PARALLISATION PROJECT

CONSENSUS PROMOTER PREDICTOR

Liam Percy | N9959807 | CAB401 | 18/9/2020

This report outlines the process behind parallelising a consensus promoter sequence prediction program written in Java, using the Java Executor Service framework.

Contents

Introduction

Overview

The consensus promoter predictor is an application written in the Java programming language. The program compares a series of reference genes with several bacteria genes in order to find similarities between these gene sequences, and further, develop a consensus for promoters within the bacteria. A promoter is a binding site that controls when a protein will be produced (transcription), and the consensus sequence represents the ideal promoter in the bacteria's DNA

MCRHSLRSDGAGFYQLAGCEYSFSAIKIAAGGQFLPVICAMAMKSHFFLISVLNRRLTLTAVQGILGRFSLF

Figure 1: Example Gene Sequence

Similarity is measured using the Smith-Waterman algorithm, a popular bio-informatics algorithm which performs sequence alignment on the two genes and outputs a similarity score. If the genes meet the required similarity threshold, the application find that particular gene in the bacteria's DNA sequence and extract the upstream region.

The application will then attempt to find whether a promoter exists within the upstream region using pattern matching algorithms. Pattern matching occurs at 10 base pairs upstream from the gene (-10), and another which is 35 base pairs up from the gene (-35). If a promoter exists, then application will add that entry to the consensus sequence, and once complete, return said consensus sequencies for all reference genes.

Gene	-35	Gap	-10	Matches
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

Figure 2: Example Output

Process Flow

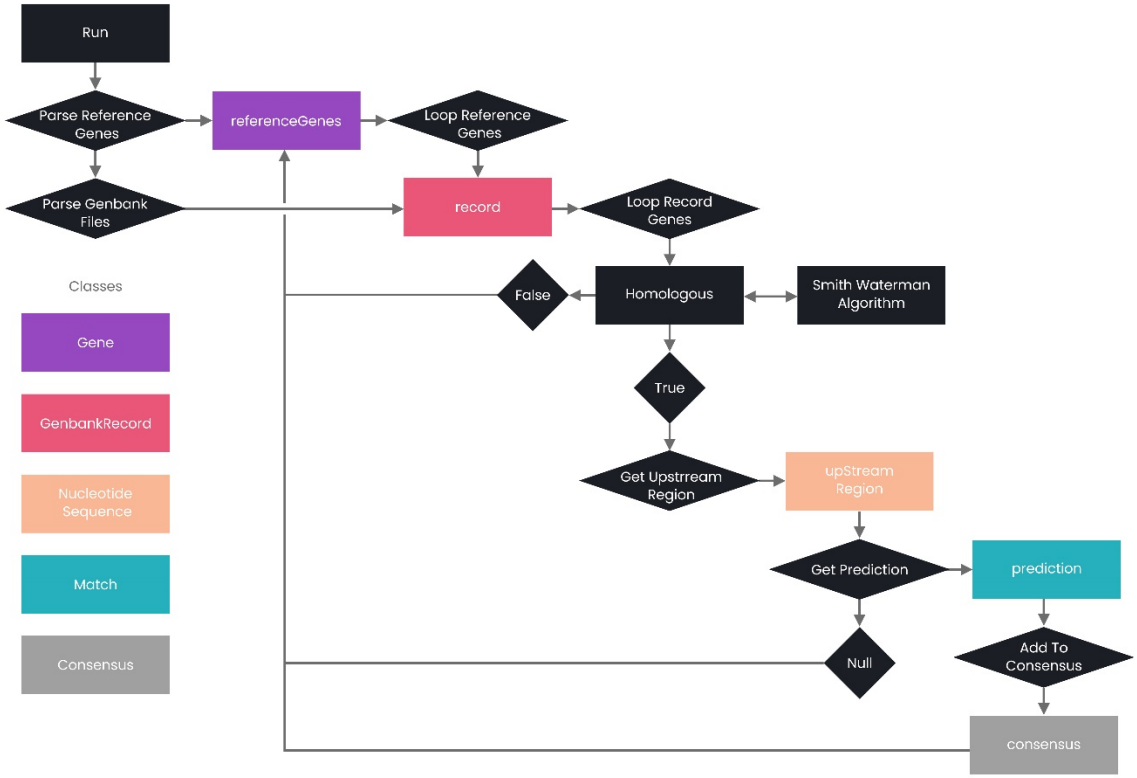

Figure 3: Process Flow Diagram

Summary

1. Parse reference genes and store as a list of *Gene* objects.
2. Iterate through Genbank records, parse and stored as *GenbankRecord* objects.
3. For each Genbank record, iterate over a reference gene, and within this loop, iterate over a gene in the genbank record.
4. Use Smith-Waterman algorithm to determine if the reference gene and the genbank gene are homologous (similar)
5. If they are, get the upstream region of the gene and store it as a *NecleotideSequence* object.
6. Use the nucleotide sequence to predict whether a promoter exits in the upstream region or not using the *BioPatterns* class.
7. If the promoter does exist in the region, add it to the consensus hash map
8. Repeat until loop is finished, then output the results.

Analysis

Identifying Constructs

There was one key area within the application that was identified as a potential areas for parallelism, namely the main 3-level nested loop responsible for comparing reference genes to bacteria genes and adding them to the consensus if specific criteria is met.

We can break apart each of these inner loops and analyse their execution time and level of granularity individually.

Section	Execution Time (s)	Granularity
Level 1: Genbank record	~44	Course
Level 2: Reference Gene	~2 - 8	Medium
Level 3: Gene comparison	~0.002	Fine

Figure 4: Execution Time for Nested Loops

As we can observe from the figure above, the first level of the nested loop takes up a significant portion of time and has particularly course grained parallelism. This indicates that, even if we were to parallelise this section only, we would still be hindered significantly by the inner loop sections.

The second level of the nested loop takes much less time compared to the first and has more fine-grained parallelism. With that said, there are only eight reference gene loops to parallelise per Genbank record, while the final level in the nested loop iterates over thousands of entries from the Genbank records. Each one of these operations is incredibly quick, meaning that the third level of the nested loop is a much more scalable option for parallelisation.

Identifying Dependencies

Initially the prospect of parallelising multiple levels within the loop was considered, however upon further investigation, a loop carried dependency was found within the third level of the nested for loop, at the point in which matches are added to the hash map. Multiple threads may read the current value of the HashMap at the same time, and hence an old value of the consensus is being used to create a new value. This can introduce race conditions into the application, which will prevent it from running correctly.

To ensure the parallelisation of the program is successful, synchronisation techniques will have to be employed to ensure that when a thread enters this critical section, no other thread can read or write to the consensus HashMap.

Implementation

Abstractions

As previously mentioned, consensus promoter predictor was written in Java. Java provides several interfaces, libraries and frameworks that assist in parallelising a program. To being the parallelisation process, Java's Executor Framework was used to create a thread pool. With this, we can create a list of callable items, using a Java's callable interface, and invoke the execute all function from our executor service.

Finally, to synchronise the parts of the program running in critical sections, a Java's standard Re-entrant lock was used. This allows us the lock down a portion of our threads responsible for writing data to the consensus HashMap, and unlock this section once the critical section is complete. In doing so, will prevent against race conditions, and ensure the program continues to run successfully.

Mapping Computation/Data

To parallelise the application, the inner-most section in the nested loop was converted into a list of callable actions.

```
    public static void run(String referenceFile, String dir) {

        List<Callable<Void>> callableList = new ArrayList<>();

        List<Gene> referenceGenes = ParseReferenceGenes(referenceFile);

        for (String filename : ListGenbankFiles(dir)) {
            System.out.println(filename);
            GenbankRecord record = Parse(filename);
            for (Gene referenceGene : referenceGenes) {
                for (Gene gene : record.genes) {
                    // Add tasks from the inner for-loop to the callable list
                    callableList.add(new GeneTask(gene, referenceGene, record));
                }
            }
        }
}
```

A Gene task then executes the exact same sequence of code as the sequential version, however in this particular case, in order to prevent against loop decencies, we have specified our re-entrant lock.

```
        private static final ReentrantLock addLock = new ReentrantLock();
        ...
        // Within the sequential if consensus function
                addLock.lock();
                consensus.get(referenceGene.name).addMatch(prediction);
                consensus.get("all").addMatch(prediction);
                addLock.unlock();
```

Because we moved the execution of each item into its own thread, we also need to consider the state fullness of any data within the functions that appear in this loop. More specifically, the *PredictPromoter* function, which returns the probability of a promoter being within the upstream region of gene, uses the global variable *sigma70_pattern*, to find a best match. The value of resulting match will depend on the state of the thread of execution, and thus we can define the variable as a ThreadLocal object.

```
private static ThreadLocal<Series> sigma70_pattern = ThreadLocal.withInitial(
        () -> Sigma70Definition.getSeriesAll_Unanchored(0.7));
```

Finally, we are able to call the executor service in order to begin executing these threads.

```
// Create a new executor service to handle assignment of tasks to threads
private static ExecutorService executorService = Executors.newFixedThreadPool(24);

try {
        executorService.invokeAll(callableList);
}       catch (InterruptedException e) {
        e.printStackTrace();
}
```

Results

Performance of Loops

The decision was made to parallelise the innermost section of the nested for loop, analysis was also completed on the second level loop. In doing so, we were able to compare the performance between the two techniques and select the most effective solution. The execution time for all of these methods can be seen below.

Technique	Innermost Loop	Second Level Loop
Cores Used		
1	179.417s	178.8s
2	90.44s	90.36s
4	46.43s	51.46s
8	25.01s	25.89s
12	17.6s	19.01s
16	14.61s	16.98s
24	10.51s	13.87s

Figure 5: Execution Time for Parallelisation of Loops

As we can observe, the innermost loop performs slightly better as we increase the number of cores available, largely due to the finer granularity of the tasks within the loop.

Comparison with Sequential Program

We can compare the execution time for the parallelised program with that of the original sequential program and calculate the overall speedup of the program in its parallelised state. When timing the sequential program, it was found that it takes approximately **177.9 seconds** to complete execution. From this, our speedup graph takes the form of:

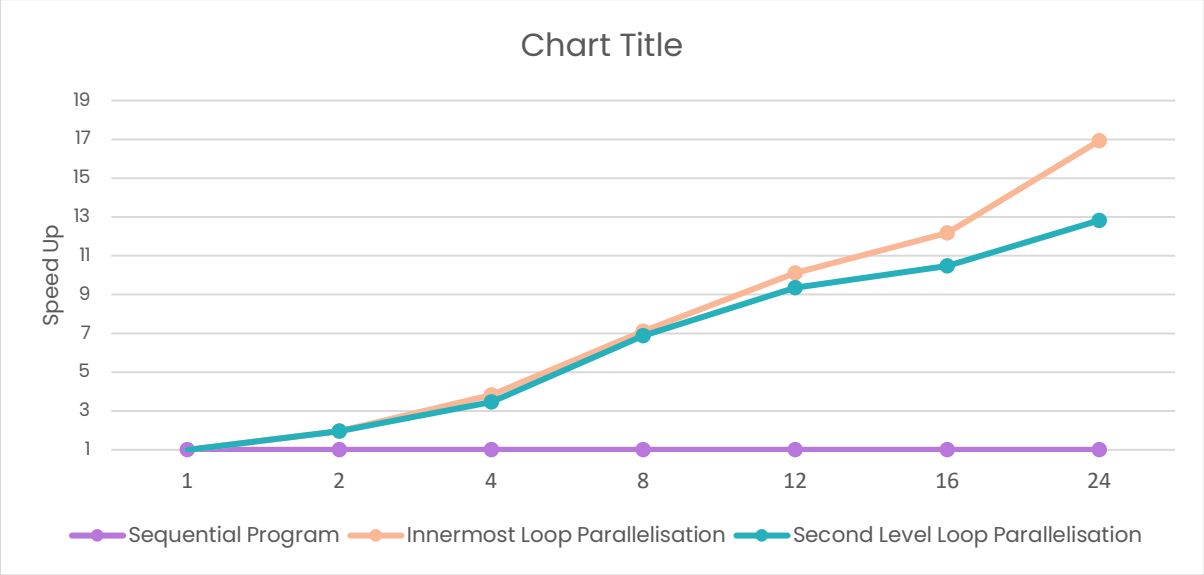

Figure 6: Speedup Graph

Cores Used	Sequential Program	Innermost Loop Parallelisation	Second Level Loop Parallelisation
1	1	0.99	0.99
2	1	1.97	1.97
4	1	3.83	3.46
8	1	7.11	6.87
12	1	10.11	9.36
16	1	12.18	10.48
24	1	16.93	12.83

Figure 7: Speedup Values

These results essentially just confirm we was discussed in the previous section. Both parallelisation techniques see an increased execution time when compared to that of the sequential program. The speedup curves follow a typical sub-linear speedup trajectory, with a slight dip at 16 threads, most likely due to CPU optimisations.

It should be noted though that this is not always the case, specifically in situations where we are using a single processor. This is due to the overhead the comes with creating a thread pool, assigning threads to the pool and managing locks between critical sections of code.

Testing

Techniques

While both of our parallelisation techniques have sped up the execution time of the application, it is of course important to ensure that the program still operates correctly. There is no point making a program run faster if it does not return correct results.

To do this, a test class was created. This class invokes the run functions in both the sequential and parallel versions of the application, and the retrieves the results from both. We can then loop of these results and compare each sequential key with it's associated parallel key.

all Consensus: -35: T T G A C A gap: 17.6 -10: T A T A A T (5430 matches)

Figure 8: Example Result Key

If any of these results do not match, then we should set a Boolean variable to true and return an error stating that the parallel version of the program did not produce the same output as the sequential program. If all of the results match, then the parallelisation of our application was successful.

```
boolean resultsMatch = true;

        HashMap<String, Sigma70Consensus> sequentialResults = Sequential.getConsensus();
        HashMap<String, Sigma70Consensus> parallelResults = Parallel.getConsensus();

        for (String key : sequentialResults.keySet()) {
            String sequentiallKey = sequentialResults.get(key).toString();
            String parallelKey = parallelResults.get(key).toString();
            if (!sequentiallKey.equals(parallelKey)) {
                resultsMatch = false;
                break;
            }
        }

        if (!resultsMatch) {
            System.out.println("The parallel version of the program did not produce the same output as
 the sequential version");
        } else {
            System.out.println("The parallel and sequential outputs match");
        }
```

Figure 9: Testing Function

Results

Sequential	Parallel
all Consensus: -35: T T G A C A gap: 17.6 -10: T A T A A T (5430 matches) fixB Consensus: -35: T T G A C A gap: 17.7 -10: T A T A A T (965 matches) carA Consensus: -35: T T G A C A gap: 17.7 -10: T A T A A T (1079 matches) fixA Consensus: -35: T T G A C A gap: 17.6 -10: T A T A A T (896 matches) caiF Consensus: -35: T T C A A A gap: 18.0 -10: T A T A A T (11 matches) caiD Consensus: -35: T T G A C A gap: 17.6 -10: T A T A A T (550 matches) yaaY Consensus: -35: T T G T C G gap: 18.0 -10: T A T A C T (4 matches) nhaA Consensus: -35: T T G A C A gap: 17.6 -10: T A T A A T (1879 matches) folA Consensus: -35: T T G A C A gap: 17.5 -10: T A T A A T (46 matches)	all Consensus: -35: T T G A C A gap: 17.6 -10: T A T A A T (5430 matches) fixB Consensus: -35: T T G A C A gap: 17.7 -10: T A T A A T (965 matches) carA Consensus: -35: T T G A C A gap: 17.7 -10: T A T A A T (1079 matches) fixA Consensus: -35: T T G A C A gap: 17.6 -10: T A T A A T (896 matches) caiF Consensus: -35: T T C A A A gap: 18.0 -10: T A T A A T (11 matches) caiD Consensus: -35: T T G A C A gap: 17.6 -10: T A T A A T (550 matches) yaaY Consensus: -35: T T G T C G gap: 18.0 -10: T A T A C T (4 matches) nhaA Consensus: -35: T T G A C A gap: 17.6 -10: T A T A A T (1879 matches) folA Consensus: -35: T T G A C A gap: 17.5 -10: T A T A A T (46 matches)
Result	The parallel and sequential outputs match

Tools Used

Compilers

Javac - The Java programming language compiler was used in order to compile source code. The compiler was installed with JDK, and also included with my IntelliJ IDEA distribution, which I used to build the project and debug code.

Software

Intellij - Intellij IDEA is an integrated development environment (IDE) written in Java, and was used to develop this particular application. This included building, debugging and importing libraries.

Visual Studio Code - VS Code a free source-code editor that supports debugging, syntax highlighting, code completion and more. It was used to product a launch.JSON file, which can be used to easily open and run the program.

Tools

Java Executor Service Framework – Executor Service is a framework provided by the JDK which provides a pool of threads and an API for assigning tasks to this thread pool.

Java Callable Interface – Tasks were assigned to the executor framework as callables. These tasks were created an invoked such that the executor service could assign them to run on different threads.

Techniques

Timing – Java's System.nanoTime() method was used throughout the project to analyse the execution time of the application, as well as the loops within it. This helped to identify areas worth parallelising and allowed us to compare the execution time of the sequential program with the parallelised version, in order to calculate speedup.

Array Data Dependence Analysis – Once a loop with significant time cost was found, analysis was completed on said loop in order to deduce if there existed any loop carried dependencies.

Synchronisation – Once a loop carried dependence was located, we used Java's Re-entrant lock to ensure mutual exclusion inside the critical section of code. This allowed us to synchronise writing to the consensus HashMap, and ensure we did not encounter any race conditions.

Problems

Thread Local Memory

A significant problem when initially trying to resolve data dependency issues within the application arose due to a lack of memory management across threads. When looking for the sigma 70 pattern in a particular gene's upstream region, the resultant match value will of course depend on the current gene we are analysing.

For this reason, it is not suitable to store the sigma 70 object in global memory. Doing so will result in 0 matches for all the consensus sequences and thus the data we receive is largely incorrect. To resolve the issue, I had to store the object in thread local memory, by simply defining it using Java's Thread Local class. Then, whenever I called the best match predictor, function, I had to get the object from the thread's local memory.

Synchronisation

Another issue that arose and was significantly more challenging to debug was an issue with synchronisation. It became apparently that I would need some way to lock each thread whenever they were writing to the consensus, otherwise we would see a number of race conditions and the program would not run correctly. Initially I attempted to use Java synchronised methods in order to synchronise my results, however every so often this would introduce small inconsistencies with the parallel program output, when compared to the sequential program output (i.e. 1 less match out of 5000).

I suspect that this was because there is a small portion of time between when a thread invokes the synchronised method, and then moves into the method, locking it from being executed by other threads. Potentially, because each task takes such a short amount of time, and because there are so many tasks, every so often an error would occur.

To resolve this issue, I decided to make use of re-entrant locks, which allow me to specify thread locking and unlocking directly within the thread itself, rather than by having to invoke a separate method. Once this change was made, the error did not occur again.

```
                addLock.lock();
                consensus.get(referenceGene.name).addMatch(prediction);
                consensus.get("all").addMatch(prediction);
                addLock.unlock();
```

Using Lock

```
    public synchronized void addMatch(Match prediction) {
        consensus.get(referenceGene.name).addMatch(prediction);
        consensus.get("all").addMatch(prediction);
    }
```

Using Synchronised Function

Additions

For Development

Source code line count:	23

There were several changes made to the application in order to introduce parallelism. The first was the separation of the innermost for-loop into a callable task. Each task accepts a gene, reference gene and record, and then completes the same tasks as the sequential version on multiple threads. These actions are added to a list and then invoked by the executor service. Once these threads have completed, the executor service will be shutdown. Finally, as stated earlier, the sigma 70 object was moved to thread-local storage using Java's ThreadLocal class. Because most of these changes were just re-factoring the existing code, the actual source code lines changed were minimal.

```
// Add a lock if using this method
private static final ReentrantLock addLock = new ReentrantLock();

private static class GeneTask implements Callable {

    private Gene gene;
    private Gene referenceGene;
    private GenbankRecord record;

    public GeneTask(Gene gene, Gene referenceGene, GenbankRecord record) {
        this.gene = gene;
        this.referenceGene = referenceGene;
        this.record = record;
    }

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

            if (prediction != null) {
                // Lock the thread to ensure flow dependence
                addLock.lock();
                consensus.get(referenceGene.name).addMatch(prediction);
                consensus.get("all").addMatch(prediction);
                addLock.unlock();
            }
        }
        return null;
    }
}
```

Figure 10: Callable class

```
    // Sigma 70 pattern match will depend on state of the thread
    private static ThreadLocal<Series> sigma70_pattern = ThreadLocal.withInitial(
            () -> Sigma70Definition.getSeriesAll_Unanchored(0.7));

    // Update predict promoter to get the thread local object
    private static Match PredictPromoter(NucleotideSequence upStreamRegion) {
            return BioPatterns.getBestMatch(sigma70_pattern.get(), upStreamRegion.toString());
    }
```

Figure 11: Thread Local Storage

```
// Create new executor service to handle the assignment of tasks to threads
// Define thread count in the constructor
private static ExecutorService executorService = Executors.newFixedThreadPool(24);

public static void run(String referenceFile, String dir) throws FileNotFoundException, IOException {
      // Create a list of callable tasks
      List<Callable<Void>> callableList = new ArrayList<>();

      List<Gene> referenceGenes = ParseReferenceGenes(referenceFile);

      for (String filename : ListGenbankFiles(dir)) {
          System.out.println(filename);
          GenbankRecord record = Parse(filename);

          for (Gene referenceGene : referenceGenes) {
              // Parallelising the inner loop
              System.out.println(referenceGene.name);

              for (Gene gene : record.genes) {
                  // Add tasks from the inner for-loop to the callable list
                  callableList.add(new GeneTask(gene, referenceGene, record));
              }
          }
      }

      // Runs all of the callable tasks within the callable list
      try {
          executorService.invokeAll(callableList);
      } catch (InterruptedException e) {
          e.printStackTrace();
      }

      for (Map.Entry<String, Sigma70Consensus> entry : consensus.entrySet()) {
          System.out.println(entry.getKey() + " " + entry.getValue());
      }

      // Shuts down the executor service
      executorService.shutdown();
  }
```

Figure 12: Run function Re-Factor Using Callables

For Testing

Source code line count:	90

Additional lines of code were also added for testing purposes. Some of these can be seen in figure 9 above, others were simply invocations of existing classes, however I did also add a getter method to the consensus object in both the sequential and parallel versions of the program.

```
  // Method to return the consensus results
  public static HashMap<String, Sigma70Consensus> getConsensus() {
      return consensus;
  }
```

Figure 13: Getter Method

This simply allowed me to access the consensus within the testing class, in order to compare the parallel and sequential keys. Alternatively, I could have made the object public.

Reflection

Overall, the object of parallelising the consensus promoter predictor was incredibly successful. We were able to achieve a speedup factor of over 16 with 24 virtual cores available and would most likely be able to achieve even greater results with additional processing power.

Such an outcome is important within the field of biology. The consensus sequence, specifically in the human genome, allows us to further our understanding of normal biological functions and how mutations lead to abnormal functions that cause disease. In doing so, we may be able to work towards mitigating or even curing such diseases.

Personally, throughout the process of parallelisation, I have become very familiar with the constructs available within Java that allow for parallelisation, specifically those available through the Executor Service framework. This includes the creation and sequent invocation of tasks, and also how to manage thread state.

Additionally, my understanding of data and loop dependencies has increased significantly after having to debug and troubleshoot code in which these issues were occurring. I now have a better conceptual understanding of how these dependencies arise, what they look like and how to control them through synchronisation.

Also worth noting is how much my understanding of molecular biology and bioinformatics has increased. I was not even aware of some of the problems this application was trying to solve when I began the project. It was both a great challenge and learning experience, trying fully to understand these concepts.

Speaking of bio-informatics however, one such algorithm that I found quite complicated was the Smith–Waterman algorithm. The algorithm is relatively resource intensive, and there does exists methods for parallelisation using C or CUDA. Perhaps if I were to go through this task again, I would look further into these methods, and see if it would be possible to apply them using Java, however when working through this task initially, these methods seemed out of reach, at least using the Executor Service framework.

Additionally, if given enough bacteria samples to compute, we may reach a stage within the application where it is worthwhile parallelising the outer loop to parse files. This would help increase the scalability of the application, when trying to read from enormous data sets, by creating threads to simultaneously read different files.