CAB401

Promoter

Parallelization Project Report

Anjana Ranjan
N11119985

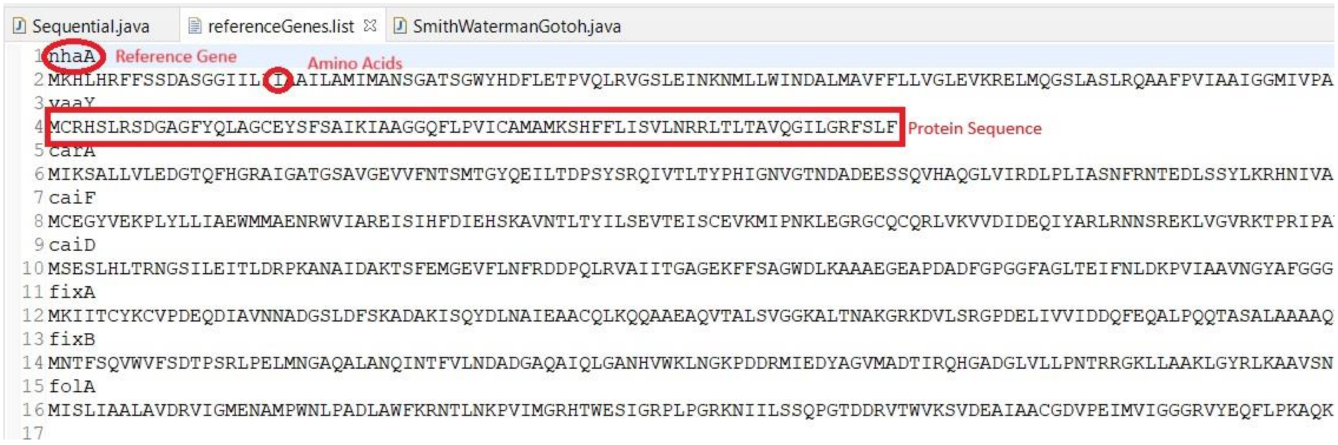
Contents

1 Overview

1.1 High level Overview

This aim of this application is to identify and analyse promoter regions in bacterial genes by comparing them with a set of reference genes. This includes steps such as parsing reference genes and their sequences, and check if it is homologous to any gene in the genbank files using the Smith-Waterman-Gotoh algorithm. If homologous, the promotor region in the upstream DNA is predicted using the Sigma70 consensus sequence and a consensus map is updated and displayed.

To delve deeper into the biology in this app, the referenceGenes.list file contains gene names and sequences. In this file, we have 8 reference genes. Each letter in the sequence encodes a specific amino acid, which form protein sequences.



```
nhaA
MKHLHRFFSSDASGGIILIAILAMIMANSGATSGWYHDFLETPVQLRVGSLEINKNMLLWINDALMAVFFLLVGLEVKRELMQGSLASLRQAAFPVIAAIGGMIVPA
yaaY
MCRHSLRSDGAGFYQLAGCEYSFSAIKIAAGGQFLPVICAMAMKSHFFLISVLNRRLTLTAVQGILGRFSLF
carA
MIKSALLVLEDGTQFHGRAIGATGSAVGEVVFNTSMTGYQEILTDPSYSRQIVTLTYPHIGNVGTNDADEESSQVHAQGLVIRDLPLIASNFRNTEDLSSYLKRHNIVA
caiF
MCEGYVEKPLYLLIAEWMMAENRWVIAREISIHFDIEHSKAVNTLTYILSEVTEISCEVKMIPNKLEGRGCQCQRLVKVVDIDEQIYARLRNNSREKLVGVRKTPRIPA
caiD
MSESLHLTRNGSILEITLDRPKANAIDAKTSFEMGEVFLNFRDDPQLRVAIITGAGEKFFSAGWDLKAAAEGEAPDADFGPGGFAGLTEIFNLDKPVIAAVNGYAFGGG
fixA
MKIITCYKCVPDEQDIAVNNADGSLDFSKADAKISQYDLNAIEAACQLKQQAAEAQVTALSVGGKALTNAKGRKDVLSRGPDELIVVIDDQFEQALPQQTASALAAAAQ
fixB
MNTFSQVWVFSDTPSRLPELMNGAQALANQINTFVLNDADGAQAIQLGANHVWKLNGKPDDRMIEDYAGVMADTIRQHGADGLVLLPNTRRGKLLAAKLGYRLKAAVSN
folA
MISLIAALAVDRVIGMENAMPWNLPADLAWFKRNTLNKPVIMGRHTWESIGRPLPGRKNIILSSQPGTDDRVTWVKSVDEAIAACGDVPEIMVIGGGRVYEQFLPKAQK
```

Figure 1 Reference Genes List

The aim of this application is to identify and analyse promoter regions in bacterial (E.Coli) genomes by comparing them with a set of reference genes. From the code, the function of the application in sequential order is to:

- Parse reference genes and their sequences.
- Parse GenBank files containing E. coli gene sequences.
- For each gene in the GenBank files, check if it's homologous (shares common evolutionary origin) to any reference gene.
- If homologous, predict the promoter region in the upstream DNA sequence of the gene.
- Update a consensus map with the prediction.

1.2 Software Architecture

The `Sequential.java` class is the main entry point for the application, which hosts the main method. The application contains the following methods for different functionalities:

- **Main:** calls a `run` method with a reference gene list and a directory containing E.coli GenBank files.
- **run**: This is the core method that processes reference genes and GenBank files. This method contains 4 for loops, of which 3 are nested.

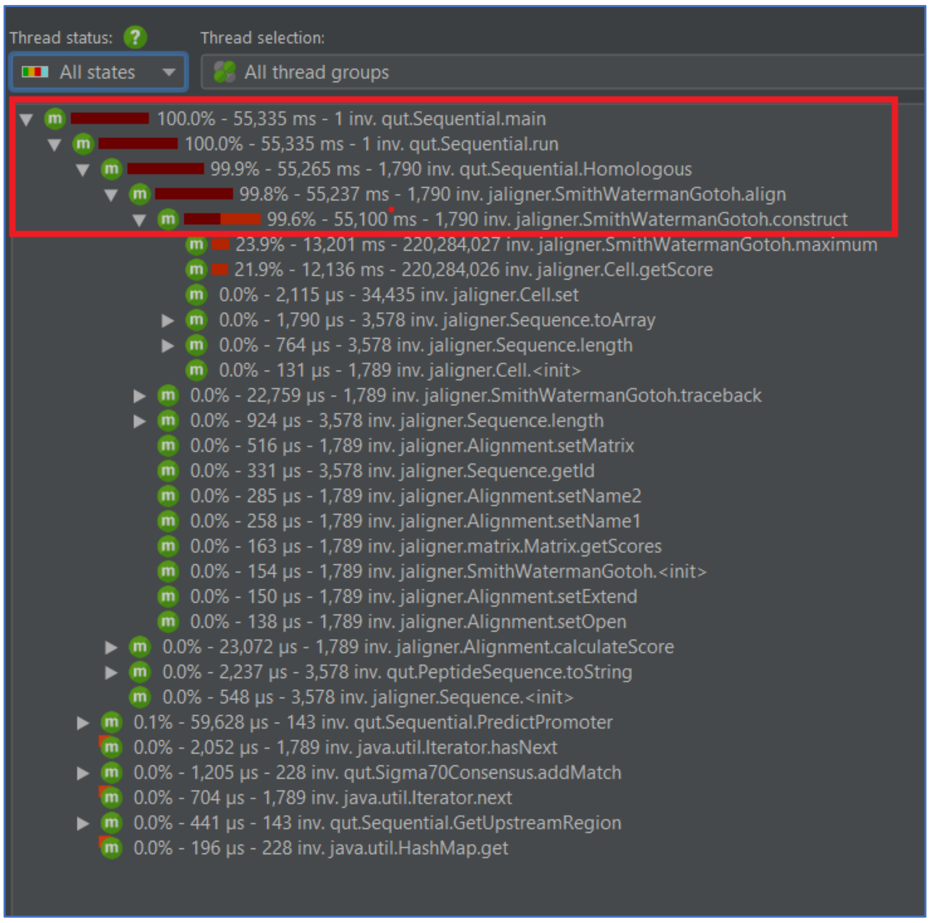
- The first, outermost for loop iterates over the GenBank files, which are quite extensive (up to 20mb). For each record, it parses the file to extract genomic data into a `GenbankRecord` object.
- The second loop iterates over the list of reference genes and checks for homology with the current GenBank record.
- The third, innermost loop iterates over all the genes in the record and checks if the gene is homologous to the current reference gene using the `Homologous` method. If found to be homologous, it predicts the promoter region for that gene and updates the consensus map with the prediction.
- The fourth independent for loop iterates over the consensus map and prints the name of each reference gene and its associated Sigma70 consensus prediction.
- **Parse**: Parses a GenBank file and returns a GenbankRecord object.
- **ParseReferenceGenes**: Reads a reference file and returns a list of reference genes. It also populates the consensus map with Sigma70Consensus objects for each gene.
- **Homologous**: Determines if two peptide sequences are homologous using the Smith-Waterman-Gotoh alignment algorithm.
- **GetUpstreamRegion**: Retrieves the upstream region of a gene from a given DNA sequence.
- **PredictPromoter**: Predicts the promoter region in an upstream DNA sequence using the Sigma70 pattern.
- **ProcessDir & ListGenbankFiles**: Utility methods to list all GenBank files in a given directory.

The application has a modular structure, with a clear separation between the sequence alignment logic (jaligner package) and the biological data handling and analysis logic (qut package). The GenBank files provide the genomic data, and the reference genes list offers a basis for comparison and analysis. The application references the external library jacobi.jar that has utilities for pattern matching promoter regions.

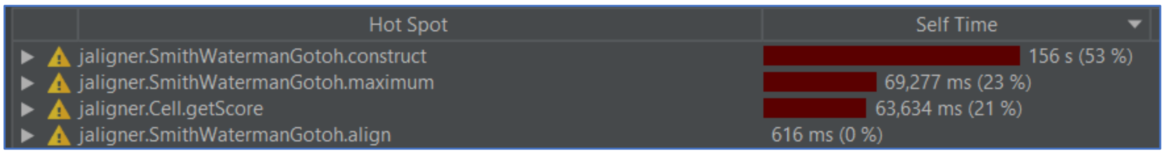
2 Application Analysis

2.1 Initial timing and profiling

Upon analysing the CPU call tree, it's evident that the 'homologous' function, located within the innermost of the three for-loops, is the primary consumer of CPU resources. In this method, the SmithWatermanGotoh algorithm is executed, with the 'align', ‘maximum’ and 'construct' methods accounting for 99.9% of the computational power used. The hotspots also tell a similar story where we can see that the primary hotspots are the different methods contained in the SmithWatermanGotoh module.



```
Thread status:        Thread selection:
All states            All thread groups

▼ 100.0% - 55,335 ms - 1 inv. qut.Sequential.main
  ▼ 100.0% - 55,335 ms - 1 inv. qut.Sequential.run
    ▼ 99.9% - 55,265 ms - 1,790 inv. qut.Sequential.Homologous
      ▼ 99.8% - 55,237 ms - 1,790 inv. jaligner.SmithWatermanGotoh.align
        ▼ 99.6% - 55,100 ms - 1,790 inv. jaligner.SmithWatermanGotoh.construct
              23.9% - 13,201 ms - 220,284,027 inv. jaligner.SmithWatermanGotoh.maximum
              21.9% - 12,136 ms - 220,284,026 inv. jaligner.Cell.getScore
              0.0% - 2,115 µs - 34,435 inv. jaligner.Cell.set
          ▶   0.0% - 1,790 µs - 3,578 inv. jaligner.Sequence.toArray
          ▶   0.0% - 764 µs - 3,578 inv. jaligner.Sequence.length
              0.0% - 131 µs - 1,789 inv. jaligner.Cell.<init>
        ▶ 0.0% - 22,759 µs - 1,789 inv. jaligner.SmithWatermanGotoh.traceback
        ▶ 0.0% - 924 µs - 3,578 inv. jaligner.Sequence.length
          0.0% - 516 µs - 1,789 inv. jaligner.Alignment.setMatrix
          0.0% - 331 µs - 3,578 inv. jaligner.Sequence.getId
          0.0% - 285 µs - 1,789 inv. jaligner.Alignment.setName2
          0.0% - 258 µs - 1,789 inv. jaligner.Alignment.setName1
          0.0% - 163 µs - 1,789 inv. jaligner.matrix.Matrix.getScores
          0.0% - 154 µs - 1,789 inv. jaligner.SmithWatermanGotoh.<init>
          0.0% - 150 µs - 1,789 inv. jaligner.Alignment.setExtend
          0.0% - 138 µs - 1,789 inv. jaligner.Alignment.setOpen
      ▶ 0.0% - 23,072 µs - 1,789 inv. jaligner.Alignment.calculateScore
      ▶ 0.0% - 2,237 µs - 3,578 inv. qut.PeptideSequence.toString
        0.0% - 548 µs - 3,578 inv. jaligner.Sequence.<init>
    ▶ 0.1% - 59,628 µs - 143 inv. qut.Sequential.PredictPromoter
      0.0% - 2,052 µs - 1,789 inv. java.util.Iterator.hasNext
    ▶ 0.0% - 1,205 µs - 228 inv. qut.Sigma70Consensus.addMatch
      0.0% - 704 µs - 1,789 inv. java.util.Iterator.next
    ▶ 0.0% - 441 µs - 143 inv. qut.Sequential.GetUpstreamRegion
      0.0% - 196 µs - 228 inv. java.util.HashMap.get
```



Hot Spot	Self Time
jaligner.SmithWatermanGotoh.construct	156 s (53 %)
jaligner.SmithWatermanGotoh.maximum	69,277 ms (23 %)
jaligner.Cell.getScore	63,634 ms (21 %)
jaligner.SmithWatermanGotoh.align	616 ms (0 %)

Figure 2 CPU HotPath

The entire program takes about 136 seconds (best time recorded in 10 runs) to run through and analyse 4 gbk files. The CPU process load is recorded to be around 6-7%.

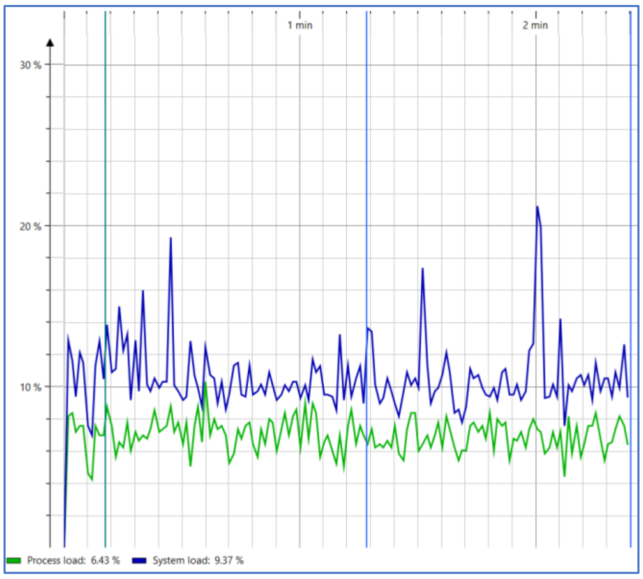


Figure 3 CPU utilization on Sequential run

This application employs a three-tiered nested for-loop that runs sequentially. Within these loops, it performs both computationally intensive tasks, such as the Homologous function, and lighter operations like extracting genes and processing GenBank records. Additionally, there are read and write operations occurring within the loops. Due to this structure, the computation time is significantly extended. By implementing parallelization and optimizing the structure of code, we could potentially reduce this processing time.

2.2 Dependency Analysis

Data Dependency:

Consensus HashMap:

- The consensus HashMap is a shared data structure that stores the Sigma70Consensus for each reference gene.
- Multiple threads updating this shared data structure can lead to race conditions unless synchronized properly.

Reference Genes:

- The referenceGenes list is populated once at the beginning and is read-only during the loops, classifying this as an input dependency. This means there's no write dependency, and multiple threads can safely read from it.

GenbankRecord:

- For each GenBank file, a new GenbankRecord object record is created. This object is local to the loop iteration and does not have shared data dependencies with other iterations.

Homologous Function:

- The Homologous function checks if two sequences are homologous. It doesn't modify any shared data and only returns a boolean value. Thus, it can be safely parallelized.

Upstream Region and Prediction:

- The GetUpstreamRegion and PredictPromoter functions are called within the innermost loop, which are flow dependencies, as it reads values written by earlier statements. They operate on local data and do not modify shared data structures. They can be safely parallelized.

Control Dependency:

Nested Loops:

The three nested for-loops have a clear control flow. The outer loop processes each GenBank file, the middle loop iterates over reference genes, and the innermost loop checks each gene in the GenBank record for homology with the reference gene.

The innermost loop's operations depend on the current GenBank file and reference gene, this could run on separate threads making it a candidate for parallelization.

Conditional Checks:

Within the innermost loop, there's a conditional check using the Homologous function. If two genes are homologous, the subsequent code block is executed. This control dependency needs to be preserved during parallelization.

2.3 Potential Parallelism

The Sequential.java file offers multiple opportunities for parallelism, especially at loop level. However, shared data structures, data dependencies, control dependencies and overhead are some points to consider to ensure efficient and correct parallel execution. Keeping this in mind, here are a few potential places in the code where parallelism can be introduced, and will be worked upon for the remainder of this report:

1. **Processing GenBank Files**:
    - The outermost loop iterates over each GenBank file. Since each file is processed independently, this loop can be parallelized. Each thread or parallel task can handle the processing of a separate GenBank file.
2. **Checking Genes for Homology**:
    - The innermost loop checks each gene in the GenBank record against the reference gene for homology. Since the processing of each gene is independent of others, this loop can also be parallelized. Each thread or parallel task can handle the homology check and subsequent operations for a separate gene.
3. **Homologous Function**:
    - The **Homologous** function determines if two sequences are homologous. This function is stateless (doesn't modify shared data) and can be executed in parallel for different gene pairs.
4. **Extracting Upstream Regions**:
    - The **GetUpstreamRegion** function extracts the upstream region of a gene. Since it operates on local data and doesn't have side effects on shared data structures, multiple threads or tasks can execute this function in parallel for different genes.

5. **Predicting Promoters**:
    - The **PredictPromoter** function predicts the promoter region for a given upstream region. Like the **GetUpstreamRegion** function, it's stateless and can be executed in parallel for different upstream regions.
6. **Displaying Results**
    - As the order of output is not of grave importance, this loop can also be parallelised. However, as this is a simple read-write operation, parallelising it may introduce overhead and could also lead to race conditions where multiple threads would attempt to print to the console simultaneously.

3 Computation

3.1 Hardware

- Processor: Intel Core i5 10300H, 2021
- CPU Clock: 2.5 GHz
- 4 Physical Cores
- 8 Virtual Cores
- 32GB RAM
- Cache:
    - L1 Data 4 x 32 KB 8-way
    - L1 Instruction 4 x 32 KB 8-way
    - L2 4 x 256 KB 4-way
    - L3 8 MB 16-way

3.2 Software

- OS: Windows 11
- OpenJDK Version: jdk-17.0.8.1, 64-bit
- IDE: Eclipse IDE Version 4.16.0
- Profiling tool: Jprofiler

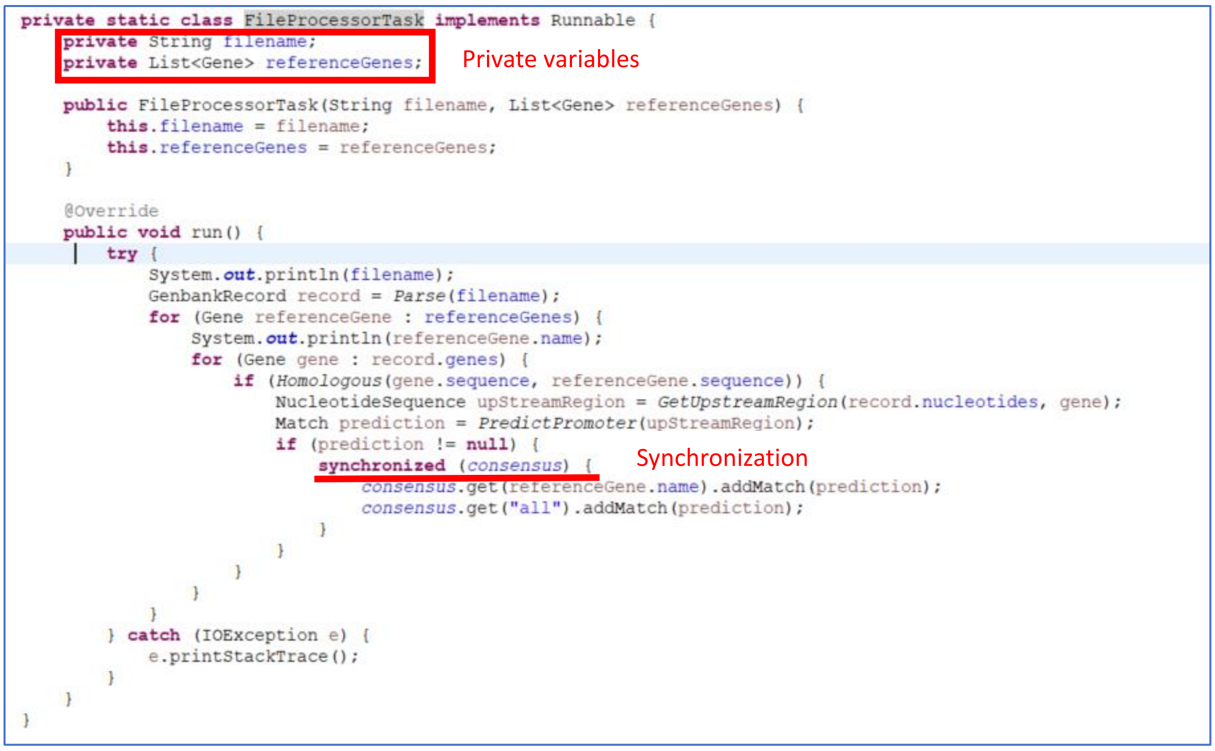
Parallelisation Techniques

3.3 Explicit Threading

Explicit threading is the first parallelization technique applied. An attempt was made to explicitly create and manage threads to perform concurrent tasks, which in this case involved creating separate threads for processing each GenBank file concurrently. The approach aimed to reduce overall execution time by leveraging the capabilities of multi-core processors, which in this case used 4 processors for the 4 GenBank files provided, therefore should culminate to a speedup of 3 (4 genbank files execute concurrently and complete in the time of 1).

Following were the changes made to implement this technique:

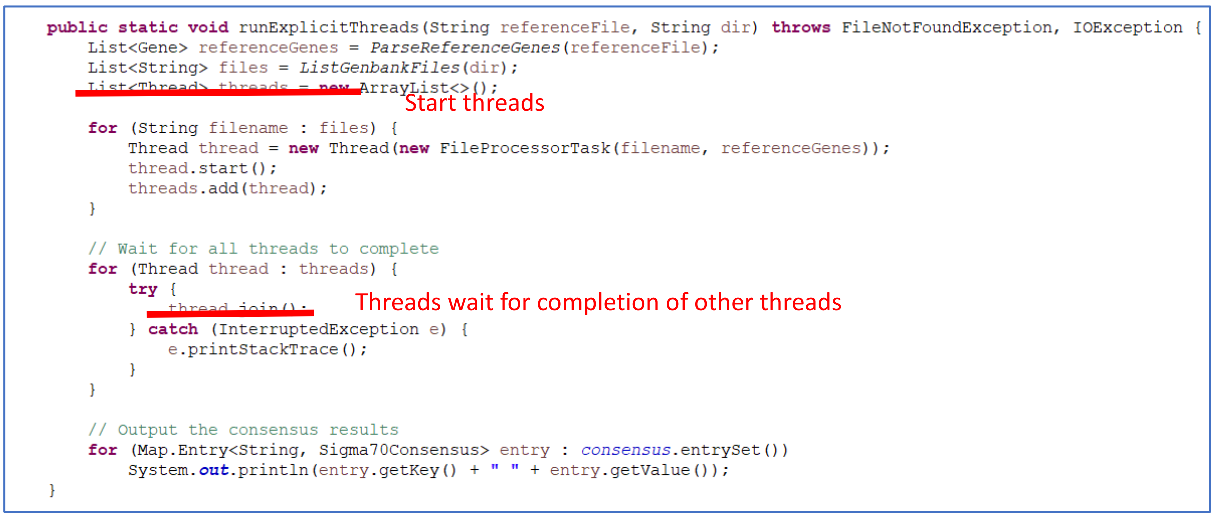
1. **FileProcessorTask**: A new inner class FileProcessorTask is created, implementing Runnable. It processes a single GenBank file. Private variables were added to enable data isolation for each task and maintaining thread safety.



```java
private static class FileProcessorTask implements Runnable {
    private String filename;
    private List<Gene> referenceGenes;

    public FileProcessorTask(String filename, List<Gene> referenceGenes) {
        this.filename = filename;
        this.referenceGenes = referenceGenes;
    }

    @Override
    public void run() {
        try {
            System.out.println(filename);
            GenbankRecord record = Parse(filename);
            for (Gene referenceGene : referenceGenes) {
                System.out.println(referenceGene.name);
                for (Gene gene : record.genes) {
                    if (Homologous(gene.sequence, referenceGene.sequence)) {
                        NucleotideSequence upStreamRegion = GetUpstreamRegion(record.nucleotides, gene);
                        Match prediction = PredictPromoter(upStreamRegion);
                        if (prediction != null) {
                            synchronized (consensus) {
                                consensus.get(referenceGene.name).addMatch(prediction);
                                consensus.get("all").addMatch(prediction);
                            }
                        }
                    }
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Figure 4 FileProcessorTask class

2. **runExplicitThreads method**: Is a modification of the run method and uses an `ExecutorService` to manage threads. It creates and starts a new thread for each GenBank file using the FileProcessorTask.



```java
public static void runExplicitThreads(String referenceFile, String dir) throws FileNotFoundException, IOException {
    List<Gene> referenceGenes = ParseReferenceGenes(referenceFile);
    List<String> files = ListGenbankFiles(dir);
    List<Thread> threads = new ArrayList<>();

    for (String filename : files) {
        Thread thread = new Thread(new FileProcessorTask(filename, referenceGenes));
        thread.start();
        threads.add(thread);
    }

    // Wait for all threads to complete
    for (Thread thread : threads) {
        try {
            thread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

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

Figure 5 runExplicitThreads method

3. **Thread Management**: this method directly manages threads by creating and starting them. It also waits for all threads to complete using `thread.join()`.
4. **Synchronization**: Access to the shared consensus HashMap is synchronized using a `synchronized` block to prevent race conditions.
5. **Await Termination**: After submitting all tasks, the method waits for all threads to complete using `executor.awaitTermination`.
6. **Output**: After all threads have completed, the method outputs the consensus results.

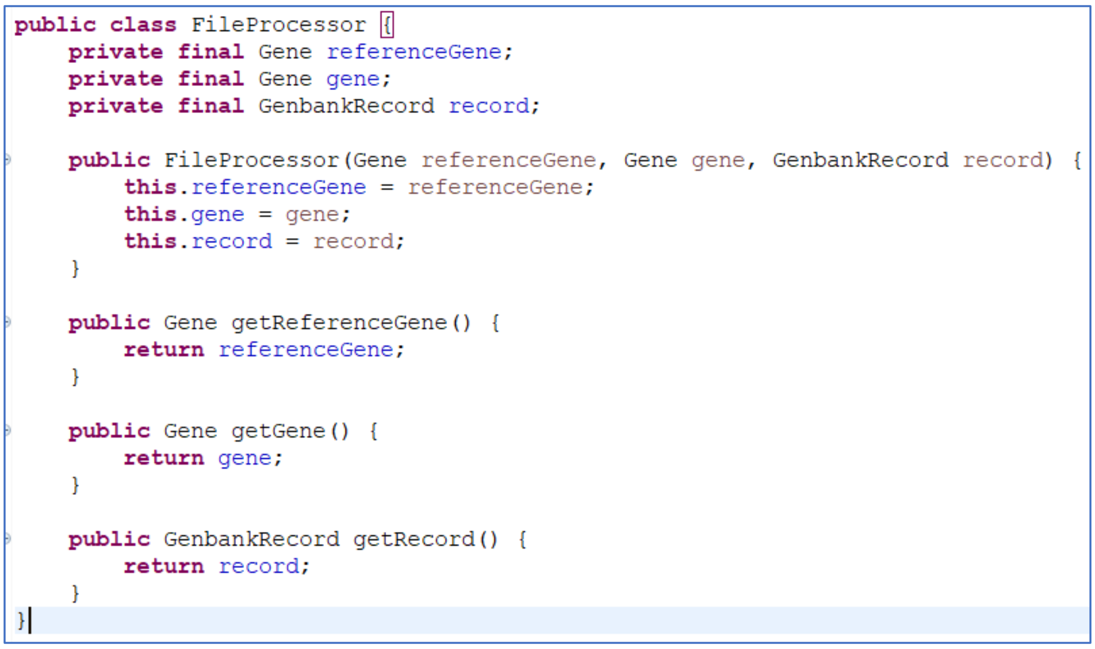
7. **Error Handling**: The run method in the FileProcessorTask class includes error handling for IOException.

3.4 Parallel Stream

Parallel streams are part of the Java Stream API introduced in Java 8. They provide a straightforward and high-level way to parallelize certain operations on separate cores. It can be considered as a form of implicit threading abstract away the complexity of managing threads and thread pools. The ForkJoin framework, which backs parallel streams, automatically divides the tasks into smaller chunks and assigns them to different threads. This work-stealing algorithm helps in efficiently utilizing CPU resources.

Following were the changes made to implement this technique:

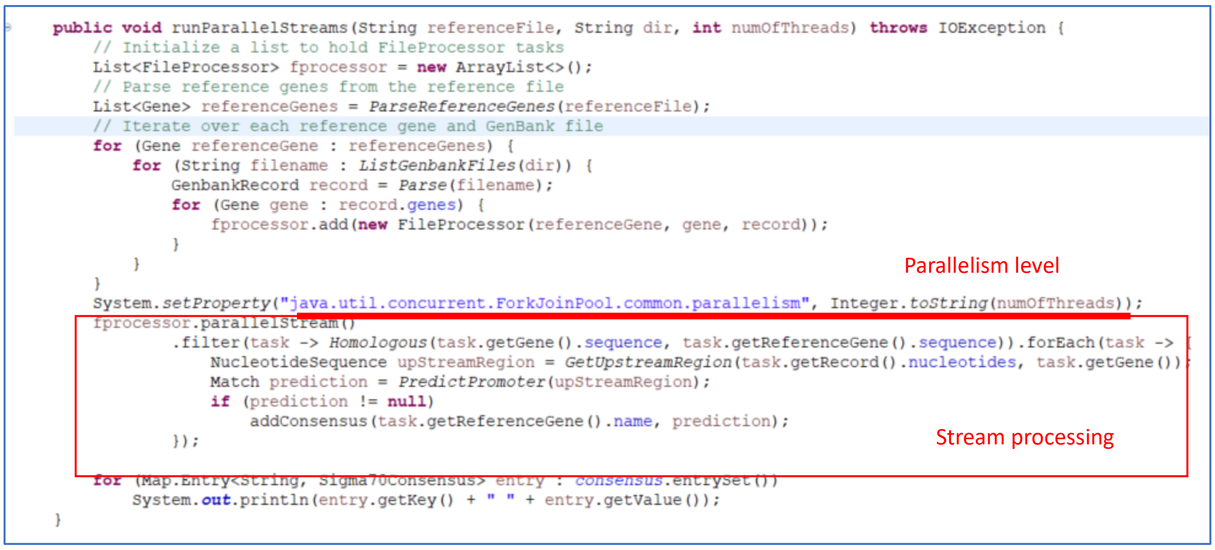
1. **FileProcessor Class**: A new class `FileProcessor` is created to encapsulate the data and operations for processing a single GenBank file. This class holds references to a referenceGene, a gene, and a record, ensuring data isolation for each task and maintaining thread safety.



```java
public class FileProcessor {
    private final Gene referenceGene;
    private final Gene gene;
    private final GenbankRecord record;

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

    public Gene getReferenceGene() {
        return referenceGene;
    }

    public Gene getGene() {
        return gene;
    }

    public GenbankRecord getRecord() {
        return record;
    }
}
```

Figure 6 File processor class

2. **Data Preparation**: The runParallelStreams method begins by preparing the data. It creates a list of FileProcessor instances, each representing a task to be processed. It iterates over the reference genes and GenBank files, creating a FileProcessor for each combination of reference gene and gene in the GenBank file.



```java
public void runParallelStreams(String referenceFile, String dir, int numOfThreads) throws IOException {
    // Initialize a list to hold FileProcessor tasks
    List<FileProcessor> fprocessor = new ArrayList<>();
    // Parse reference genes from the reference file
    List<Gene> referenceGenes = ParseReferenceGenes(referenceFile);
    // Iterate over each reference gene and GenBank file
    for (Gene referenceGene : referenceGenes) {
        for (String filename : ListGenbankFiles(dir)) {
            GenbankRecord record = Parse(filename);
            for (Gene gene : record.genes) {
                fprocessor.add(new FileProcessor(referenceGene, gene, record));
            }
        }
    }
    System.setProperty("java.util.concurrent.ForkJoinPool.common.parallelism", Integer.toString(numOfThreads));
    fprocessor.parallelStream()
            .filter(task -> Homologous(task.getGene().sequence, task.getReferenceGene().sequence)).forEach(task -> {
                NucleotideSequence upStreamRegion = GetUpstreamRegion(task.getRecord().nucleotides, task.getGene());
                Match prediction = PredictPromoter(upStreamRegion);
                if (prediction != null)
                    addConsensus(task.getReferenceGene().name, prediction);
            });

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

Figure 7 runParallelStreams method

3. **Setting Parallelism Level**: The method sets the level of parallelism for the common ForkJoinPool by setting the system property java.util.concurrent.ForkJoinPool.common.parallelism to the desired number of threads (`numOfThreads`). This controls the number of threads used by parallel streams.

4. **Parallel Stream Processing**: The method uses a parallel stream to process the list of FileProcessor tasks. It filters tasks based on the homology of gene sequences and then processes each task in parallel. The processing involves getting the upstream region, predicting promoters, and updating the consensus if a match is found.

5. **Consensus Update**: The method includes a synchronized block within the `addConsensus` function to safely update the shared consensus HashMap. This prevents race conditions when multiple threads try to update the consensus concurrently.

6. **Output**: After processing all tasks, the method outputs the consensus results by iterating over the entries in the consensus HashMap.

7. **Error Handling**: Error handling is incorporated to catch and handle IOException that may occur during file processing.

3.5 Executor Service

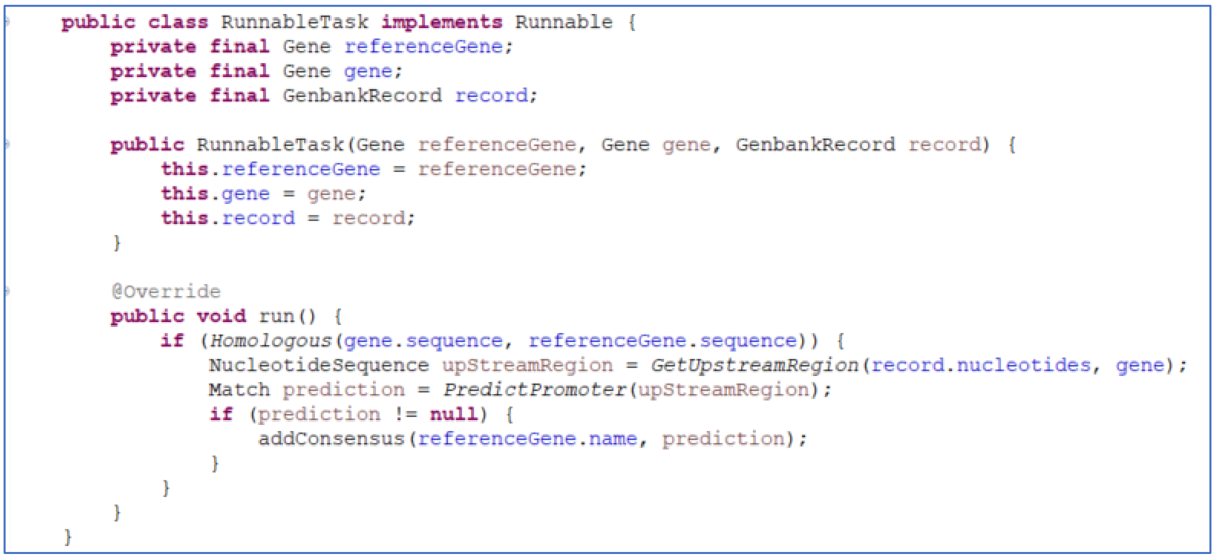
ExecutorService in Java is a framework provided by the `java.util.concurrent` package that simplifies the execution of tasks in asynchronous mode. It provides a way to manage a pool of threads and assign tasks to them, handling thread creation, life cycle, and task submission.

In this application, we use this technique to set each thread to processes a part of the workload (each GenBank files). By using ExecutorService, the function can easily manage the threads, submit tasks to them, and wait for their completion. This leads to better resource utilization and potentially improved performance compared to manually managing threads.

Following were the changes made to implement this technique:

1. **RunnableTask Class**: A new inner class RunnableTask is created, implementing the Runnable interface. This class is designed to process a single task, which involves comparing gene

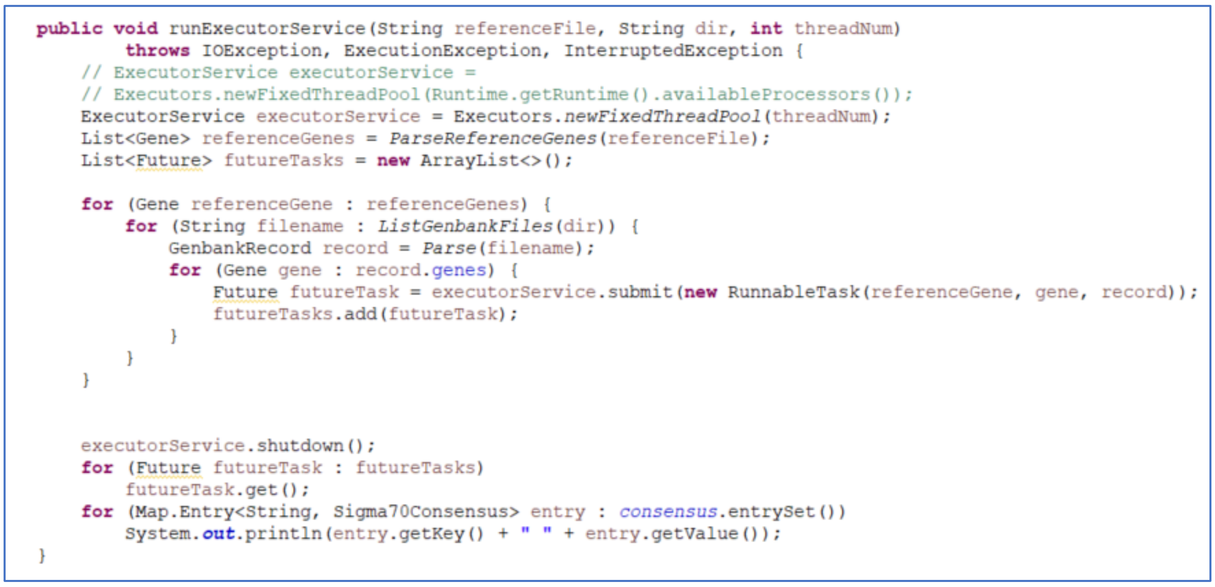
sequences, predicting promoters, and updating the consensus. Private variables referenceGene, gene, and record are added to enable data isolation for each task, maintaining thread safety.



```java
public class RunnableTask implements Runnable {
    private final Gene referenceGene;
    private final Gene gene;
    private final GenbankRecord record;

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

    @Override
    public void run() {
        if (Homologous(gene.sequence, referenceGene.sequence)) {
            NucleotideSequence upStreamRegion = GetUpstreamRegion(record.nucleotides, gene);
            Match prediction = PredictPromoter(upStreamRegion);
            if (prediction != null) {
                addConsensus(referenceGene.name, prediction);
            }
        }
    }
}
```

Figure 8 runnableTask class

2. **runExecutorService Method**: This method is a modification of the original run method. It uses an ExecutorService to manage threads efficiently. The method creates and submits a new RunnableTask for each gene in each GenBank file.



```java
public void runExecutorService(String referenceFile, String dir, int threadNum)
        throws IOException, ExecutionException, InterruptedException {
    // ExecutorService executorService =
    // Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());
    ExecutorService executorService = Executors.newFixedThreadPool(threadNum);
    List<Gene> referenceGenes = ParseReferenceGenes(referenceFile);
    List<Future> futureTasks = new ArrayList<>();

    for (Gene referenceGene : referenceGenes) {
        for (String filename : ListGenbankFiles(dir)) {
            GenbankRecord record = Parse(filename);
            for (Gene gene : record.genes) {
                Future futureTask = executorService.submit(new RunnableTask(referenceGene, gene, record));
                futureTasks.add(futureTask);
            }
        }
    }


    executorService.shutdown();
    for (Future futureTask : futureTasks)
        futureTask.get();
    for (Map.Entry<String, Sigma70Consensus> entry : consensus.entrySet())
        System.out.println(entry.getKey() + " " + entry.getValue());
}
```

Figure 9 runExecutorService method

3. **ExecutorService**: An ExecutorService is used to manage threads. It's set to use a fixed thread pool size, which can be specified as a parameter (threadNum). This allows for control over the number of threads used for parallel processing.

4. **Task Submission**: The method iterates over reference genes and GenBank files, creating and submitting RunnableTask instances to the ExecutorService. Each task is added to a list of Future objects, allowing for tracking and synchronization.

5. **Shutdown and Await Termination**: After submitting all tasks, the ExecutorService is shut down to stop accepting new tasks. The method then waits for all tasks to complete by iterating over the Future objects and calling get() on each, ensuring all tasks are finished before proceeding.

6. **Output**: After all tasks have completed, the method outputs the consensus results by iterating over the entries in the consensus HashMap.

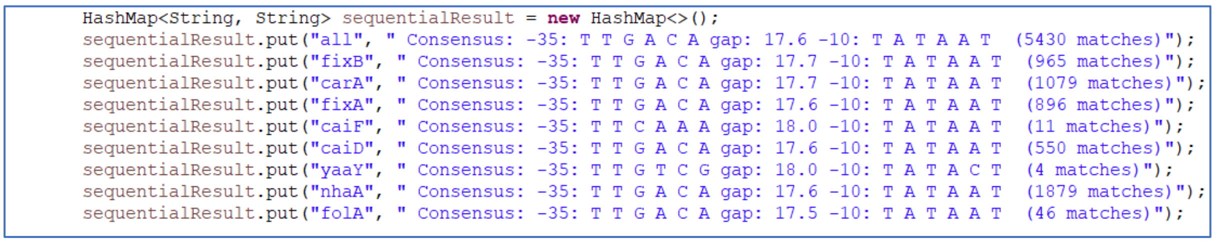
7. **Error Handling**: Error handling is included in the run method of the RunnableTask class to catch and handle any IOException that might occur during the processing of files.

3.6 Testing

3 separate unit tests were written to compare the results of the 3 different parallelisation techniques. In this `ParallelTest` class:

1. An instance of the Parallel class is created and declare default consensus results.

2. The original sequential version was declared using a HashMap object to save time during testing. Nevertheless, the main sequential class can also be made to run. The parallel test functions with the same input dataset were declared.



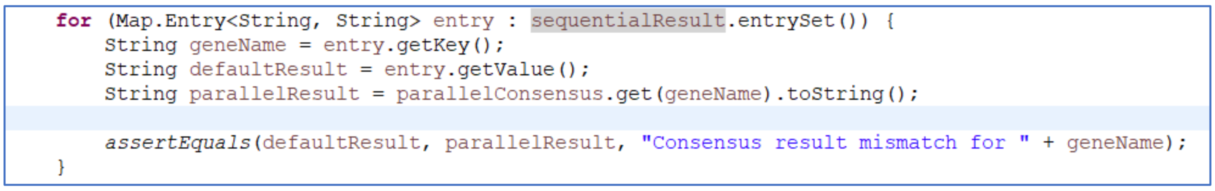
```
        HashMap<String, String> sequentialResult = new HashMap<>();
        sequentialResult.put("all", " Consensus: -35: T T G A C A gap: 17.6 -10: T A T A A T  (5430 matches)");
        sequentialResult.put("fixB", " Consensus: -35: T T G A C A gap: 17.7 -10: T A T A A T  (965 matches)");
        sequentialResult.put("carA", " Consensus: -35: T T G A C A gap: 17.7 -10: T A T A A T  (1079 matches)");
        sequentialResult.put("fixA", " Consensus: -35: T T G A C A gap: 17.6 -10: T A T A A T  (896 matches)");
        sequentialResult.put("caiF", " Consensus: -35: T T C A A A gap: 18.0 -10: T A T A A T  (11 matches)");
        sequentialResult.put("caiD", " Consensus: -35: T T G A C A gap: 17.6 -10: T A T A A T  (550 matches)");
        sequentialResult.put("yaaY", " Consensus: -35: T T G T C G gap: 18.0 -10: T A T A C T  (4 matches)");
        sequentialResult.put("nhaA", " Consensus: -35: T T G A C A gap: 17.6 -10: T A T A A T  (1879 matches)");
        sequentialResult.put("folA", " Consensus: -35: T T G A C A gap: 17.5 -10: T A T A A T  (46 matches)");
```

Figure 10 Sequential Result stored while testing

3. After both runs are complete, we compare the consensus results obtained from both versions, and alert if a mismatch is found.

A `compareConsensusResults` method is declared to compare the consensus results for each gene and outputs whether they match or not. If there is a mismatch, it prints the default and parallel results for further examination.

Make sure to adjust the file paths and the number of threads used in the parallel version according to your setup.



```
    for (Map.Entry<String, String> entry : sequentialResult.entrySet()) {
        String geneName = entry.getKey();
        String defaultResult = entry.getValue();
        String parallelResult = parallelConsensus.get(geneName).toString();

        assertEquals(defaultResult, parallelResult, "Consensus result mismatch for " + geneName);
    }
```

Figure 11 comparison of sequential and parallel results for testing

The tests concluded with all the parallelization techniques achieving correct outcomes that match the results from the sequential run.

4 Speedup Result

The speedup of all techniques (average of 10 runs) is documented in the table below. This was calculated as a ratio of best sequential time (taken as 140s) to the execution time of the 3 parallel algorithms.

Number of cores	Linear Speedup	Explicit Threading Speedup	Parallel streams Speedup	Executor Service Speedup
1	1	1.00	1.75	1.04
2	2		2.03	1.79
3	3		2.69	2.59
4	4	2.92	3.11	3.33
5	5		3.68	3.59
6	6		3.89	4.12
7	7		4.12	4.24
8	8		4.24	4.38

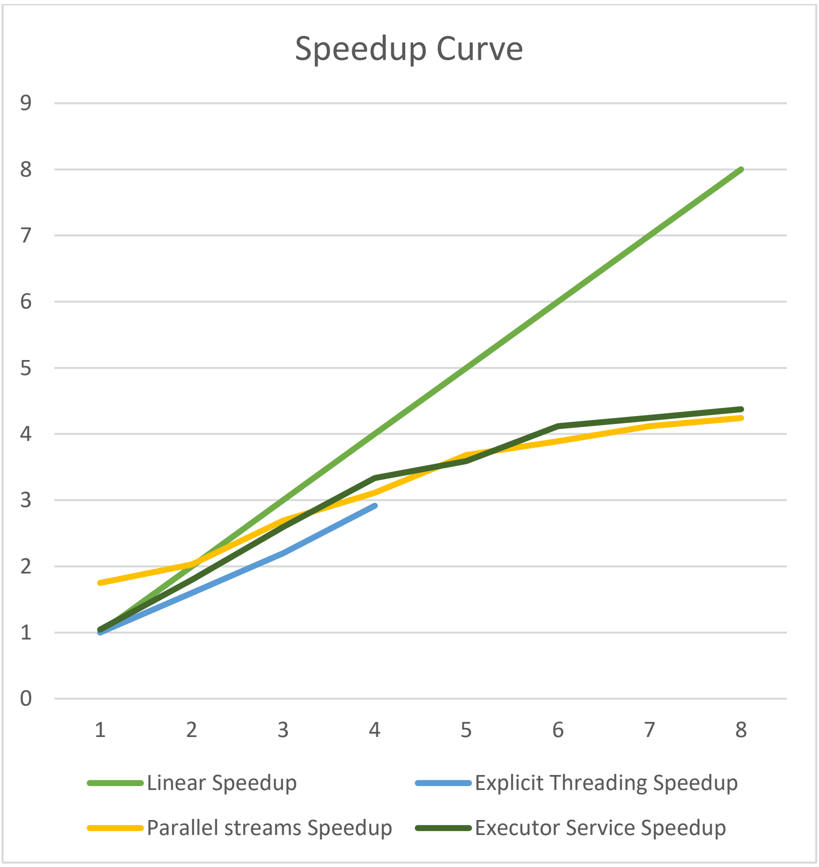


Figure 12 Speedup curve for different parallelisation techniques

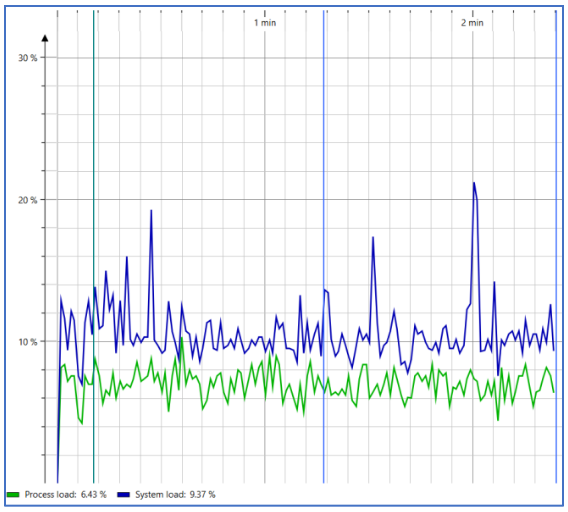


Figure 13 CPU utilization on sequential run before parallelisation

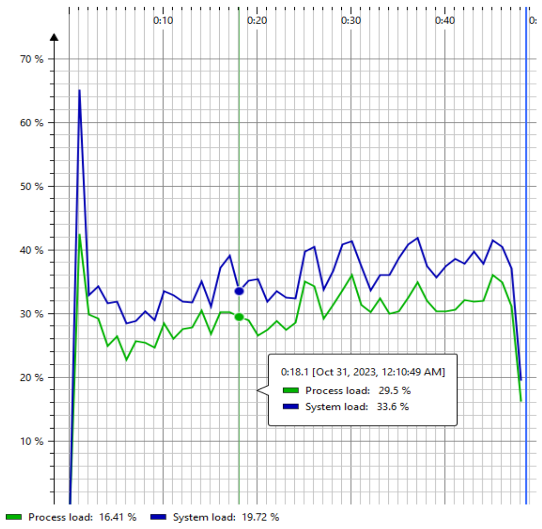


Figure 14 CPU utilization of explicit threading running 4 threads

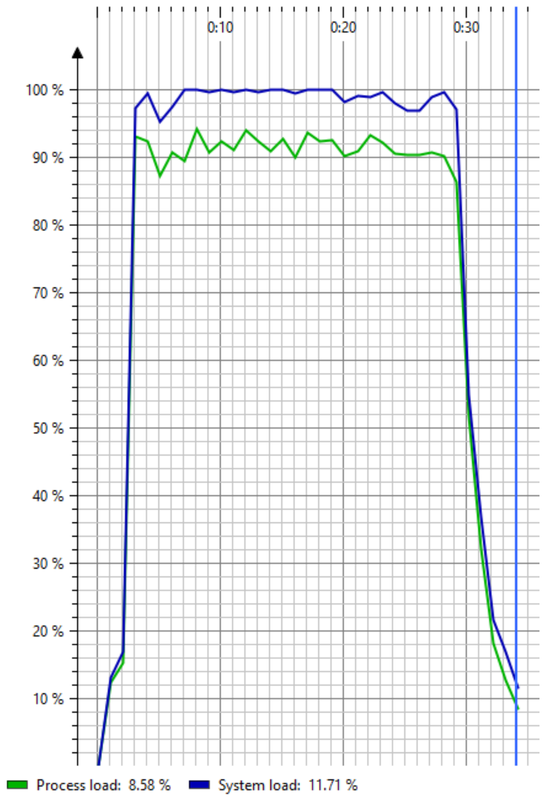


Figure 16 CPU utilization on parallel streams running 8 threads

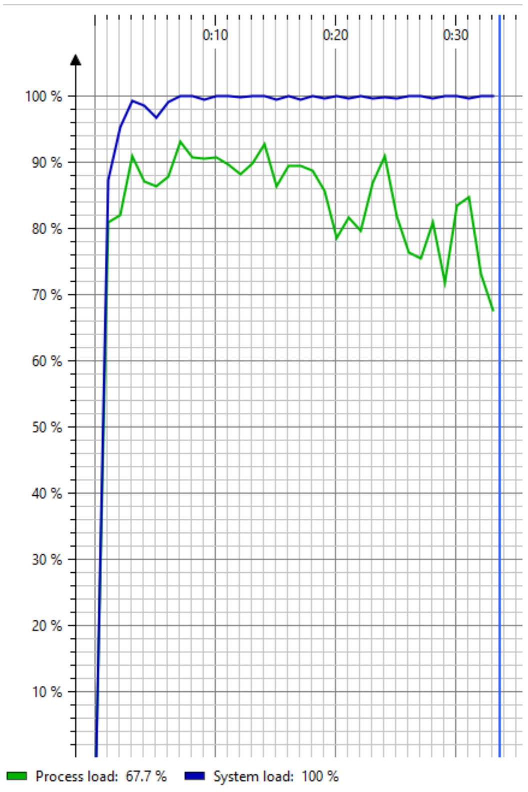


Figure 15 CPU utlilization on ExecutorService running 8 threads

4.1 Evaluation

Explicit threading experienced a speedup of about 3 times using 4 cores (for each file). This method exhibited the least speedup, and can be attributed to the fact that the thread management was more manual. For complex programs, it can get more complicated to organize and add tasks to threads to make the most use of all CPU cores. Implicit threading mechanisms solves this problem by automating task distribution for us.

Parallel streams and Executor Service achieved a speedup > 4 (4.4 max). The speedup curve for these 2 methods exhibit the typical sub-linear speedup pattern. Although speedup is observed as we increment the number of threads in each step, it doesn't tend to be linear especially when we reach a higher thread count. This can be attributed to the overhead of splitting tasks and managing a large number concurrent threads.

Overall, speedup curve highlights the trade-offs and benefits of different parallelization techniques. While explicit threading can provide speedup, it suffers from overheads. Parallel Streams and Executor Service are more efficient, with Executor Service offering the best speedup especially for higher number of threads, indicating better thread management. This further goes to say that proper thread management and load balancing are essential for achieving optimal parallelism.

5 Overcoming Barriers

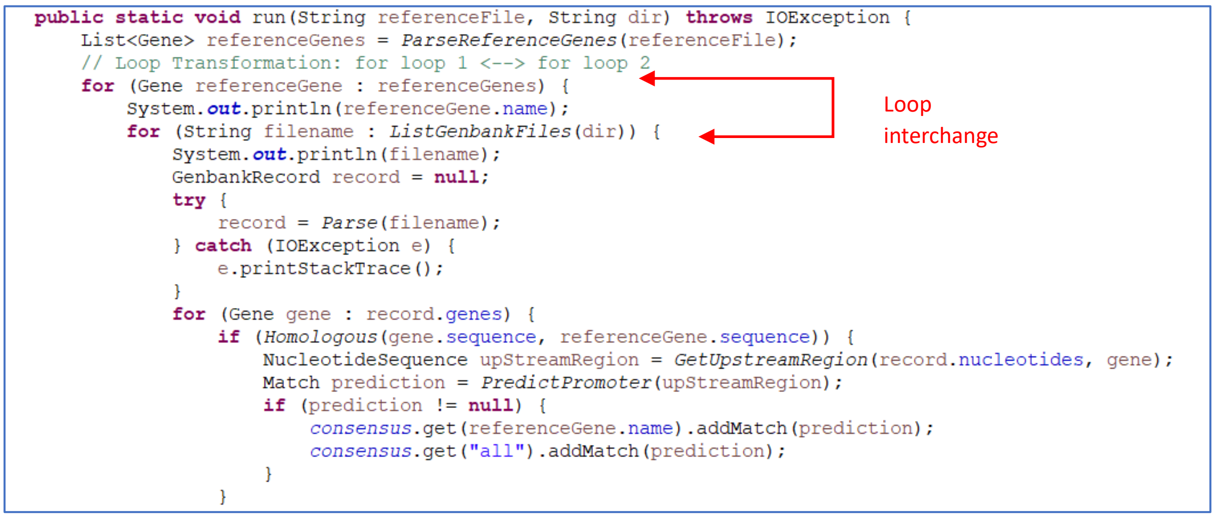
5.1 Data Locality

5.1.1.1 Loop Transformation

On close inspection of the 3 nested for loops in the original sequential run method, I noticed that the 'record' object was accessed in the outermost and the innermost loop. To enhance spatial and temporal locality, these 3 nested loops were rearranged to make for loop 2 encompass for loop 1 and 3. The new sequence for this 3-tried nested for loop is as follows:

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

Although this alteration places referenceGene further from the inner section (specifically the if (Homologous(…){ …}) part), the frequent usage of the variable ensures that its memory location remains cached.



```java
public static void run(String referenceFile, String dir) throws IOException {
    List<Gene> referenceGenes = ParseReferenceGenes(referenceFile);
    // Loop Transformation: for loop 1 <--> for loop 2
    for (Gene referenceGene : referenceGenes) {
        System.out.println(referenceGene.name);
        for (String filename : ListGenbankFiles(dir)) {
            System.out.println(filename);
            GenbankRecord record = null;
            try {
                record = Parse(filename);
            } catch (IOException e) {
                e.printStackTrace();
            }
            for (Gene gene : record.genes) {
                if (Homologous(gene.sequence, referenceGene.sequence)) {
                    NucleotideSequence upStreamRegion = GetUpstreamRegion(record.nucleotides, gene);
                    Match prediction = PredictPromoter(upStreamRegion);
                    if (prediction != null) {
                        consensus.get(referenceGene.name).addMatch(prediction);
                        consensus.get("all").addMatch(prediction);
                    }
                }
```

Figure 17 Loop transformation

This process had a minor effect to the tune of a few milliseconds (3ms on an average of 10 runs vs original sequential code).

Other restructuring was also tested, but didn't have much effect and logical justification.

5.2 Data Race

5.2.1.1 Locking Mechanism

The ‘synchronised’ keyword in java was used to lock the updation of the `consensus` map. It is used to ensure that only one thread at a time can modify the consensus map as multiple threads are processing files and updating the consensus map concurrently. Without synchronization, there could be data races leading to incorrect or unpredictable updates to the map.

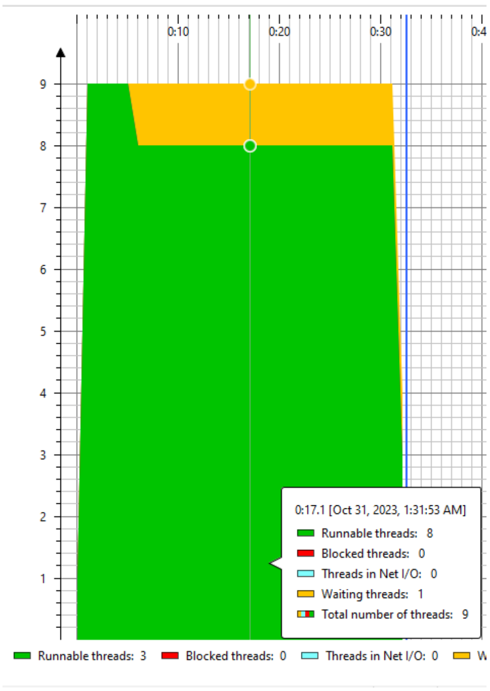
The addConsensus method is declared with the synchronized keyword, making the entire method a synchronized block.

```java
public synchronized void addConsensus(String name, Match prediction) {
    consensus.get(name).addMatch(prediction);
    consensus.get("all").addMatch(prediction);
}
```

Figure 18 preventing data race with synchronised block

The ‘ReentrantLock` method was also tried, but ultimately provided the same results. Synchronised was kept due to its simplicity, and its intrinsic lock features.

There seems to not be much time spent by the threads waiting. **8 threads** (corresponding to 8 cores) were initialised to run concurrently for the executor service and parallel streams methods and the following results were recorded:



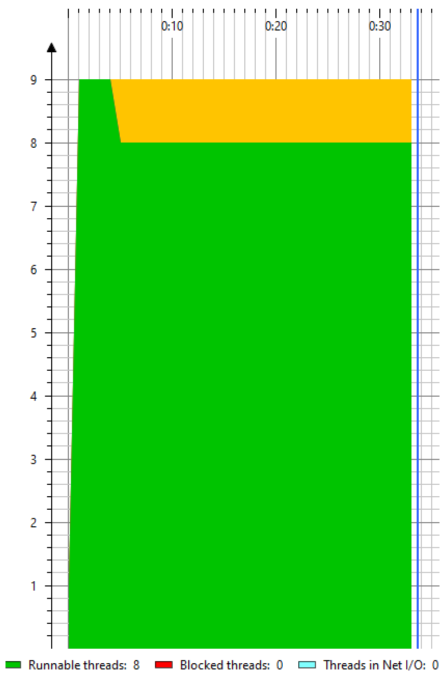


Figure 20 Thread level of parallelStreams running 8 threads

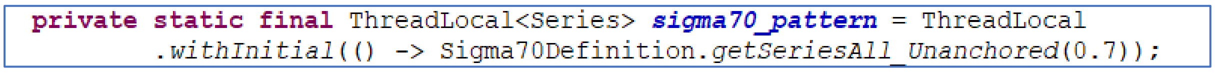
Figure 19 Thread level exectution profile of executorService running 8 threads

Overall, there was no excessive wait time was observed for the synchronization process. There doesn’t seem to be much overhead involved in the process of acquiring and releasing locks utilising the synchronisation mechanism.

5.2.1.2 Thread safety mechanism

Since we work in a multi-threaded environment, it becomes necessary for each thread to have its own copy of variables. In this instance, it became imperative to isolate the instance of the sigma70_patetrn series object to ensure that changes made by one thread did not affect others.

As this resource was being accessed very frequently, synchronization would lead to contention (IndexOutOfBound errors) and reduced performance. By providing each thread with its own instance of the Series object, ThreadLocal eliminated the need for synchronization in this context, thereby improving performance.



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

Figure 21 ThreadLocal<Series> for sigma70_pattern in the code is a strategic decision to ensure thread safety

5.3 Load Imbalance

5.3.1.1 Dynamic Scheduling

The dynamic scheduling mechanism used in this code is the work-stealing algorithm provided by the ForkJoinPool, which is leveraged by the parallel streams API in Java. When a thread completes its tasks, it can "steal" work from other threads that are still busy. This ensures that all threads remain as busy as possible, reducing idle time and improving overall efficiency. This mechanism allows for efficient load balancing and utilization of processing resources in a multi-threaded environment.

It's important to note that this dynamic scheduling is implicit and managed by the Java runtime environment. As a developer, I did not need to implement the scheduling logic manually; it is handled automatically by the parallel streams framework and the underlying ForkJoinPool.

5.3.1.2 Over Decomposition

From a first few tries, it doesn't seem like the app could benefit much from allocating threads > number of cores. The speedup achieved is very minimal in comparison to the application running on 8 threads for both the parallelisation techniques. After 12 threads, there seems to be significant overhead overbearing any improvements in speedup.

6 Conclusion

Initially, the complex program was broken down to improve understanding. The software architecture and ways of working of this codebase was understood, after which the original application was timed and profiled. This took about 2-3 minutes to run in my environment. The aim was to achieve at least 4 times this speed. The hotpath of the program was mapped to glance at what methods consume the most resources and time, so that we can cut this down using parallelism.

The program was evaluated for dependency analysis and all the different types of data dependencies were documented. This was a critical exercise to conduct before finding out potential parallelism sites as it gave a clear picture of how to manipulate different objects, which consequently allowed all our parallelism results to match with the output of the sequential program.

Three different types of parallelism techniques were employed- namely explicit threading, parallel streams and executor services. The ‘synchronisation’ keyword was used to manage critical sections. Other thread safety measures were also in place to make sure each threads had access to it’s own local variables. Executor service experienced the most speedup (4.38) when it came to using higher number of threads (8 threads). This result met the initial expectations and we were able to leverage all 8 cores of the CPU to almost 100% of its capability.

7 Reflection

As a person who doesn’t have much of a background in biology and genetics, it was initially very challenging to wrap my head around what the program is trying to achieve. Thinking from the technical lens and doing a high-level research for the report helped me in this process.

The overall parallelism aspect needed a very deep understanding of the code base. It was a significant learning curve for me and I was challenged but also enjoyed the thrilling experience making code work faster. When the 3 methods I explored yielded good results, I knew I was going in the right path. The lectures and materials available for this unit helped me understand the terminologies much better and also taught me significantly about evaluation of barriers and how to overcome them. In the end, I am quite happy with the result but I would improve on the evaluation of the hotpath and cutting down time taken by most computation heavy parts of the app.

Some of my takeways are that Implicit threading always works better in terms of speedup and it is much more efficient to let the program decide and manage threads on its own. What I learned in terms of different parallelisation methods is that there can be multiple different ways to parallelise a program. Not all developers need to follow the similar technique. Speedup is also reliant on the environment we work on, especially hardware specifications. There is no particular parallelisation technique that can fit all cases and achieve best performance. A lot of trail and error goes into parallelising a program and it is a work of patience and dedication.

8 Appendix

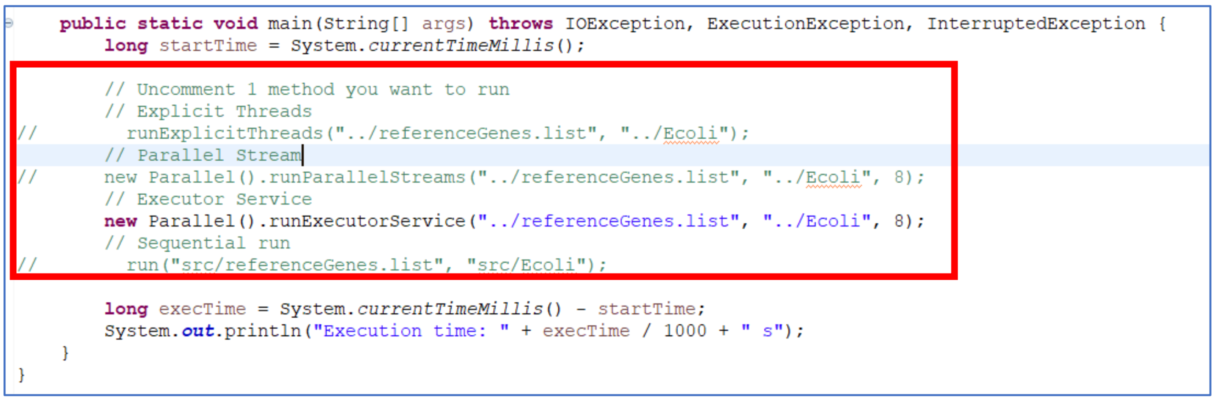
8.1.1 Running the application

Prerequisites:

- Java 11 or higher + JDK installed
- jacobi and junit libraries installed in the workspace

1. Unzip the file and open the folder parallel_final in the IDE of choice.

2. Navigate to the main method (line 299) located in qut→ Parallel.java. Uncomment the version you would like to run.



```
    public static void main(String[] args) throws IOException, ExecutionException, InterruptedException {
        long startTime = System.currentTimeMillis();

        // Uncomment 1 method you want to run
        // Explicit Threads
//          runExplicitThreads("../referenceGenes.list", "../Ecoli");
        // Parallel Stream
//      new Parallel().runParallelStreams("../referenceGenes.list", "../Ecoli", 8);
        // Executor Service
        new Parallel().runExecutorService("../referenceGenes.list", "../Ecoli", 8);
        // Sequential run
//          run("src/referenceGenes.list", "src/Ecoli");

        long execTime = System.currentTimeMillis() - startTime;
        System.out.println("Execution time: " + execTime / 1000 + " s");
    }
}
```

4. You can change the number of threads in the third parameter of the runParallelStream and runExecutorService ranging from 1 to match the maximum number of cores supported by your environment.
5. Compile and run the Parallel.java file
6. To run tests, compile and run the qut → ParallelTest.java file

8.1.2 Parallel.java file

```
package qut;

import jaligner.*;
import jaligner.matrix.*;
import edu.au.jacobi.pattern.*;
import java.io.*;
import java.util.*;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class Parallel {
        private static HashMap<String, Sigma70Consensus> consensus = new
HashMap<>();
//          private static Series sigma70_pattern =
Sigma70Definition.getSeriesAll_Unanchored(0.7);
        private static final ThreadLocal<Series> sigma70_pattern =
ThreadLocal
                    .withInitial(() ->
Sigma70Definition.getSeriesAll_Unanchored(0.7));
```

```
      private static final Matrix BLOSUM_62 = BLOSUM62.Load();
      private static byte[] complement = new byte['z'];

      static {
            complement['C'] = 'G';
            complement['c'] = 'g';
            complement['G'] = 'C';
            complement['g'] = 'c';
            complement['T'] = 'A';
            complement['t'] = 'a';
            complement['A'] = 'T';
            complement['a'] = 't';
      }

      public static HashMap<String, Sigma70Consensus> getConsensus() {
            return consensus;
      }

      private static List<Gene> ParseReferenceGenes(String referenceFile)
throws FileNotFoundException, IOException {
            BufferedReader reader = new BufferedReader(new
InputStreamReader(new FileInputStream(referenceFile)));
            List<Gene> referenceGenes = new ArrayList<Gene>();
            while (true) {
                  String name = reader.readLine();
                  if (name == null)
                        break;
                  String sequence = reader.readLine();
                  referenceGenes.add(new Gene(name, 0, 0, sequence));
                  consensus.put(name, new Sigma70Consensus());
            }
            consensus.put("all", new Sigma70Consensus());
            reader.close();
            return referenceGenes;
      }

      private static boolean Homologous(PeptideSequence A, PeptideSequence
B) {
            return SmithWatermanGotoh.align(new Sequence(A.toString()), new
Sequence(B.toString()), BLOSUM_62, 10f, 0.5f)
                        .calculateScore() >= 60;
      }

      private static NucleotideSequence
GetUpstreamRegion(NucleotideSequence dna, Gene gene) {
            int upStreamDistance = 250;
            if (gene.location < upStreamDistance)
                  upStreamDistance = gene.location - 1;

            if (gene.strand == 1)
                  return new NucleotideSequence(
                              Arrays.copyOfRange(dna.bytes, gene.location -
upStreamDistance - 1, gene.location - 1));
            else {
                  byte[] result = new byte[upStreamDistance];
                  int reverseStart = dna.bytes.length - gene.location +
upStreamDistance;
                  for (int i = 0; i < upStreamDistance; i++)
                        result[i] = complement[dna.bytes[reverseStart -
i]];
                  return new NucleotideSequence(result);
            }
      }

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

      private static void ProcessDir(List<String> list, File dir) {
            if (dir.exists())
                  for (File file : dir.listFiles())
                        if (file.isDirectory())
                              ProcessDir(list, file);
                        else
                              list.add(file.getPath());
      }

      private static List<String> ListGenbankFiles(String dir) {
            List<String> list = new ArrayList<String>();
            ProcessDir(list, new File(dir));
            return list;
      }

      private static GenbankRecord Parse(String file) throws IOException {
            GenbankRecord record = new GenbankRecord();
            BufferedReader reader = new BufferedReader(new
InputStreamReader(new FileInputStream(file)));
            record.Parse(reader);
            reader.close();
            return record;
      }

      public synchronized void addConsensus(String name, Match prediction)
{
            consensus.get(name).addMatch(prediction);
            consensus.get("all").addMatch(prediction);
      }

      /*
       *
===========================================================================
==
       *                                   PARALLEL WITH EXPLICIT THREADS
       *
===========================================================================
==
       *
       */

    private static class FileProcessorTask implements Runnable {
        private String filename;
        private List<Gene> referenceGenes;

        public FileProcessorTask(String filename, List<Gene>
referenceGenes) {
            this.filename = filename;
            this.referenceGenes = referenceGenes;
        }

        @Override
        public void run() {
            try {
                System.out.println(filename);
                GenbankRecord record = Parse(filename);
                for (Gene referenceGene : referenceGenes) {
                    System.out.println(referenceGene.name);
                    for (Gene gene : record.genes) {
                        if (Homologous(gene.sequence,
referenceGene.sequence)) {
                            NucleotideSequence upStreamRegion =
GetUpstreamRegion(record.nucleotides, gene);
                            Match prediction =
PredictPromoter(upStreamRegion);
                            if (prediction != null) {
                                synchronized (consensus) {

consensus.get(referenceGene.name).addMatch(prediction);

consensus.get("all").addMatch(prediction);
                                }
                            }
                        }
                    }
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }

    public static void runExplicitThreads(String referenceFile, String dir)
throws FileNotFoundException, IOException {
        List<Gene> referenceGenes = ParseReferenceGenes(referenceFile);
        List<String> files = ListGenbankFiles(dir);
        List<Thread> threads = new ArrayList<>();

        for (String filename : files) {
            Thread thread = new Thread(new FileProcessorTask(filename,
referenceGenes));
            thread.start();
            threads.add(thread);
        }

        // Wait for all threads to complete
        for (Thread thread : threads) {
            try {
                thread.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

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


      /*
```

```
       *
**==========================================================================
==
       *                                                             PARALLEL
STREAMS
       **
===========================================================================
=
       */

      public void runParallelStreams(String referenceFile, String dir, int
numOfThreads) throws IOException {
            // Initialize a list to hold FileProcessor tasks
            List<FileProcessor> fprocessor = new ArrayList<>();
            // Parse reference genes from the reference file
            List<Gene> referenceGenes = ParseReferenceGenes(referenceFile);
            // Iterate over each reference gene and GenBank file
            for (Gene referenceGene : referenceGenes) {
                  for (String filename : ListGenbankFiles(dir)) {
                        GenbankRecord record = Parse(filename);
                        for (Gene gene : record.genes) {
                              fprocessor.add(new
FileProcessor(referenceGene, gene, record));
                        }
                  }
            }

      System.setProperty("java.util.concurrent.ForkJoinPool.common.parallel
ism", Integer.toString(numOfThreads));
            fprocessor.parallelStream()
                        .filter(task -> Homologous(task.getGene().sequence,
task.getReferenceGene().sequence)).forEach(task -> {
                              NucleotideSequence upStreamRegion =
GetUpstreamRegion(task.getRecord().nucleotides, task.getGene());
                              Match prediction =
PredictPromoter(upStreamRegion);
                              if (prediction != null)

      addConsensus(task.getReferenceGene().name, prediction);
                        });

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

      /*
       *
===========================================================================
==
                                                 EXECUTOR SERVICE CODE
       *
===========================================================================
==
       */

      public void runExecutorService(String referenceFile, String dir, int
threadNum)
                  throws IOException, ExecutionException,
InterruptedException {
```

```
            // ExecutorService executorService =
            //
Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());
            ExecutorService executorService =
Executors.newFixedThreadPool(threadNum);
            List<Gene> referenceGenes = ParseReferenceGenes(referenceFile);
            List<Future> futureTasks = new ArrayList<>();

            for (Gene referenceGene : referenceGenes) {
                  for (String filename : ListGenbankFiles(dir)) {
                        GenbankRecord record = Parse(filename);
                        for (Gene gene : record.genes) {
                              Future futureTask =
executorService.submit(new RunnableTask(referenceGene, gene, record));
                              futureTasks.add(futureTask);
                        }
                  }
            }


            executorService.shutdown();
            for (Future futureTask : futureTasks)
                  futureTask.get();
            for (Map.Entry<String, Sigma70Consensus> entry :
consensus.entrySet())
                  System.out.println(entry.getKey() + " " +
entry.getValue());
      }

      public class RunnableTask implements Runnable {
            private final Gene referenceGene;
            private final Gene gene;
            private final GenbankRecord record;

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

            @Override
            public void run() {
                  if (Homologous(gene.sequence, referenceGene.sequence)) {
                        NucleotideSequence upStreamRegion =
GetUpstreamRegion(record.nucleotides, gene);
                        Match prediction = PredictPromoter(upStreamRegion);
                        if (prediction != null) {
                              addConsensus(referenceGene.name, prediction);
                        }
                  }
            }
      }

      /*
       *
===========================================================================
==
                                                                  SEQUENTIAL
```

```
         *
===========================================================================
==
         */

	public static void run(String referenceFile, String dir) throws
IOException {
		List<Gene> referenceGenes = ParseReferenceGenes(referenceFile);
		// Loop Transformation: for loop 1 <--> for loop 2
		for (Gene referenceGene : referenceGenes) {
			System.out.println(referenceGene.name);
			for (String filename : ListGenbankFiles(dir)) {
				System.out.println(filename);
				GenbankRecord record = null;
				try {
					record = Parse(filename);
				} catch (IOException e) {
					e.printStackTrace();
				}
				for (Gene gene : record.genes) {
					if (Homologous(gene.sequence,
referenceGene.sequence)) {
						NucleotideSequence upStreamRegion =
GetUpstreamRegion(record.nucleotides, gene);
						Match prediction =
PredictPromoter(upStreamRegion);
						if (prediction != null) {

	consensus.get(referenceGene.name).addMatch(prediction);

	consensus.get("all").addMatch(prediction);
						}
					}
				}
			}
		}
	}

	public static void main(String[] args) throws IOException,
ExecutionException, InterruptedException {
		long startTime = System.currentTimeMillis();

		// Uncomment 1 method you want to run
		// Explicit Threads
//		runExplicitThreads("../referenceGenes.list", "../Ecoli");
		// Parallel Stream
//		new Parallel().runParallelStreams("../referenceGenes.list",
"../Ecoli", 8);
		// Executor Service
		new Parallel().runExecutorService("../referenceGenes.list",
"../Ecoli", 8);
		// Sequential run
//		run("src/referenceGenes.list", "src/Ecoli");

		long execTime = System.currentTimeMillis() - startTime;
		System.out.println("Execution time: " + execTime / 1000 + "
s");
	}
}
```

8.1.3 Test Script

```
package qut;

import static org.junit.jupiter.api.Assertions.assertEquals;

import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.ExecutionException;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

public class ParallelTest {

    private Parallel parallel;

    @BeforeEach
    public void setUp() {
        parallel = new Parallel();
    }

    @Test
    public void testParallelStreams() throws IOException {
        // Declare default consensus results
        HashMap<String, String> sequentialResult = new HashMap<>();
        sequentialResult.put("all", " Consensus: -35: T T G A C A gap: 17.6
-10: T A T A A T  (5430 matches)");
        sequentialResult.put("fixB", " Consensus: -35: T T G A C A gap:
17.7 -10: T A T A A T  (965 matches)");
        sequentialResult.put("carA", " Consensus: -35: T T G A C A gap:
17.7 -10: T A T A A T  (1079 matches)");
        sequentialResult.put("fixA", " Consensus: -35: T T G A C A gap:
17.6 -10: T A T A A T  (896 matches)");
        sequentialResult.put("caiF", " Consensus: -35: T T C A A A gap:
18.0 -10: T A T A A T  (11 matches)");
        sequentialResult.put("caiD", " Consensus: -35: T T G A C A gap:
17.6 -10: T A T A A T  (550 matches)");
        sequentialResult.put("yaaY", " Consensus: -35: T T G T C G gap:
18.0 -10: T A T A C T  (4 matches)");
        sequentialResult.put("nhaA", " Consensus: -35: T T G A C A gap:
17.6 -10: T A T A A T  (1879 matches)");
        sequentialResult.put("folA", " Consensus: -35: T T G A C A gap:
17.5 -10: T A T A A T  (46 matches)");

        // Run the original Sequential version
        parallel.run("../referenceGenes.list", "../Ecoli");

        // Run the Parallel version with the same input dataset
        parallel.runParallelStreams("../referenceGenes.list",   "../Ecoli",
8);

        // Get the consensus results
        HashMap<String,     Sigma70Consensus>     parallelConsensus     =
parallel.getConsensus();

        // Compare the consensus results
        compareConsensusResults(sequentialResult, parallelConsensus);
    }
    @Test
    public void testExplicitThreads() throws IOException {
        // Declare default consensus results
        HashMap<String, String> sequentialResult = new HashMap<>();
        sequentialResult.put("all", " Consensus: -35: T T G A C A gap: 17.6
-10: T A T A A T  (5430 matches)");
        sequentialResult.put("fixB", " Consensus: -35: T T G A C A gap:
17.7 -10: T A T A A T  (965 matches)");
        sequentialResult.put("carA", " Consensus: -35: T T G A C A gap:
17.7 -10: T A T A A T  (1079 matches)");
        sequentialResult.put("fixA", " Consensus: -35: T T G A C A gap:
17.6 -10: T A T A A T  (896 matches)");
        sequentialResult.put("caiF", " Consensus: -35: T T C A A A gap:
18.0 -10: T A T A A T  (11 matches)");
        sequentialResult.put("caiD", " Consensus: -35: T T G A C A gap:
17.6 -10: T A T A A T  (550 matches)");
        sequentialResult.put("yaaY", " Consensus: -35: T T G T C G gap:
18.0 -10: T A T A C T  (4 matches)");
        sequentialResult.put("nhaA", " Consensus: -35: T T G A C A gap:
17.6 -10: T A T A A T  (1879 matches)");
        sequentialResult.put("folA", " Consensus: -35: T T G A C A gap:
17.5 -10: T A T A A T  (46 matches)");

        // Run the Parallel version with explicit threads
        parallel.runExplicitThreads("../referenceGenes.list", "../Ecoli");

        // Get the consensus results
        HashMap<String, Sigma70Consensus> parallelConsensus =
parallel.getConsensus();

        // Compare the consensus results
        compareConsensusResults(sequentialResult, parallelConsensus);
    }

    @Test
    public void testExecutorService() throws IOException,
ExecutionException, InterruptedException {
        // Declare default consensus results
        HashMap<String, String> sequentialResult = new HashMap<>();
        sequentialResult.put("all", " Consensus: -35: T T G A C A gap: 17.6
-10: T A T A A T  (5430 matches)");
        sequentialResult.put("fixB", " Consensus: -35: T T G A C A gap:
17.7 -10: T A T A A T  (965 matches)");
        sequentialResult.put("carA", " Consensus: -35: T T G A C A gap:
17.7 -10: T A T A A T  (1079 matches)");
        sequentialResult.put("fixA", " Consensus: -35: T T G A C A gap:
17.6 -10: T A T A A T  (896 matches)");
        sequentialResult.put("caiF", " Consensus: -35: T T C A A A gap:
18.0 -10: T A T A A T  (11 matches)");
        sequentialResult.put("caiD", " Consensus: -35: T T G A C A gap:
17.6 -10: T A T A A T  (550 matches)");
        sequentialResult.put("yaaY", " Consensus: -35: T T G T C G gap:
18.0 -10: T A T A C T  (4 matches)");
        sequentialResult.put("nhaA", " Consensus: -35: T T G A C A gap:
17.6 -10: T A T A A T  (1879 matches)");
        sequentialResult.put("folA", " Consensus: -35: T T G A C A gap:
17.5 -10: T A T A A T  (46 matches)");

        // Run the Parallel version with ExecutorService
        parallel.runExecutorService("../referenceGenes.list", "../Ecoli",
8);
```

```
        // Get the consensus results
        HashMap<String,      Sigma70Consensus>      parallelConsensus      =
parallel.getConsensus();

        // Compare the consensus results
        compareConsensusResults(sequentialResult, parallelConsensus);
    }

    // Compare the consensus results with the default results
    public     void     compareConsensusResults(HashMap<String,     String>
sequentialResult,
                                                  HashMap<String,    Sigma70Consensus>
parallelConsensus) {
        // Compare each consensus result
        for (Map.Entry<String, String> entry : sequentialResult.entrySet())
{
            String geneName = entry.getKey();
            String defaultResult = entry.getValue();
            String                     parallelResult                     =
parallelConsensus.get(geneName).toString();

            assertEquals(defaultResult,  parallelResult,  "Consensus  result
mismatch for " + geneName);
        }
    }
}
```

We've processed your Assignment extension request FORM-AEX-387859

no-reply@qut.edu.au <no-reply@qut.edu.au>

Tue 10/31/2023 10:26 AM

To:Anjana Ranjan <anjana.ranjan@connect.qut.edu.au>

Hi Anjana,

Thank you for your assignment extension request.

We have **approved your request (FORM-AEX-387859).** The due date for your assignment **Assignment 2 - Parallelization Project**, for unit CAB401 has been extended **until 01/11/2023 11.59pm AEST.**

Your request has been approved in this instance, however, for future applications, please ensure that your supporting documentation meet the guidelines listed on the Special Circumstances page and Late Assignments and Extensions page on HiQ. Medical certificates should cover the due date and/or 48-hour late submission period. This certificate will be accepted as it is dated within the late submission period. However, future applications with similar documentation may not be accepted.

When you submit your assignment, add a comment to advise that you have an approved assignment extension and attach a copy of this email. See our instructions on submitting your assignment.

Be aware that a copy of this email is kept on file. You should not alter this email in any way. Email notifications that have been altered or differ in any way from the original may result in an allegation of student misconduct as set out in the Student Code of Conduct.

Need extra support? You can access free, confidential counselling with qualified professionals. We also offer planning and support if you have a disability, injury or health condition that affects your ability to study. If you are struggling to meet your university commitments, academic support is also available.

Have a question? Our advice on late assignments and extensions covers answers to common queries. You can also contact us by email, phone, on campus or via online chat. Visit the HiQ website for our contact details and opening hours.

Email us Phone us

About HiQ