## Sequential Program Explanation

### What It Does

In essence, this program predicts the classification of a genome based on its consensus promoter properties. Two inputs are received by the program: a reference file, and the file path containing all genomes to be predicted. For prediction, a comparison is made between the genome and each genome in the reference file. If the two are found to be homologous – in the same gene family – the upstream promotor section is obtained and used for prediction.

### How It Works

To perform the predictions, the program follows the process: for each combination of reference gene and prediction gene, decide if the two are homologous (using the Smith-Waterman-Gotoh algorithm), get the relevant upstream region, and subsequently predict its category.

1. **for** referenceGene in referenceFiles:
2. **for** gene in predictionGenes:
3. **if** Homologous(referenceGene, gene):
4. upStreamRegion = GetUpstreamRegion(nucleotides, gene);
5. prediction = PredictPromoter(upStreamRegion);
6. consensus.addPrediction(prediction);

**Figure 1:** *Pseudo-code.* This figure shows pseudo-code describing the implementation of this Java program, used to predict the classification of genomes. Full code in appendix 1.

To determine whether or not the genomes are homologous, the Smith-Waterman-Gotoh algorithm is called. This algorithm evaluates the difference between the genomes (represented in a matrix) and subsequently compares them to a scoring matrix. The reference genome with the highest score is recorded as the closest match, providing a likely match is found (Gotoh, 1982).

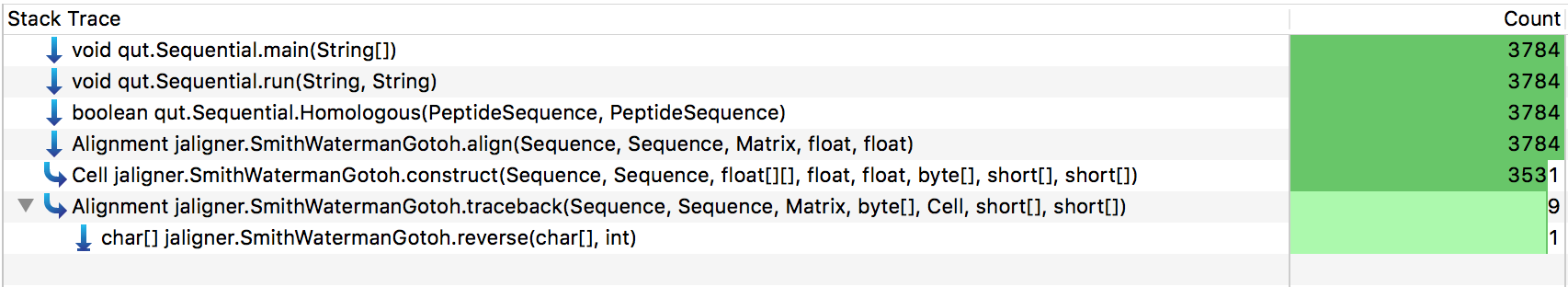
If the genomes are homologous, the upstream promotor region of the genome to be predicted is retrieved. The prediction then occurs on this region and the best-match is returned (null if no best-match is found).

## Analysis of Potential Parallelism

Using the Java Flight Recorder tool, method profiling was conducted on the sequential program. The generated report made recommendations in relation to parallelisation and method optimisation.

As suspected due to the nature of this program, the report found that only 1 thread was utilised during the run – 4 methods were sampled during this time. The machine used to run the program has 8 cores and can therefore manage up to 8 threads.

It was reported that the method *SmithWatermanGotoh.construct* was the most sampled method in the program; this means that it was the method that ran for the largest percentage of time during execution.



**Figure 2.** *Stack trace of expensive method.* A detailed stack trace of the *SmithWatermanGotoh* method; identified to the be most computationally expensive method in the program. High level break down of program method profile may be found in appendix 2.

From the stack trace drill-down shown in Figure 2, it can be confirmed that the majority of runtime in this program can be attributed to the *SmithWatermanGotoh.contruct* method. This method will subsequently be the focus of parallelisation.

## Architecture of Program Parallelisation

## Timing and Profiling

## Software Used

## Performance Barriers Encountered

## Explanation of Code

## Reflection

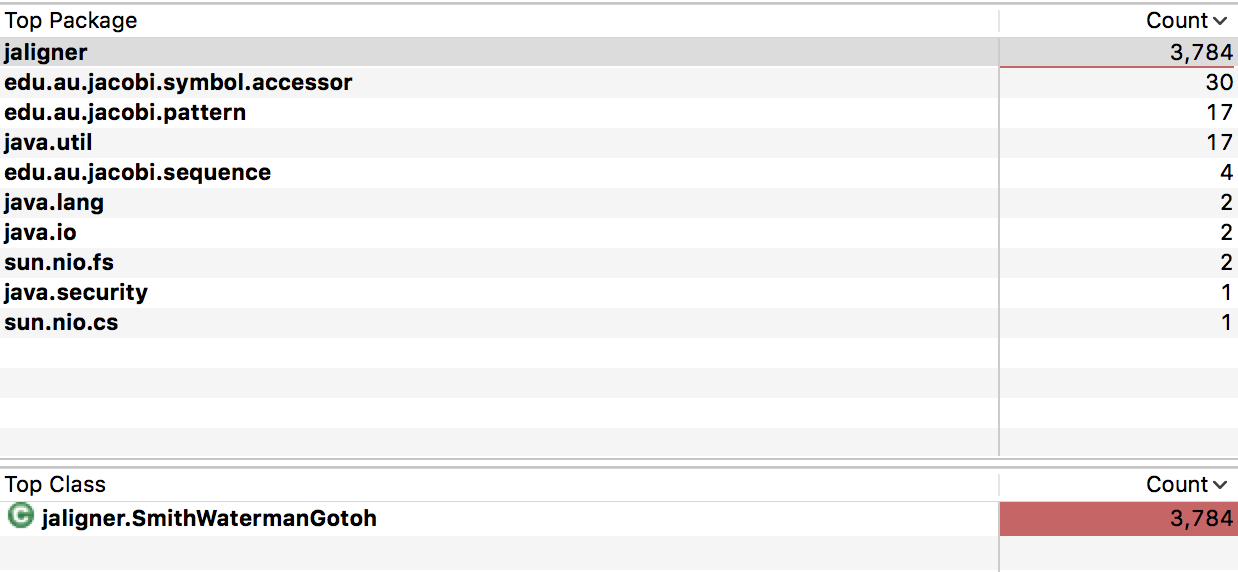
# 5.0 Appendix

## 5.1 Appendix 1 – Implementation of Java Program

1. **public** **static** **void** run(String referenceFile, String dir) **throws** FileNotFoundException, IOException
2. {
3. List<Gene> referenceGenes = ParseReferenceGenes(referenceFile);
5. **for** (String filename : ListGenbankFiles(dir))
6. {
8. System.out.println(filename);
9. GenbankRecord record = Parse(filename);
10. **for** (Gene referenceGene : referenceGenes)
11. {
12. stem.out.println(referenceGene.name);
14. **for** (Gene gene : record.genes)
15. **if** (Homologous(gene.sequence, referenceGene.sequence))
16. {
18. NucleotideSequence upStreamRegion = GetUpstreamRegion(record.nucleotides, gene);
19. Match prediction = PredictPromoter(upStreamRegion);
20. **if** (prediction != **null**)
21. {
22. consensus.get(referenceGene.name).addMatch(prediction);
23. consensus.get("all").addMatch(prediction);
24. }
25. }
26. }
27. }
29. **for** (Map.Entry<String, Sigma70Consensus> entry : consensus.entrySet())
30. System.out.println(entry.getKey() + " " + entry.getValue());
31. }

This code excerpt displays the implementation of the promoter prediction program in Java.

## 5.2 Appendix 2 – Method Profiling of Sequential Program



The large majority of run time was spent in the *SmithWatermanGotoh* class.