object with a pool of nThreads that are reused when needed.

```
01  int maxThreads = Runtime.getRuntime().availableProcessors();
    ...
02  ExecutorService executorService = Executors.newFixedThreadPool(maxThreads);
```

Code Snip 3 Creating an ExecutorService

2. *executorService.invokeAll(Collection<? extends Callable<T>> tasks)* executes tasks concurrently. It synchronises and returns a list of Futures holding statuses and results once all tasks are complete.

```
01  List<Callable<T>> callableList = new ArrayList<>();
02  callableList.add(new T(...));
03  List<Future<T>> resultList = executorService.invokeAll(callableList)
```

Code Snip 4 Running Tasks Ad Hoc via the ExecutorService

3. *executorService.shutdown()* initiates a shutdown without accepting new tasks, allowing a reclamation of its resources.

```
01  executorService.shutdown();
```

Code Snip 5 Shutting Down the ExecutorService

These basic tools are enough for an initial implementation.

Implementation v1: Homologous Parallelisation

The first parallel implementation singularly aims to parallelise the homologous comparison process (Code Snip 6. Line 08). The Callable to do this is simply given the two compared genes and the relevant upstream Ecoli nucleotide sequence. The result and surrounding information is stored within a GenesHomologous object (Appendix B) and returned. This is achieved in the following code snippet.

```
01  private class GeneThreadFine implements Callable<GenePrediction> {
        ...
        // Constructor
02      public GeneThreadFine(Gene gene, Gene referenceGene, NucleotideSequence nucleotides) {
03          this.gene = gene;
04          this.referenceGene = referenceGene;
05          this.nucleotides = nucleotides;
06      }
        @Override
07      public GenePrediction call() {
08          return new GenePrediction(gene, referenceGene,
                    Homologous(gene.sequence, referenceGene.sequence), nucleotides);
09      }
10  }
```

Code Snip 6 v1 Callable

In the run method loops aggregate these into a collection of Callable objects (line 09) before concurrently scheduling them with an ExecutorService object (line 14). Once this finishes the GenesHomologous data storing objects are extracted from returned Futures and used to support a sequential prediction process as before.

```
01  public void Run(Integer threads) throws IOException {
        // Scheduling setup
02      ExecutorService executorService = Executors.newFixedThreadPool(threads);
03      List<Callable<GenesHomologous>> callableList = new ArrayList<>();
        // Iteratively aggregate callables
04      List<Gene> referenceGenes = ParseReferenceGenes(referenceFile);
05      for (String filename : ListGenbankFiles(dir)) {
06          GenbankRecord record = Parse(filename);
07          for (Gene referenceGene : referenceGenes) {
08              for (Gene gene : record.genes) {
09                  callableList.add(new GeneThreadFine(gene, referenceGene, record.nucleotides));
10              }
11          }
12      }
        // Run all scheduled tasks
13      try {
14          List<Future<GenesHomologous>> resultList = executorService.invokeAll(callableList);
15          for (int i = 0; i < resultList.size(); i++) {
16              Future<GenesHomologous> future = resultList.get(i);
17              GenesHomologous result = future.get();
18              if (result.homologous) {
                    // As before
19                  NucleotideSequence upStreamRegion = GetUpstreamRegion(
                        result.nucleotides, result.gene);
20                  Match prediction = PredictPromoter(upStreamRegion);
21                  if (prediction != null) {
22                      consensus.get(result.referenceGene.name).addMatch(prediction);
23                      consensus.get("all").addMatch(prediction);
24                  }
25              }
26          }
27          executorService.shutdown();
28      } catch (InterruptedException| ExecutionException ex) {
29          ex.printStackTrace();
30      }
31  }
```

Code Snip 7 v1 Run Method

Implementation v1 Results

A profile shows that the initial parallelisation has worked very well. For almost the entire duration the CPU is running all threads concurrently, keeping it at full load. There is an initial period for file parsing and setup, and a closing period where the promoter predictions are being made which only operate on a single thread. While more threads are added these period remains fixed, which will make the speedup less linear. As the dataset grows speedup will worsen as this period ratiometrically lengthens. These reasons affect scalability.

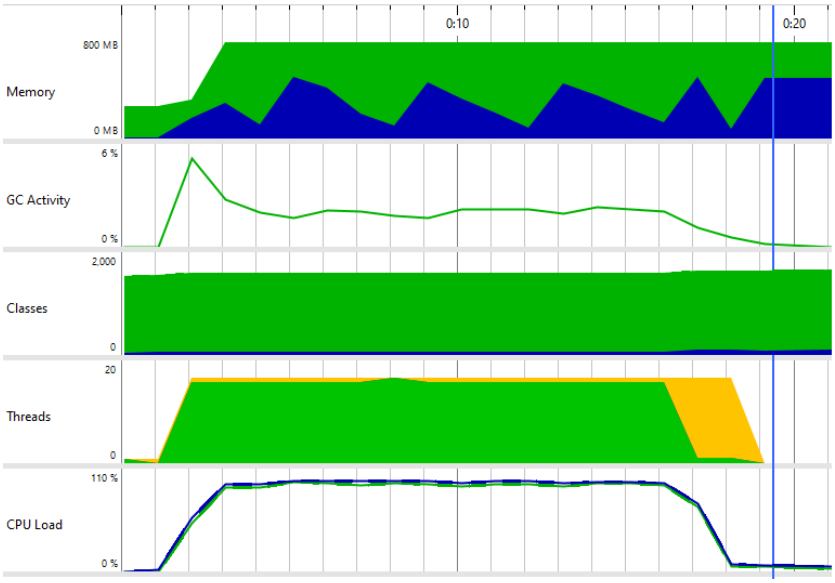

Figure 5 Implementation v1 Application Profile

There are strong wall-time results, as displayed in the following chart. A slowdown at a single thread reflects parallelisation overheads. Speedup begins to regress around eight threads. This is probably due to the scalability limitations mentioned and the start of core hyperthreading. Another variable is Ryzen Precision Boost automatic overclocking, which will raise frequency at low threads when the processor has active headroom in temperature, workload and power consumption [4].

Threads	Time (s)	Speedup
1	179.537	0.9887
2	91.6226	1.9374
4	48.6122	3.6516
6	33.8796	5.2395
8	27.2256	6.52
10	23.0206	7.711
12	20.137	8.8152
14	18.161	9.7743
16	16.9144	10.4947

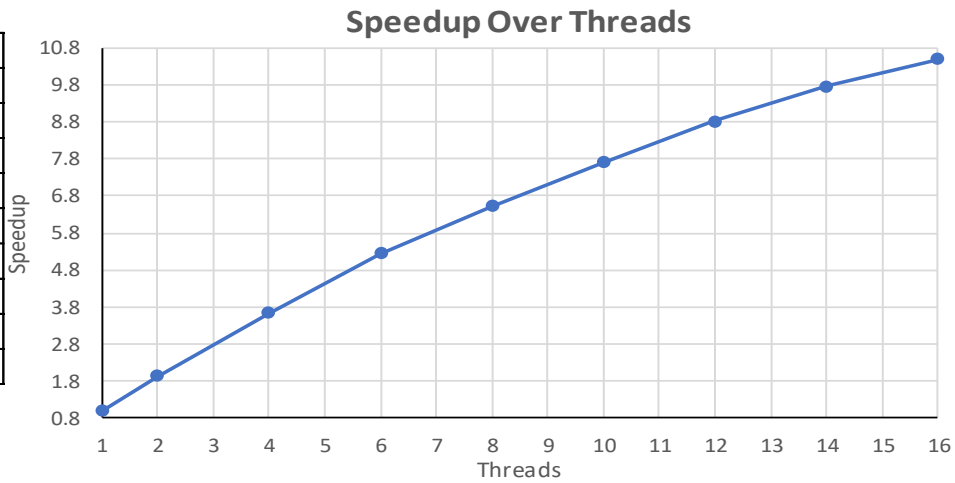

Figure 6 Implementation v1 Speedup Results

To improve the speedup and make it more scalable, the prediction process must be parallelised in succeeding implementations.

Implementation v2: Full Parallelisation

This implementation also aims to also parallelise the prediction process, however this is difficult as the sigma70_pattern and consensus objects are concurrently acted on by different threads. To overcome this ThreadLocal and Lock concepts are introduced.

When calling PredictPromoter the sigma70_pattern object is iteratively compared to the nucleotide sequence at different starting positions to get the best match. The IPattern interface this implements specifies that this is not thread safe, as the match method overrides the contents of the previous call. For the pattern to be used in a different thread, it must be cloned. The ThreadLocal class enables this by providing a way to create variables which may only be read and written by the same thread. If several threads are running the same code, referencing the same ThreadLocal variable they cannot see the other's ThreadLocal variable [5]. The following code snippet shows class attribute declaration (line 01) variable retrieval (line 04).

```
01  private ThreadLocal<Series> sigma70_pattern = ThreadLocal.withInitial(()
            -> Sigma70Definition.getSeriesAll_Unanchored(0.7));
02  ...
03  private Match PredictPromoter(NucleotideSequence upStreamRegion) {
04      return BioPatterns.getBestMatch(
            sigma70_pattern.get(), upStreamRegion.toString());
05  }
```

Code Snip 8 ThreadLocal Usage

Predictions made are added onto Sigma70Consensus objects keyed from a HashMap. This non-atomic process must be fully completed before beginning again for data integrity; the relevant code must be mutually exclusive. For this purpose a ReentrantLock is chosen, which is an explicit mutex lock with methods for fine grained control [6]. This is declared as an immutable class attribute (line 01), locked (line 03) before the unsafe code (lines 05, 06) and unlocked (line 09) after. There will be overhead associated with this and threads will lock each other out even if they are editing different hash objects. A try-finally block is used to ensure the lock is released despite any exceptions.

```
01  private final ReentrantLock lock = new ReentrantLock();
    ...
02  if (prediction != null) {
03      lock.lock();
04      try {
05          consensus.get(referenceGene.name).addMatch(prediction);
06          consensus.get("all").addMatch(prediction);
07      }
08      finally {
09          lock.unlock();
10      }
11  }
```

Code Snip 9 Lock Usage

With these concepts the callable is augmented to perform the entire prediction process. There is no need to return a result and hence the GenesHomologous class is no longer used.

```
01  private class GeneThreadFine implements Callable<Void> {
        ...
02      // Constructor
03      public GeneThreadFine(Gene gene, Gene referenceGene, NucleotideSequence nucleotides) {
04          this.gene = gene;
            this.referenceGene = referenceGene;
05          this.nucleotides = nucleotides;
06      }
07      @Override
08      public Void call() {
09          // Copy paste of inner loop, with locking for shared heap
10          if (Homologous(gene.sequence, referenceGene.sequence)) {
11              Match prediction = PredictPromoter(GetUpstreamRegion(nucleotides, gene));
                if (prediction != null) {
12                  lock.lock();
13                  try {
14                      consensus.get(referenceGene.name).addMatch(prediction);
15                      consensus.get("all").addMatch(prediction);
16                  }
17                  finally {
18                      lock.unlock();
19                  }
20              }
21          }
22          return null;
23      }
24  }
```

Code Snip 10 v2 Callable

This also simplifies the run method, as there is no need to act on Futures objects from the ExecutorService.

```
01  public void Run(Integer threads) throws IOException {
        // Scheduling setup
02      ExecutorService executorService = Executors.newFixedThreadPool(threads);
03      List<Callable<Void>> callableList = new ArrayList<>();
        // Iteratively aggregate callables
04      List<Gene> referenceGenes = ParseReferenceGenes(referenceFile);
05      for (String filename : ListGenbankFiles(dir)) {
06          GenbankRecord record = Parse(filename);
07          for (Gene referenceGene : referenceGenes) {
08              for (Gene gene : record.genes) {
09                  callableList.add(new GeneThreadFine(gene, referenceGene, record.nucleotides));
10              }
11          }
12      }
        // Run all scheduled tasks
13      try {
14          executorService.invokeAll(callableList);
15      } catch (InterruptedException ex) {
16          ex.printStackTrace();
17      }
18      executorService.shutdown();
19  }
```

Code Snip 11 v2 Run Method

Implementation v2 Results

A profile shows that this second implementation has worked very well. There is no longer a sequential prediction process at closing, it instead occurs concurrently throughout execution. There is still an initial period where setup is performed. This is displayed in the succeeding figure.

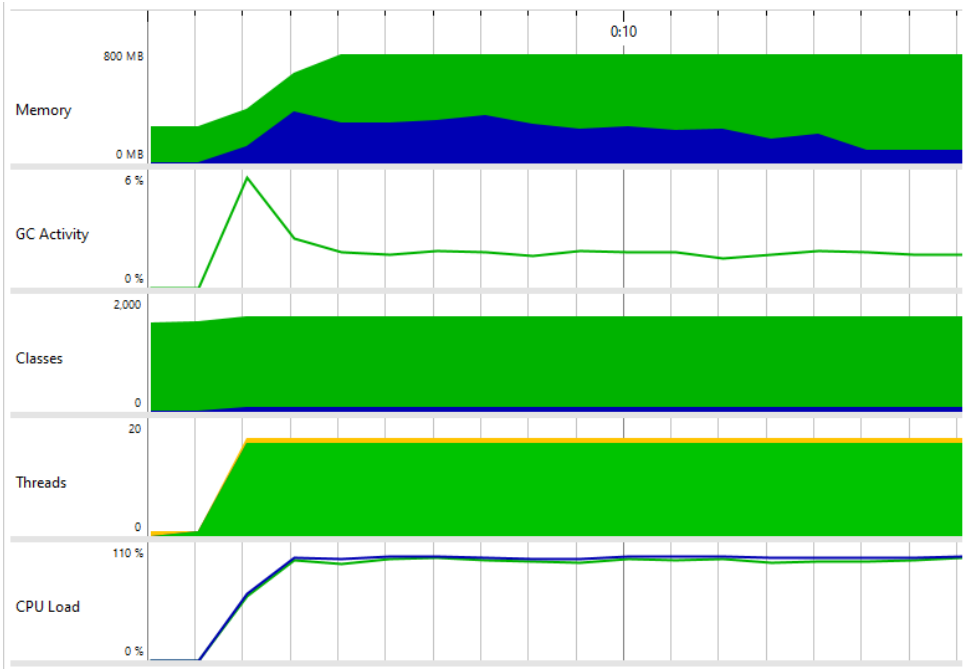

Figure 7 Implementation v2 Application Profile

The speedup at max threads is now 11.61, which is up from 10.7% from the previous implementation. Scalability has improved; the regression begins later and is not as sharp as speedup improvements scale with thread count.

Threads	Time (s)	Speedup
1	179.204	0.9906
2	90.109	1.97
4	47.0904	3.7696
6	32.233	5.5071
8	25.6328	6.9252
10	21.6588	8.1958
12	18.5566	9.5659
14	16.6108	10.6865
16	15.2894	11.6101

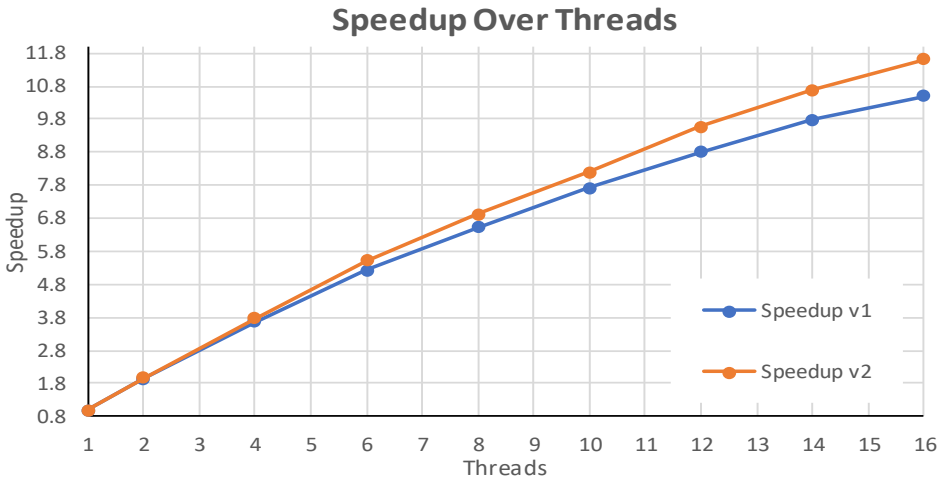

Figure 8 Implementation v2 Speedup Results

Now that the majority of work is done in parallel, optimisations may be made.

Implementation v3: Parallelisation Optimisations

One possible optimisation is to make the parallelism less coarse, and hence reduce thread management overhead. This will also enforce variable reuse and so may also enable caching to reduce memory stalls. The trade-off is that load balancing will become more difficult and less precise. This change will involve moving loops into the callable object to be performed concurrently. The Genbank loop is a possibility but is an unattractive option as the Ecoli loop depends upon it and extra infrastructure is required to avoid repeated file parsing. The other Ecoli and Reference loops

have no data flow dependencies so either one or a combination of both is possible. Options are outlined in the following table with coarseness in ascending order. The speedup at maximum threads is detailed in the final column.

Table 2 Granularity Options

#	Parallel Loops	Sequential Loops	Net Callables Required	Max Threads Speedup
1	-	Genbank, Ecoli, Reference (v2)	$number\ ref\ genes \times number\ total\ Ecoli\ genes$	11.610
2	Reference	Genbank, Ecoli	$number\ total\ Ecoli\ genes$	11.705
3	Ecoli	Genbank, Reference	$number\ Ecoli\ files \times number\ refence\ genes$	10.251
4	Ecoli, Reference	Genbank	$number\ Ecoli\ files$	3.828

This ordering is set as there are only eight reference genes, whereas the Ecoli genbank files may contain thousands of genes. This order of magnitude difference means that options one and two are largely unaffected by load balancing, however options three and four struggle. This is reflected by the maximum speedup where options one and two have similar results and options three and four are very negatively affected. Option two has a very slight edge so will be used; this does not consistently hold at a low amount of cores where load balancing is more difficult.

Another optimisation is removing the ReentrantLock in favour of a ConcurrentHashMap to reduce associated overhead. This acts like a HashMap however supports full concurrency for non-blocking retrievals and high concurrency for updates. These updates can be made atomic to preserve consensus integrity by using the compute() remapping method [7]. The intent behind this change is that updates to different keys do not lock each other out. At maximum threads this improves the implementation v2 speedup to 11.752 which is an underwhelming increase of 1.2%. This is not a surprise as this affects a very small portion of CPU time.

With these optimisations combined the final implementation callable appears in Code Snip 12 as follows.

```
01  public ConcurrentHashMap<String, Sigma70Consensus> consensus = new ConcurrentHashMap<>();
    ...
02  private class GeneThreadFine implements Callable<Void> {
03      private Gene gene;
04      private List<Gene> referenceGenes;
05      private NucleotideSequence nucleotides;
06      public GeneThreadFine(Gene gene, List<Gene> referenceGenes, NucleotideSequence nucleotides) {
07          this.gene = gene;
08          this.referenceGenes = referenceGenes;
09          this.nucleotides = nucleotides;
10      }
        @Override
11      public Void call() {
12          for (Gene referenceGene : referenceGenes) {
13              if (Homologous(gene.sequence, referenceGene.sequence)) {
14                  Match prediction = PredictPromoter(GetUpstreamRegion(nucleotides, gene));
15                  if (prediction != null) {
16                      consensus.compute(referenceGene.name,(k,v)->{v.addMatch(prediction); return v;});
17                      consensus.compute("all",(k,v)->{v.addMatch(prediction); return v;});
18                  }
19              }
20          }
21          return null;
22      }
23  }
```

Code Snip 12 Implementation v3 callable

Callables are now aggregated through less loops.

```
...
// For each Ecoli file
for (String filename : ListGenbankFiles(dir)) {
    GenbankRecord record = Parse(filename);
    // For each gene in the Ecoli file
    for (Gene gene : record.genes) {
        callableList.add(new GeneThreadFine(gene, referenceGenes, record.nucleotides));
    }
}
...
```

Code Snip 13 Implementation v3 run body

Implementation v3 Results

The final speedups are displayed in the succeeding figure. These results are almost the exact same as the previous implementation's, providing a very small and almost unnoticeable edge at each level of threading.

Threads	Time (s)	Speedup
1	178.294	0.9956
2	90.033	1.9716
4	47.591	3.7299
6	31.807	5.5809
8	25.423	6.9823
10	21.026	8.4425
12	18.53	9.5797
14	16.537	10.7342
16	14.847	11.956

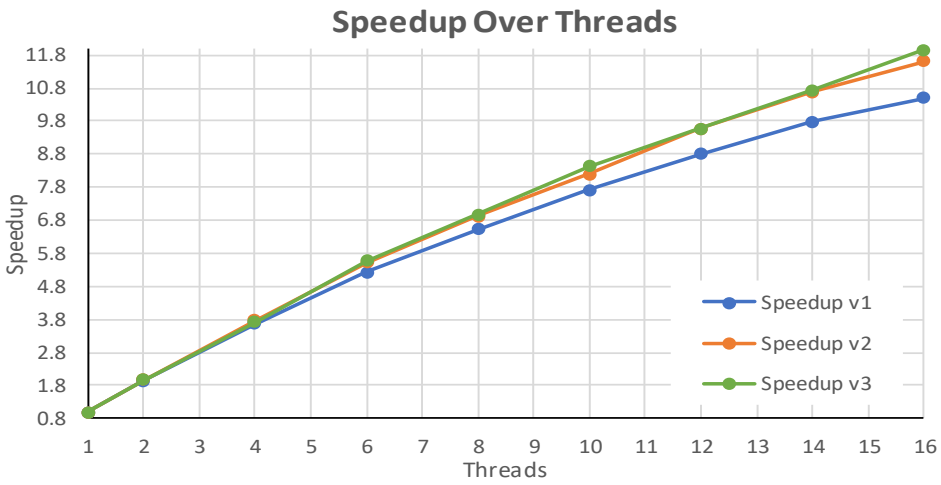

Figure 9 Implementation v3 Speedup Results

Inspecting the CPU view tree for this implementation shows that a total of 99.7% of CPU time is spent in calls to concurrent methods. This shows that the parallelisation process is complete, however does not provide any guarantees on optimisation completeness.

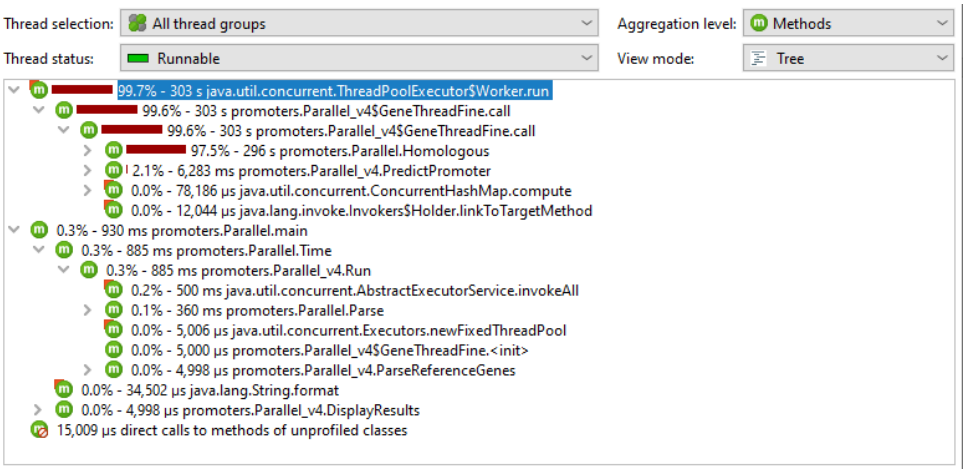

Figure 10 Implementation v3 CPU Views

Discussion

This parallelisation process has been a resounding success – not only the final product but also the learning outcomes. It has been my first exposure to non-trivial parallel programming, and I believe that I have a taken away strong understandings of the parallelisation process, struggles and key concepts.

Figure 11 shows the maximum achievable speedup, which is given by Ahmdal's law as follows [8].

$$1/((1-P)+P/S)$$

The parallelisable fraction is denoted P, and taken at 99.6% as given by the sum of Homologous and PredictPromoter CPU time in the sequential profile. Speedup of the parallel section S is given perfectly as the number of threads. The actual outcome does not meet the maximum given by Ahmdal; I believe the attempt has been successful though as perfect linear speedup is an unrealistic expectation due to aforementioned reasons. Linearity is mostly achieved until eight threads, where sharing of system resources, hyperthreading and Ryzen precision boost drop-off begin to have exacerbated effects. The speedup realised is still scalable, as it does not hit a plateau. With a larger dataset (copy-pasted Ecoli folders) results are analogous which provides some validation of the outcome. It would be interesting to see scaling at a higher amount of threads.

Threads	S	P	Ahmdal
1	1	0.996	1
2	2	0.996	1.992032
4	4	0.996	3.952569
6	6	0.996	5.882353
8	8	0.996	7.782101
10	10	0.996	9.65251
12	12	0.996	11.49425
14	14	0.996	13.30798
16	16	0.996	15.09434

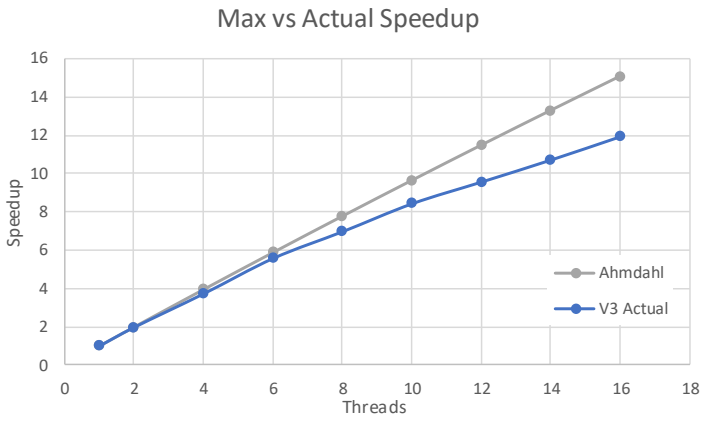

Figure 11 Ahmdals Expectation

The importance of application profiling to direct parallelisation efforts was a key takeaway from this project. The first parallel implementation had a transformative effect on the application runtime by isolating the major CPU expense seen in the Homologous function call. This direct focus made the implementation very simple, almost bug free and trivial to implement. It also meant that I did not have to understand how every single element of application worked to make this change – which is especially nice when looking at another programmer's code. After exhausting this major profiling hint implementations v2 and v3 were not as impactful.

Another learning is that the optimisation process will never be complete, and an end goal needs to be set to avoid prolonged development efforts with diminishing returns. Parallel iterations brought in more elements and programming concepts however did not replicate the initial success. Being more complex they were more difficult to implement and took more time to develop and test. There were also some failed improvements which are not documented in this report. Understanding the application goals is important; if this program would be run on CPUs with a similar amount of cores and the dataset size would not grow the very first parallel implementation would most likely be sufficient.

Throughout this process the memory efficiency was largely ignored. In the profiled telemetries up to 800MB would be used by the Java runtime and so undoubtedly there were many processor stalls attributed to memory latency. While it is hard to efficiently manage such large amounts of memory this is an area I would like to optimise and plan for in my next parallelisation project.

Although I have had strong learning outcomes to carry into the future, I wish I had extended myself further. It would have been interesting to use a distributed computing model or work with the QUT supercomputer. I had hopes to parallelise on the GPU at the start of the semester, however due to troubles finding a suitable application and running low on time this fell through. In future I look forwards to trying these more advanced concepts.

References

[1] Lumen Learning, “Prokaryotic Transcription,” 2020. [Online]. Available: https://courses.lumenlearning.com/wm-biology1/chapter/prokaryotic-transcription/. [Accessed 23 10 2020].

[2] J. Jenkov, “Java Callable,” Jenkov Tutorials, 27 10 2018. [Online]. Available: http://tutorials.jenkov.com/java-util-concurrent/java-callable.html. [Accessed 2020 10 23].

[3] J. Jenkov, “Java ExecutorService,” Jenkov Tutorials, 28 10 2018. [Online]. Available: http://tutorials.jenkov.com/java-util-concurrent/executorservice.html. [Accessed 23 10 2020].

[4] AMD, “AMD Ryzen Technology: Precision Boost 2 Performance Enhancement,” 2020. [Online]. Available: https://www.amd.com/en/support/kb/faq/cpu-pb2. [Accessed 23 10 2020].

[5] J. Jenkov, “Java ThreadLocal,” Jenkov Tutorials, 28 09 2020. [Online]. Available: http://tutorials.jenkov.com/java-concurrency/threadlocal.html. [Accessed 23 10 2020].

[6] B. Winterberg, “Java 8 Concurrency Tutorial: Synchronization and Locks,” Winterbe, 2020 10 23. [Online]. Available: https://winterbe.com/posts/2015/04/30/java8-concurrency-tutorial-synchronized-locks-examples/. [Accessed 4 30 2015].

[7] T. Tapper, “Java 8: ConcurrentHashMap Atomic Updates,” DZone, 19 4 18. [Online]. Available: https://dzone.com/articles/java-8-concurrenthashmap-atomic-updates. [Accessed 23 10 2020].

[8] a. Jenkov, “Amdahl's Law,” Jenkov Tutorials, 26 6 2015. [Online]. Available: http://tutorials.jenkov.com/java-concurrency/amdahls-law.html. [Accessed 23 10 2020].

Appendix

Appendix A: Running and Compiling

Builds (Java version 14.0.2) are available in the OutputRunner and OutputParallel directories as jar files. The first runs the sequential and parallel implementations, the second only runs parallel implementations. After navigating to the relevant directory run the jar file as follows.

java -jar promoter.jar

There are args which may be passed into this jar file.

Arg	Description	Usage
+vn	Run specific parallelisation version n, between one and three (will run all if none are specified)	+v1
-i n	Run for n iterations, and average the result (default 1 iteration)	-i 5
-t n	Wait for n seconds in between the iterations to allow the processor to wind down and reset (default 0 seconds)	-t 60
-all	Run across the span of all threads (will just run max threads if not specified)	-all

Example: run the third implementation, averaging results over five iterations and waiting sixty seconds in-between

java -jar promoter.jar -i 5 -t 60 +v3

The jar files are currently built with IntelliJ. There is no build script supplied. If a recompilation is necessary, perform the following steps.

1. Navigate to the root directory
2. javac -classpath lib/jacobi.jar src/jaligner/matrix/*.java src/jaligner/util/*.java src/jaligner/*.java src/qut/*.java src/promoters/*.java
3. Move referencelist and Ecoli directory one level above the root directory
4. java -classpath "src;lib/*" promoters.Parallel *args **OR**
   java -classpath "src;lib/*" promoters.Runner *args

Appendix B: GenesHomologous Class

Class used to return structured data from callable in first implementation.

```
01  public class GenesHomologous {
02      public Gene gene;
03      public Gene referenceGene;
04      public boolean homologous;
05      private NucleotideSequence nucleotides;
        //Constructor
06      public GenesHomologous(Gene gene, Gene referenceGene,
07              boolean homologous, NucleotideSequence nucleotides) {
08          this.gene = gene;
09          this.referenceGene = referenceGene;
10          this.homologous = homologous;
11          this.nucleotides = nucleotides;
12      }
13  }
```

Appendix C: Full Results

SEQUENTIAL							
Threads	IT1	IT2	IT3	IT4	IT5	AVG	
Seq	177.06	178.272	178.03	176.166	178.028	177.5112	
VERSION1							
Threads	IT1	IT2	IT3	IT4	IT5	AVG	SPEEDUP
1	179.538	179.819	178.8	180.379	179.148	179.5368	0.988718
2	90.77	91.102	92.331	92.054	91.856	91.6226	1.937417
4	48.869	48.393	48.701	48.51	48.588	48.6122	3.651577
6	33.996	33.851	33.937	33.9	33.714	33.8796	5.239472
8	27.319	27.279	27.346	27.176	27.008	27.2256	6.520011
10	23.895	22.95	22.967	22.675	22.616	23.0206	7.710972
12	20.059	19.842	19.867	20.622	20.295	20.137	8.815176
14	18.134	18.171	18.059	18.265	18.176	18.161	9.774308
16	16.64	16.694	17.191	17.283	16.764	16.9144	10.49468
VERSION2							
Threads	IT1	IT2	IT3	IT4	IT5	AVG	SPEEDUP
1	179.268	178.943	179.234	179.451	179.122	179.2036	0.990556
2	90.247	90.476	90.129	89.422	90.271	90.109	1.969961
4	47.037	47.254	47.118	46.906	47.137	47.0904	3.769584
6	32.108	32.109	32.376	32.358	32.214	32.233	5.507126
8	25.833	25.428	25.524	25.329	26.05	25.6328	6.925158
10	21.906	21.726	21.751	21.51	21.401	21.6588	8.1958
12	18.418	18.419	18.77	18.631	18.545	18.5566	9.565933
14	16.951	16.867	16.812	16.249	16.175	16.6108	10.68649
16	15.5	15.588	15.087	15.138	15.134	15.2894	11.61008
VERSION3							
Threads	IT1	IT2	IT3	IT4	IT5	AVG	SPEEDUP
1	176.505	178.619	179.38	178.644	178.324	178.2944	0.995607
2	89.996	89.257	90.777	90.395	89.738	90.0326	1.971632
4	46.862	48.007	47.684	47.878	47.523	47.5908	3.729948
6	32.215	31.931	31.75	31.58	31.559	31.807	5.580885
8	25.646	25.159	25.153	25.08	26.079	25.4234	6.982198
10	21.312	21.04	20.992	20.926	20.861	21.0262	8.442381
12	18.822	18.386	18.445	18.378	18.62	18.5302	9.579562
14	16.585	16.533	16.508	16.309	16.748	16.5366	10.73444
16	14.774	14.922	14.839	14.903	14.798	14.8472	11.95587