**1 Introduction**

The application chosen to be parallelised was the Bioinformatics – Genome Similarity Using Frequency Vectors. This application’s purpose is to compare different types of bacteria to each other by calculating the correlation between two different bacteria.

**1.1 Software Architecture**

Figure 1 below shows the basic process of Bioinformatics – Genome Similarity Using Frequency Vectors’ architecture. The detailed software architecture can be found in full in section 6.3 of the Appendix of this report.

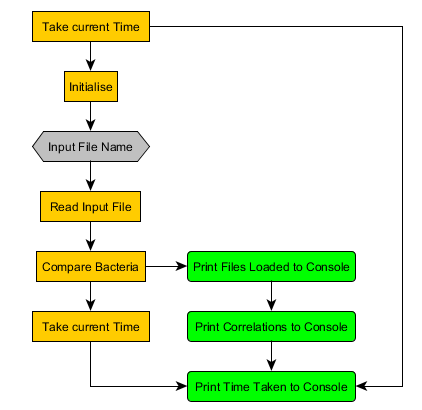


Figure 1: Basic View of Software Architecture

**1.1.1 Time**

The application takes the time before and after the application and the difference is calculated. This is printed to the console to notify the user how long the program took to complete the Bioinformatics – Genome Similarity Using Frequency Vectors application.

**1.1.2 Initialise**

This method initialises long variables M1 and M, based on the predefined value for AA\_NUMBER and LEN, for the class Bacteria.

**1.1.3 Input Filename and Read Input File**

The program takes a users’ input argument. This argument is the filename for a text file. This text file contains the number of bacteria and the names of the associated bacteria files. The first line contains the number of bacteria. From the second line down, these lines contain the name of each \*.faa file. The extension \*.faa is used by the NCBI for FASTA animo acids. In this application, forty-one protein FASTA files are being used. Figure 2, on the next page, shows the detail view software architecture for this function.

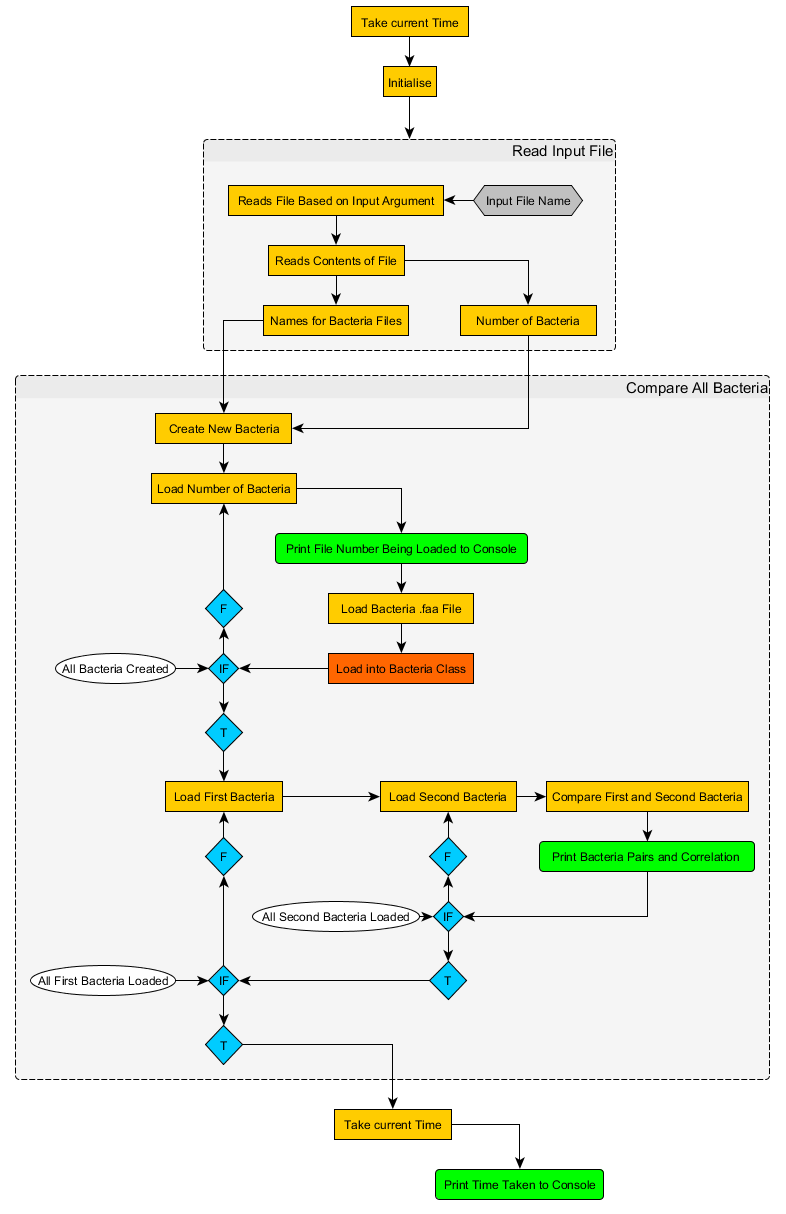


Figure 2: Read Input File Software Architecture

**1.1.4 Compare All Bacteria**

This component of the application is where most of the runtime and computational work in located. This function does three things. The first creates the number of classes of bacteria based on the number of bacteria to be read. The second component is made of two parts. The first part is printing to the console which file is being loaded. The second part is loading the \*.faa files and loading the data into their respective bacteria class. The third calls for the compare bacteria function which calculates the correlation between two pairs of different bacteria and then printing the result to the console. Figure 3 on the next page shows a detailed view of the CompareAllBacteria function.

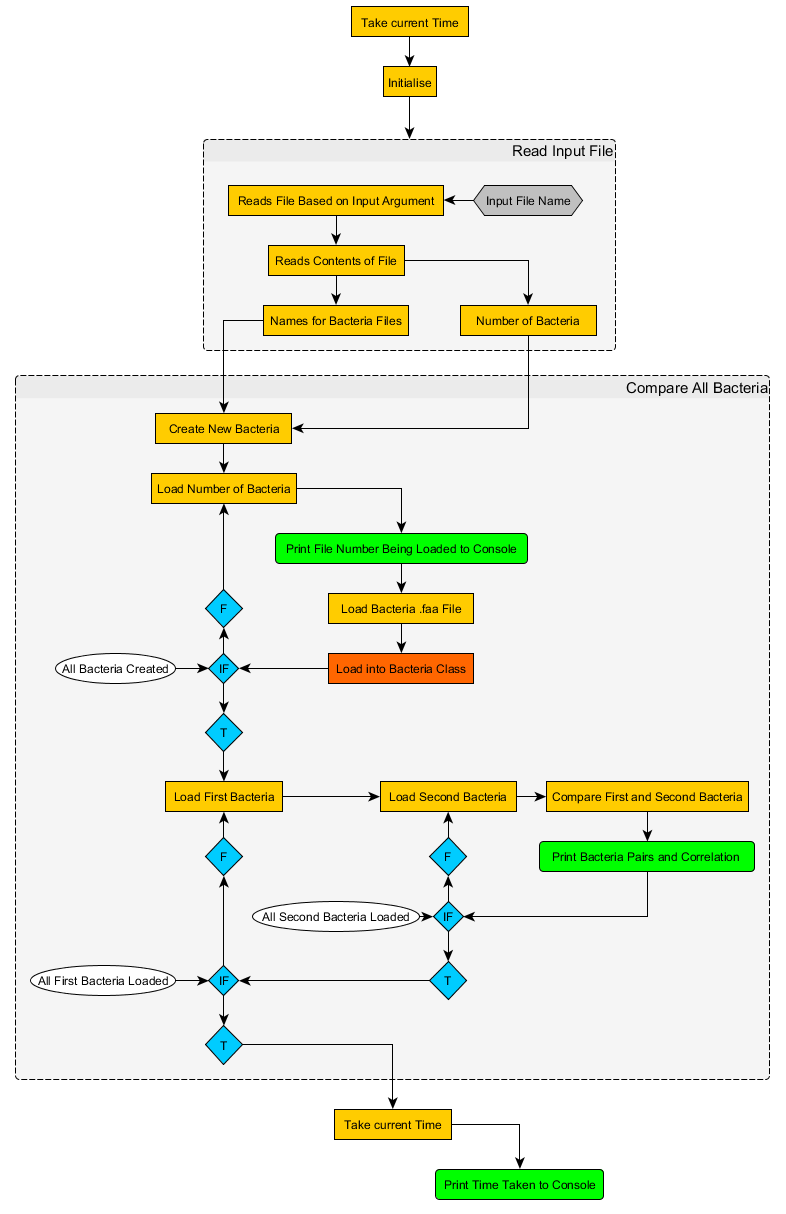


Figure 3: Compare All Bacteria Software Architecture

**1.1.5 Bacteria Class**

The bacteria files are opened and vectors are initialised. The bacteria class holds data based on the contents in the \*.faa files. The class obtains the necessary characters from the respective \*.faa file. Figure 4, shown below, shows some of the contents of file AcMNPV.faa.

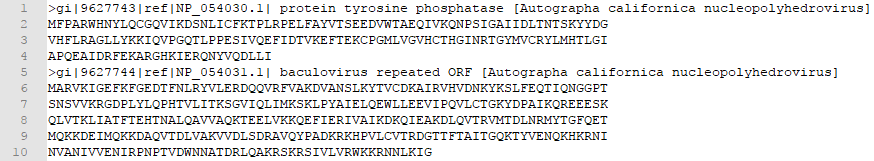


Figure 4: AcMNPV Animo Acid File Content Example

The first stage of the bacteria class is obtaining the data in the respective \*.faa file. In this process there are three parts. The first part is whether the end of the file has been met. If so, the search to get the characters for the bacteria class ends. If not, then the second part comes into effect. If the character identified is a ‘>’, then that line is skipped. This line is skipped because the line does not contain the data required, rather just the name and reference of the amino acid. This can be seen in lines one and five in Figure 4 above. When the line is skipped the characters are read initialise buffer is called. The last part is whether a ‘\n’, or space, character is identified. If not, the process continues to get a character. If a ‘\n’ is identified, then continue buffer is called. This section of the bacteria class can be seen below in Figure 5

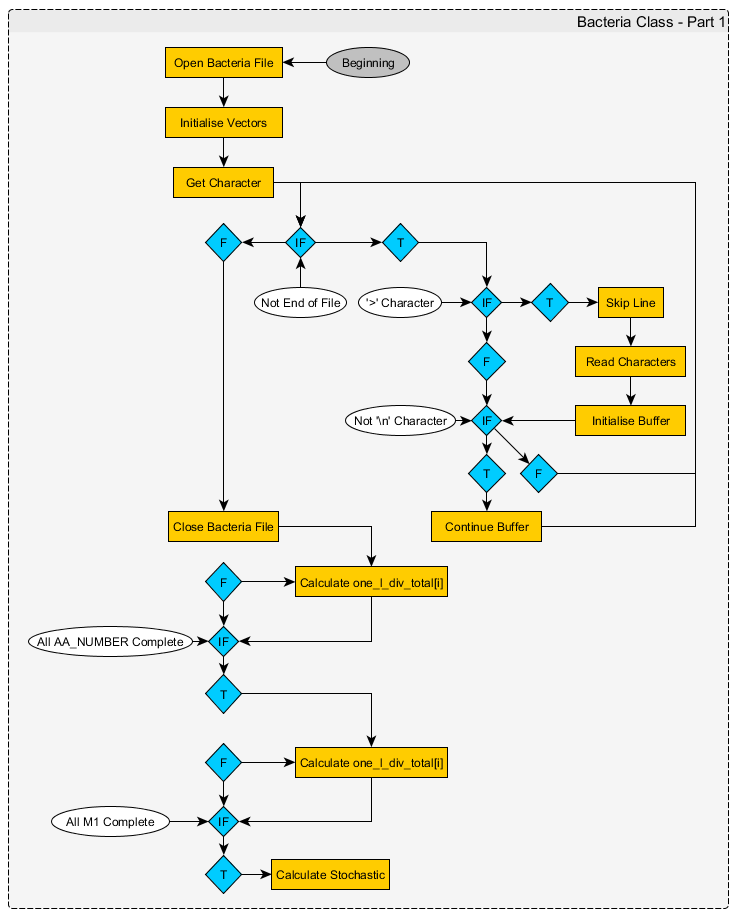


Figure 5: Get Data from \*.faa File

When the end of the file has been confirmed the respective file is closed. From here the calculations are performed. Figure 6, shown below, shows the software architecture for this stage up to the point where stochastic is calculated.

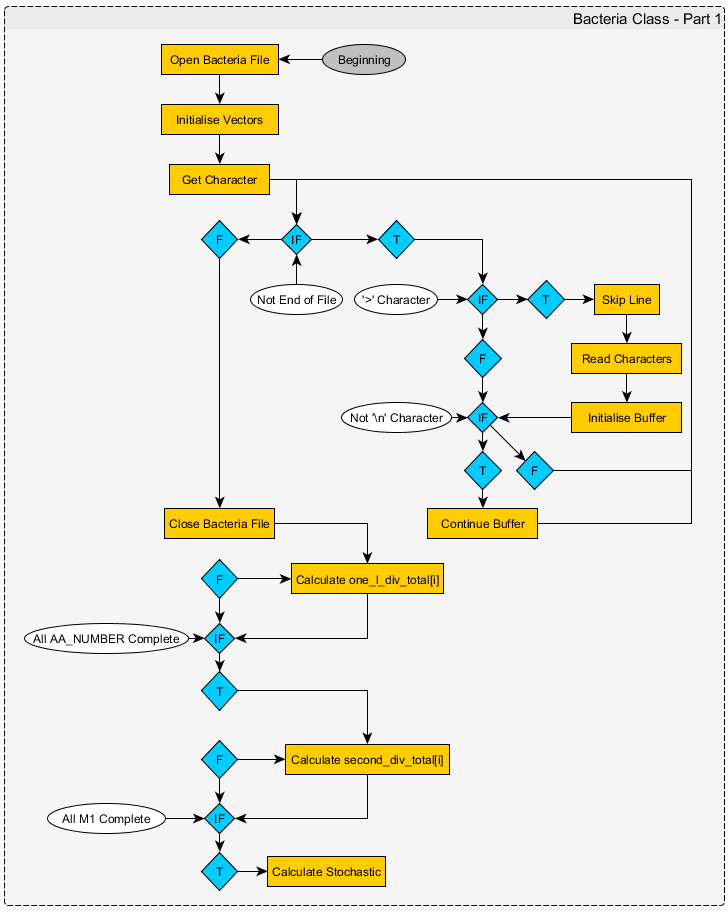


Figure 6: Calculate Variables in Bacteria Class – Part 1

The bacteria class proceeds to calculate other variables which can be seen below in Figure 7.

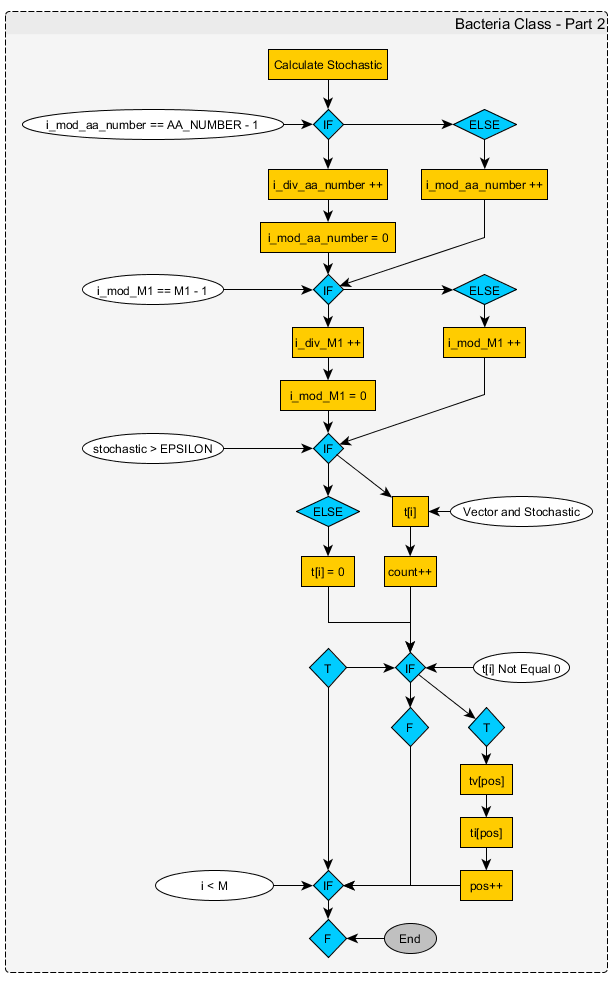


Figure 7: Other Values Calculated in Bacteria Class

The main importance of the bacteria class is the calculated values for tv[] and ti[]. These variables are used when comparing two different bacteria. Figure 8 below shows the software architecture for the rest of the bacteria class. The full software architecture diagram for the bacteria class can be found in the appendices. Bacteria Class – Part 1 software architecture can be found in section 6.4 of this report. Bacteria Class – Part 2 software architecture can be found in section 6.5 of this report.

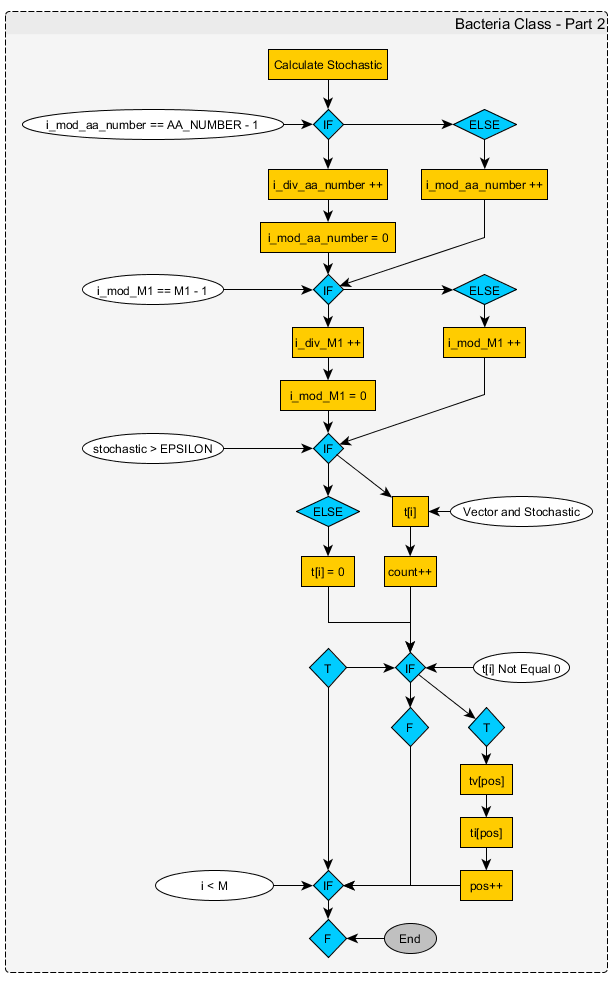


Figure 8: Bacteria Class Ending

**1.1.6 Compare Bacteria**

The compare bacteria function takes two bacteria as inputs and proceeds to calculate the correlation between the two bacteria based on their vectors, ti and tv variables. In this application, the two bacteria that are taken are two different bacteria. This occurs in the CompareAllBacteria function which includes a nested loop to ensure that every possible unique pair of bacteria are compared. The detail software architecture for the CompareBacteria function can be seen in Figure 9 on the next page and in section 6.6 of the appendices in this report.

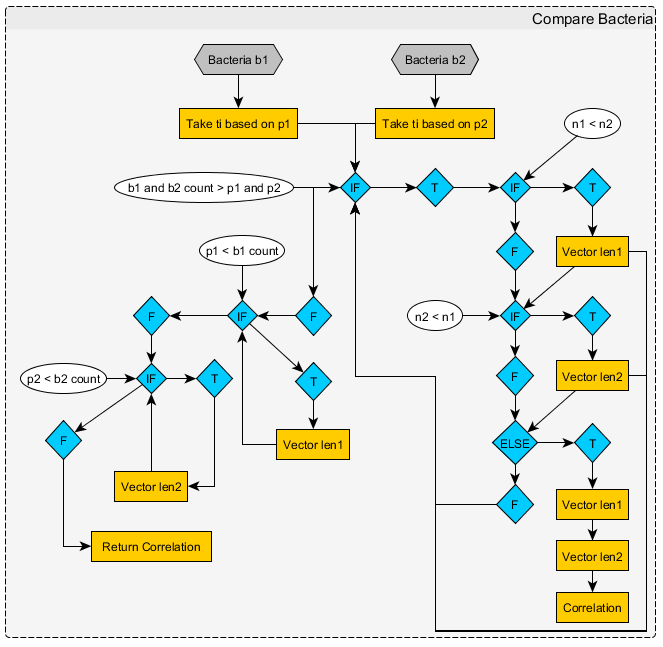


Figure 9: Detail View of Compare Bacteria

**2 Bioinformatics – Genome Similarity Using Frequency Vectors**

This section of the report discusses what parts of the original code that were analysed. This includes sections of code that resulted in success and failure to be parallelised and what parts of the application can be scalable. This section also discusses the software used and techniques during this analysis.

**2.1 Software, Compiler, Hardware and Techniques**

**2.1.1 Software**

The integrated development environment (IDE) used when analysing the application and its code was Microsoft’s Visual Studio 2017. This IDE’s diagnostic tools were used to identify which section of code took the most amount of time to process. Furthermore, the diagnostic tools show the memory and CPU usage that is being used by the application is real-time. This assisted in analysing for-loop parallelisation as to how much memory was being used and the difference between the original and parallelised version of code’s CPU usage.

OpenMP 2.0 is included as part of Visual Studio 2017. This was enabled and chosen for the following reasons. Its simplicity, ease of use and integration in the IDE makes it a wonderful choice when parallelising for-loops and multi-threading other components of the application. The second reason is that I have not used the OpenMP API before. The third reason, OpenMP performed faster results than using POSIX threads. Unfortunately, OpenMP does not perform while-loop parallelisation. While-loops can be parallelised using OpenMP tasks, but this was found to provide no further performance gain and all while loops did not benefit from this.

**2.1.2 Compiler**

The compiler used was the default compiler that Microsoft has included in Visual Studio 2017. This version of the compiler was Visual C++ 14.1.

**2.1.3 Hardware**

In this report two machines were used. Both machines use the Windows 10 OS and are running on High Performance mode.

The machine used, named miniTOP, has an i7-4710MQ Intel processor operating at 2.50GHz to 3.50GHz with 16.0 GB of DDR3 RAM. This CPU has four physical cores and eight threads. This machine does not overclock the CPU. This machine’s RAM clock rate is at 1600 MHz.

**2.1.4 Techniques**

OpenMP’s sections, parallel and for techniques were used make the program complete within a faster time frame during testing. Sections to perform two different tasks at the same time or splitting sections of work. Parallel and for were used to parallelise for-loops. Other techniques attempted were removing a nested for-loop for a while loop. Utilising an array to store the correlation variables before printing to the console in order.

**2.2 Analysis of Code and Parallelisation Process**

**2.2.1 Successful Parallelisation**

Most of the processing time and memory being used in this application is when the CompareAllBacteria function is called from main. This function contains one for-loop and a nested for-loop. These for-loops call the bacteria class to be filled with \*.faa file data and compare bacteria calculates the correlation of the two bacteria. This can be seen in Figure 10 below.

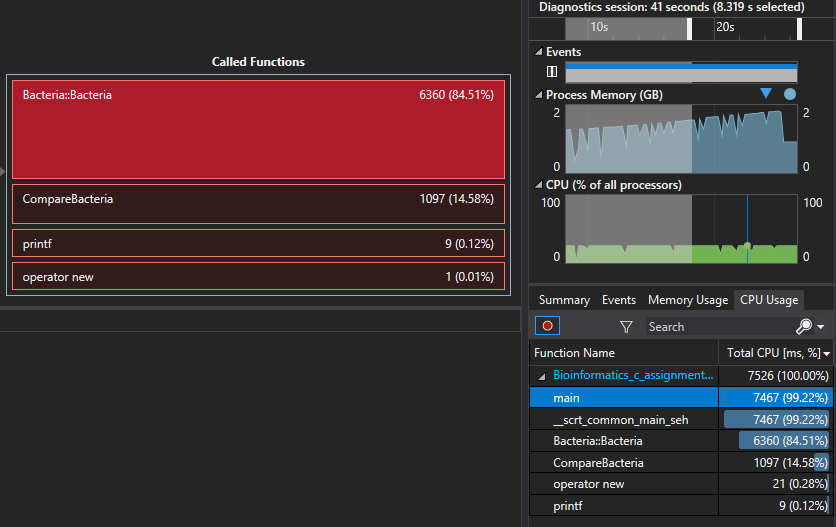


Figure 10: Total CPU (ms, %) of Functions in Bioinformatics – Genome Similarity Using Frequency Vectors Application

The largest components are the bacteria class when files are being loaded and when two bacteria are being compared.

The first for-loop loads each \*.faa file into its respective bacteria class. The original code loads each of these files sequentially, one at a time, when there are forty-one files to be loaded. This also fills data into each bacteria class sequentially as well for the respective bacteria file loaded. This for-loop could be parallelised to load multiple bacteria files on numerous threads. This section of code can be found in section 6.1 of the appendix, lines 239 to 243.

The nested for-loop inputs which two bacteria should be compared. This then calculates the correlation between two different bacteria. This nested loop then prints the result of the two bacteria to the console. This nested-loop sequentially calculates and prints each result to the console. This nested-for loop could be parallelised by calculating the correlation of two different bacteria over numerous threads and having them print to the console. However, this could be printed out of order which would lead to a disorganised console output. This section of code can be found in section 6.1 of the appendix, lines 245 to 252.

A solution to keep order but keeping parallelisation would be to load the correlated results into an array and then use the array to print the correlated results in sequential order. This will require changing of the code.

These for-loops should be scalable based on the number of bacteria to be loaded and compared. This will be shown further in section 4 of this report.

**2.2.2 Further Analysed Parallelisation Attempts**

While most of the run-time is performed in the CompareAllBacteria function, there are other for-loops, while-loops and other sections of code that were analysed to see if there would be any performance increase. However, this was not the case for the following.

**2.2.2.1 Two Functions at the Same Time in Main**

In the main function of the application, there are two functions that are called. Both functions were found to have a low, if negligible, computational time. However, the ReadInputFile function does not have any variables that are dependant of the previous called Init function. It would be interesting to see if there is a time reduction when running both functions at the same time on two different threads instead of the sequential process of Init being called then ReadInputData. However, this did not provide any faster computation. This is because the code in Init and ReadInputFile proved to have insignificant computation time.

**2.2.2.2 Parallelising Bacteria Class**

In the bacteria class there are four for-loops. The first for-loop, at lines 108 to 109 at section 6.1 of this report, parallelisation resulted with no additional performance gain. The second for-loop, at lines 112 to 113 at section 6.1 of this report, parallelisation resulted with no additional performance gain.

The third for-loop, at lines 117 to 118 at section 6.1 of this report, OpenMP’s parallel and for techniques resulted in different outcomes. The parallel for (combined) parallelisation technique resulted in a memory access violation for the variable p2 calculation. The for parallelisation technique resulted in an infinite loop where nothing is being processed according to Visual Studio’s diagnostic tools. The parallel parallelisation technique resulted the application printing the correct results to the console, but at no reduce time.

The fourth for-loop, at lines 158 to 166 at section 6.1 of this report, also had different outcomes when using the OpenMP for-loop parallelisation techniques. Parallel and for combined resulted with incorrect correlation values being printed to the console. The for technique caused a never-ending loop where the processor did no computational work. The parallel technique resulted with a working application printing the correct results, but this did not lower the time spent.

After further analysis with the calculations and variable dependency in the bacteria class, OpenMP sections were utilised to split up the work load over numerous threads. However, this continued to cause a memory access violation at locations at the second for-loop, lines 112 to 113 section 6.1 of this report. This resulted in accepting that the bacteria class could not have its work load split into pieces to reduce to time taken for the application.

**2.2.2.3 Parallelising Comparing Two Bacteria**

In the CompareBacteria function there is a while-loop at lines 195 to 219 in section 6.1 in the appendices of this report. This while-loop was tested with OpenMP’s parallel technique and resulted in a worse performance time for the application. OpenMP’s parallel technique was also used on the while-loops from lines 220 to 231 in section 6.1 of this report. The result had no changes to the time taken for the application. These while loops were then split into OpenMP sections as they do not have any related variable or data dependencies with the exception to the variable count. This resulted in the application to crash while running the Debug model. No warnings displayed, or exceptions caught. In the Release model, an infinite loop occurs.

**3 Code Implemented**

This section of the report includes the code from the original Bioinformatics – Genome Similarity Using Frequency Vectors application in comparison to the final parallelised version. These code changes will start from the top most changes down to the bottom of the code.

**3.1 Code Changes in Compare All Bacteria Function**

The original version of code for the CompareAllBacteria function can be seen below in Figure 11. Below Figure 11, Figure 12 can be seen which includes the code in the high-performance version.

236 void CompareAllBacteria()

237 {

238 Bacteria\*\* b = new Bacteria\*[number\_bacteria];

239 for (int i = 0; i<number\_bacteria; i++)

240 {

241 printf("load %d of %d\n", i + 1, number\_bacteria);

242 b[i] = new Bacteria(bacteria\_name[i]);

243 }

244

245 for (int i = 0; i<number\_bacteria - 1; i++)

246 for (int j = i + 1; j<number\_bacteria; j++)

247 {

248 printf("%2d %2d -> ", i, j);

249 double correlation = CompareBacteria(b[i], b[j]);

250 printf("%.20lf\n", correlation);

251 }

252 }

Figure 11: Original Code for Compare All Bacteria Function

280 void CompareAllBacteria() {

281 Bacteria\*\* b = new Bacteria\*[number\_bacteria];

282 /\*

283 [8] Success with major improvement

288 \*/

289 #pragma omp parallel for

290 for (int i = 0; i < number\_bacteria; i++) {

291 printf("load %d of %d\n", i + 1, number\_bacteria);

292 b[i] = new Bacteria(bacteria\_name[i]);

293 }

294

295 /\*

296 [9]

336 \*/

337 #pragma omp parallel for

338 for (int i = 0; i < number\_bacteria - 1; i++) {

339 /\*

340 [10]

341 \*/

342 for (int j = i + 1; j < number\_bacteria; j++) {

343 double correlation = CompareBacteria(b[i], b[j]);

344 printf("%2d %2d -> %.20lf\n", i, j, correlation);

345 }

346 }

347 }

Figure 12: Parallelised Code for Compare All Bacteria Function

The complete version of the code for the original version of the application can be found in section 6.1 of the appendices. The complete version of the code for the parallelised version of the application can be found in section 6.2 of the appendices. In Figure 12, there are missing lines of code. These lines were removed as they are commented notes taken during the analysis and parallelisation process.

The lines of code in Figure 12, lines 289 and 337, are the OpenMP parallel and for functionality that OpenMP can perform. OpenMP parallel spawns a group of threads. OpenMP for divides the loop iterations between the spawned threads. Both the ‘parallel’ and ‘for’ techniques offered by OpenMP were combined. Through testing they provided the quickest time to achieving that all the \*.faa files were loaded, all respective data was used to create each respective bacteria class, all bacteria comparisons were calculated, and those results are printed to the console.

Below in Figures 13 and 14 it can be seen what happens in the first second of the two versions of the application when it starts. In Figure 13, the first file is being loaded. In Figure 14, four bacteria \*.faa files are being read and those classes are being created for bacteria one, twelve, thirty-two and twenty-two. This is because the test was running on a four core, four read capable CPU so only a maximum of four files can be read at any one time when using this machine. However, on a four core, eight thread capable CPU, a maximum of eight files can be read. This can be seen in Figure 15 on the next page.

|  |  |
| --- | --- |
| Figure 13: Original Loading Bacteria | Figure 14: Parallel Loading Bacteria Four Threads Max |
| D:\Downloads\22850072_10211238801179946_197361970_o.png  Figure 15: Parallel Loading Bacteria Eight Threads Max | |

Using OpenMP’s parallel and for techniques for the for-loop at lines 289 to 293, shown at Figure 12, multiple files can be read, and bacteria classes created at the same time. The obvious limitation here is the read speed of the disc that the files are stored on. However, the use of OpenMP allows for the application to be scalable. This scalability is dependent of the maximum number of threads of a CPU, the clock rate of the CPU, enough memory and the number of bacteria being compared. Below, in Figures 16 and 17, the amount of memory being used increases with each bacteria file being read and class created.

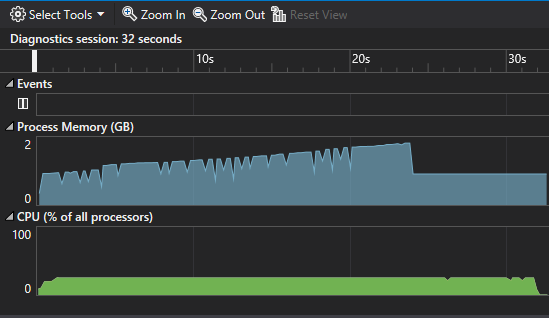


Figure 16: Sequential Application Memory and CPU Usage

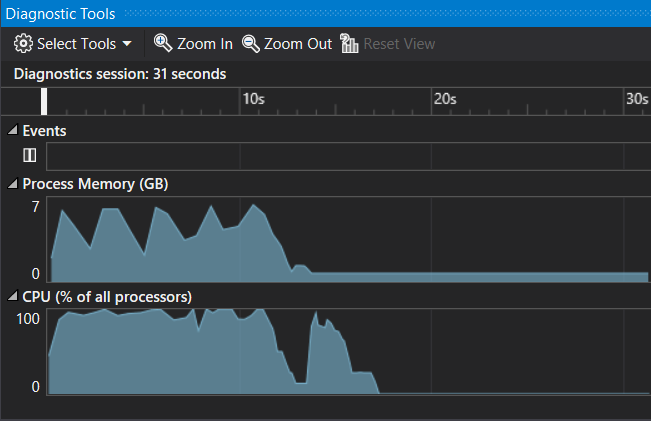


Figure 17: Parallelised Application Memory and CPU Usage

For the parallelisation of the nested for-loop, from lines 337 to 346 in Figure 12, the correlation printed will be in a mixed order. While, the results are still the same, only the order of the printed results are different. If the user wished to have the correlation printed in numerical order, bacteria 0 and bacteria 1 to bacteria 39 and bacteria 40, but still wish for a faster performance time using OpenMP then Figure 18 on the next page can be used instead.

295 /\*

296 [9]

336 \*/

337 #pragma omp for

338 for (int i = 0; i < number\_bacteria - 1; i++) {

339 /\*

340 [10]

341 \*/

342 for (int j = i + 1; j < number\_bacteria; j++) {

343 double correlation = CompareBacteria(b[i], b[j]);

344 printf("%2d %2d -> %.20lf\n", i, j, correlation);

345 }

346 }

347 }

Figure 18: Parallelised Nested For-Loop in Order

The difference between Figure 12 and Figure 18 is that the OpenMP statement removed the parallel functionality. This added a few seconds of extra time computational time to the application, but the printed correlation and bacteria pairs are printed to the console in order. This order difference can be seen in Figures 19 and 20 below.

|  |  |
| --- | --- |
| Figure 19: Correlation in Order | Figure 20: Correlation not in Order |

This version of the code was not taken though as the purpose of this assignment was to take a sequential program that could be parallelised and then parallelised to create the same results but in a shorter time frame that still produces the same results.

**4 Results**

In this section of the report, the correlation and times taken to run the application, both the original and high-performance models, on two different machines are discussed.

**4.1 Time Taken for Execution of Sequential and Parallelised Applications**

The systems specification for this machine can be found in section 2.1.3 of this report.

Table 1 below shows the time to complete the original and the parallelised versions of the Bioinformatics – Genome similarity using Frequency Vectors application. The application was executed five times and the average mean time was calculated. The parallelised version had each OpenMP line of code set to the specified number of threads below.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Time Taken (Seconds)** | | | | | |
| **Application Version** | 1st | 2nd | 3rd | 4th | 5th | **Average** |
| Original | 28 secs | 29 secs | 28 secs | 28 secs | 28 secs | **28.2 secs** |
| High Performance  (1 Thread) | 29 secs | 29 secs | 29 secs | 29 secs | 29 secs | **29.0 secs** |
| High Performance  (2 Threads) | 20 secs | 20 secs | 20 secs | 20 secs | 20 secs | **20.0 secs** |
| High Performance  (3 Threads) | 18 secs | 17 secs | 17 secs | 17 secs | 17 secs | **16.2 secs** |
| High Performance  (4 Threads) | 16 secs | 17 secs | 17 secs | 17 secs | 17 secs | **16.8 secs** |
| High Performance  (5 Threads) | 16 secs | 16 secs | 16 secs | 16 secs | 16 secs | **16.0 secs** |
| High Performance  (6 Threads) | 16 secs | 16 secs | 16 secs | 16 secs | 16 secs | **16.0 secs** |
| High Performance  (7 Threads) | 16 secs | 16 secs | 16 secs | 16 secs | 16 secs | **16.0 secs** |
| High Performance  (8 Threads) | 16 secs | 16 secs | 16 secs | 16 secs | 16 secs | **16.0 secs** |

Table 1: Sequential and Different Thread Count for Parallelised Application Run-Time

From the results shown above I cannot state whether the application is limited to four threads or if the test results were limited due to the number of physical cores available on the hardware used. A machine with more than four physical cores is required to prove the previous statement.

**4.2 Profiling Reports**

Figure 21 below is a CPU profiling report created in Visual Studio 2017 showing the CPU usage of the sequential application and percentage of work for functions during run-time.

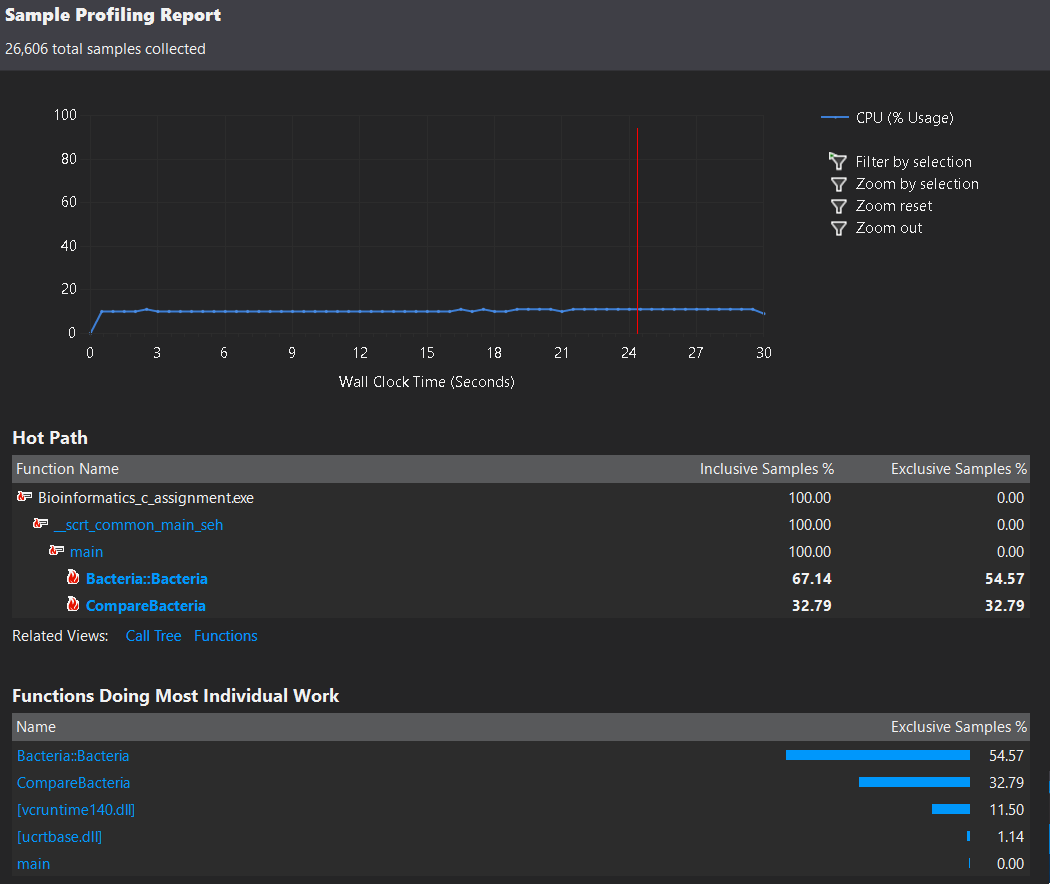


Figure 21: Sequential CPU Profiler

Figure 22 below is a CPU profiling report created in Visual Studio 2017 showing the CPU usage of the parallelised application and percentage of work for functions during run-time.

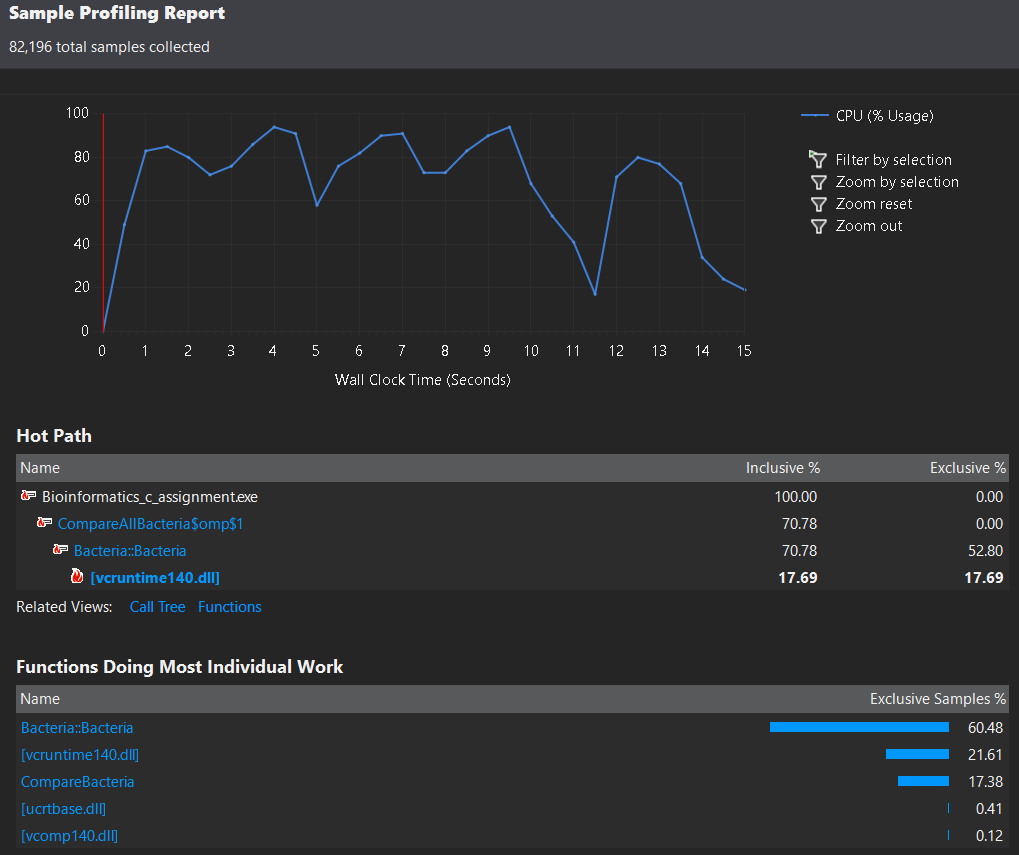


Figure 22: Parallelised CPU Profiler

**4.3 Speed Up Graph**

The graph below shows the speed up of the parallelised version of the application compared to the sequential application based on the time difference and the number of threads being used.

Figure 23: Speed Up Graph of Sequential vs Parallelised Over Threads

**5 Reflection**

Originally the Bioinformatics – Genome similarity using Frequency Vectors application was supposed to be parallelised using MPI, mpich3 or openMP, on my Raspberry Pi Beowulf cluster. This was scrapped because the application immediately gets a segmentation fault. Instead of fiddling around to get the code for the application to run properly I opted-out for running the application on through Visual Studio on a Windows OS machine so I could get the application parallelised. In terms of what I have learned, I usually always utilised Code::Blocks and the mingw 32 and 64 bit compilers when I program in C/C++. This is mainly because the IDE and compiler is free and widely used. However, during this semester, the CAB401 and IGB381 courses showed me how useful Visual Studio can be. The profiler and diagnostic tools allows me to see what functions and code could be further improved. Much easier than slowly debugging your way through the application manually. Another thing I have learnt is how simple OpenMP is to parallelise code with. I’ve always used POSIX threads when I wanted to parallelise for-loops or create multiple simultaneous executions of work. While there is a difference between the two, low-level vs high-level portable multiprocessing paradigms, it was a good experience to utilise and learn what OpenMP has to offer and how it works.

I believe my attempt at analysing the original code and going beyond trying different techniques to get the application to perform faster was almost a success. The one component that bothers me is the nested for-loop in the CompareAllBacteria function. An issue was encountered here that I could not fix.

**5.1 Issues Encountered**

I attempted to store the correlated data into a vector, dynamic array by splitting up the objectives. These objective were to calculate the correlation and printing to the console. This code can be seen in Figure 24 below.

std::vector<double> correlation;

int vectorSize = (number\_bacteria \* number\_bacteria) - number\_bacteria;

correlation.resize(vectorSize);

#pragma omp parallel for

for (int i = 0; i < number\_bacteria - 1; i++) {

for (int j = i + 1; j < number\_bacteria; j++) {

correlation[(i \* 10) + j] = CompareBacteria(b[i], b[j]);

}

}

for (int i = 0; i < number\_bacteria - 1; i++) {

for (int j = i + 1; j < number\_bacteria; j++) {

printf("%2d %2d -> %.20lf\n", i, j, correlation[(i \* 10) + j]);

}

}

Figure 24: Restructured Nested For-Loop Attempt

The result of this code sometimes provided the correct correlation values. At other times I found the correct correlation value, being returned by the function CompareBacteria, was changed when it was stored in the vector. This produced incorrect results. The purpose of this was to allow for the exact run-time to be achieved but with the correlated values printed in order. Splitting these objectives into two components allowed for the CompareBacteria to be parallelised providing a faster run-time. Then a sequential for-loop to print out the results in order.

It annoys me to know that I could have improved the ease of use, printing correlation in order, especially when a colleague of mine in CAB401 did the same thing and it worked for him.

As discussed in section 2.2.2, of this report, it was found that many data dependencies in other functions prevented further changes and parallelism for this application. At times the incorrect correlation between bacteria would occur. Further problems such as infinite loops and application crashes were encountered to further improve the application. However, solutions to these problems provided parallelisation that had no effect on the applications run-time making the parallelisation of multiple sections of code running at the same time on different threads pointless.

For more notes during testing the high-performance version of the Bioinformatics – Genome similarity using Frequency Vectors application can be found commented in the \*.cpp file of the Visual Studio project.

**6 Appendix**

**6.1 Original Code**

The original code for the Bioinformatics – Genome similarity using Frequency Vectors (C++) application was provided by Dr Wayne Kelly on the CAB401 High Performance and Parallel Computing QUT Blackboard page. A copy of the original code can be seen below. All bacteria files and the Visual Studio 2017 project can be found at the link here <https://github.com/Starwolf-001/CAB401/tree/master/Bioinformatics_c_assignment>.

1 #include <stdio.h>

2 #include <string.h>

3 #include <time.h>

4 #include <math.h>

5 #include <iostream>

6

7 int number\_bacteria;

8 char\*\* bacteria\_name;

9 long M, M1, M2;

10 short code[27] = { 0, 2, 1, 2, 3, 4, 5, 6, 7, -1, 8, 9, 10, 11, -1, 12, 11 13, 14, 15, 16, 1, 17, 18, 5, 19, 3 };

12 #define encode(ch) code[ch-'A']

13 #define LEN 6

14 #define AA\_NUMBER 20

15 #define EPSILON 1e-010

16

17 void Init()

18 {

19 M2 = 1;

20 for (int i = 0; i<LEN - 2; i++) // M2 = AA\_NUMBER ^ (LEN-2);

21 M2 \*= AA\_NUMBER;

22 M1 = M2 \* AA\_NUMBER; // M1 = AA\_NUMBER ^ (LEN-1);

23 M = M1 \*AA\_NUMBER; // M = AA\_NUMBER ^ (LEN);

24 }

25

26 class Bacteria

27 {

28 private:

29 long\* vector;

30 long\* second;

31 long one\_l[AA\_NUMBER];

32 long indexs;

33 long total;

34 long total\_l;

35 long complement;

36

37 void InitVectors()

38 {

39 vector = new long[M];

40 second = new long[M1];

41 memset(vector, 0, M \* sizeof(long));

42 memset(second, 0, M1 \* sizeof(long));

43 memset(one\_l, 0, AA\_NUMBER \* sizeof(long));

44 total = 0;

45 total\_l = 0;

46 complement = 0;

47 }

48

49 void init\_buffer(char\* buffer)

50 {

51 complement++;

52 indexs = 0;

53 for (int i = 0; i<LEN - 1; i++)

54 {

55 short enc = encode(buffer[i]);

56 one\_l[enc]++;

57 total\_l++;

58 indexs = indexs \* AA\_NUMBER + enc;

59 }

60 second[indexs]++;

61 }

62

63 void cont\_buffer(char ch)

64 {

65 short enc = encode(ch);

66 one\_l[enc]++;

67 total\_l++;

68 long index = indexs \* AA\_NUMBER + enc;

69 vector[index]++;

70 total++;

71 indexs = (indexs % M2) \* AA\_NUMBER + enc;

72 second[indexs]++;

73 }

74

75 public:

76 long count;

77 double\* tv;

78 long \*ti;

79

80 Bacteria(char\* filename)

81 {

82 FILE \* bacteria\_file = fopen(filename, "r");

83 InitVectors();

84

85 char ch;

86 while ((ch = fgetc(bacteria\_file)) != EOF)

87 {

88 if (ch == '>')

89 {

90 // skip rest of line

91 while (fgetc(bacteria\_file) != '\n');

92 char buffer[LEN - 1];

93 fread(buffer, sizeof(char), LEN - 1, bacteria\_file);

94 init\_buffer(buffer);

95 }

96 else if (ch != '\n')

97 cont\_buffer(ch);

98 }

99

100 long total\_plus\_complement = total + complement;

101 double total\_div\_2 = total \* 0.5;

102 int i\_mod\_aa\_number = 0;

103 int i\_div\_aa\_number = 0;

104 long i\_mod\_M1 = 0;

105 long i\_div\_M1 = 0;

106

107 double one\_l\_div\_total[AA\_NUMBER];

108 for (int i = 0; i<AA\_NUMBER; i++)

109 one\_l\_div\_total[i] = (double)one\_l[i] / total\_l;

110

111 double\* second\_div\_total = new double[M1];

112 for (int i = 0; i<M1; i++)

113 second\_div\_total[i] = (double)second[i] / total\_plus\_complement;

114 count = 0;

115 double\* t = new double[M];

116

117 for (long i = 0; i<M; i++)

118 {

119 double p1 = second\_div\_total[i\_div\_aa\_number];

120 double p2 = one\_l\_div\_total[i\_mod\_aa\_number];

121 double p3 = second\_div\_total[i\_mod\_M1];

122 double p4 = one\_l\_div\_total[i\_div\_M1];

123 double stochastic = (p1 \* p2 + p3 \* p4) \* total\_div\_2;

124

125 if (i\_mod\_aa\_number == AA\_NUMBER - 1)

126 {

127 i\_mod\_aa\_number = 0;

128 i\_div\_aa\_number++;

129 }

130 else

131 i\_mod\_aa\_number++;

132

133 if (i\_mod\_M1 == M1 - 1)

134 {

135 i\_mod\_M1 = 0;

136 i\_div\_M1++;

137 }

138 else

139 i\_mod\_M1++;

140

141 if (stochastic > EPSILON)

142 {

143 t[i] = (vector[i] - stochastic) / stochastic;

144 count++;

145 }

146 else

147 t[i] = 0;

148 }

149

150 delete second\_div\_total;

151 delete vector;

152 delete second;

153

154 tv = new double[count];

155 ti = new long[count];

156

157 int pos = 0;

158 for (long i = 0; i<M; i++)

159 {

160 if (t[i] != 0)

161 {

162 tv[pos] = t[i];

163 ti[pos] = i;

164 pos++;

165 }

166 }

167 delete t;

168

169 fclose(bacteria\_file);

170 }

171 };

172

173 void ReadInputFile(char\* input\_name)

174 {

175 FILE\* input\_file = fopen(input\_name, "r");

176 fscanf(input\_file, "%d", &number\_bacteria);

177 bacteria\_name = new char\*[number\_bacteria];

178

179 for (long i = 0; i<number\_bacteria; i++)

180 {

181 bacteria\_name[i] = new char[20];

182 fscanf(input\_file, "%s", bacteria\_name[i]);

183 strcat(bacteria\_name[i], ".faa");

184 }

185 fclose(input\_file);

186 }

187

188 double CompareBacteria(Bacteria\* b1, Bacteria\* b2)

189 {

190 double correlation = 0;

191 double vector\_len1 = 0;

192 double vector\_len2 = 0;

193 long p1 = 0;

194 long p2 = 0;

195 while (p1 < b1->count && p2 < b2->count)

196 {

197 long n1 = b1->ti[p1];

198 long n2 = b2->ti[p2];

199 if (n1 < n2)

200 {

201 double t1 = b1->tv[p1];

202 vector\_len1 += (t1 \* t1);

203 p1++;

204 }

205 else if (n2 < n1)

206 {

207 double t2 = b2->tv[p2];

208 p2++;

209 vector\_len2 += (t2 \* t2);

210 }

211 else

212 {

213 double t1 = b1->tv[p1++];

214 double t2 = b2->tv[p2++];

215 vector\_len1 += (t1 \* t1);

216 vector\_len2 += (t2 \* t2);

217 correlation += t1 \* t2;

218 }

219 }

220 while (p1 < b1->count)

221 {

222 long n1 = b1->ti[p1];

223 double t1 = b1->tv[p1++];

224 vector\_len1 += (t1 \* t1);

225 }

226 while (p2 < b2->count)

227 {

228 long n2 = b2->ti[p2];

229 double t2 = b2->tv[p2++];

230 vector\_len2 += (t2 \* t2);

231 }

232

233 return correlation / (sqrt(vector\_len1) \* sqrt(vector\_len2));

234 }

235

236 void CompareAllBacteria()

237 {

238 Bacteria\*\* b = new Bacteria\*[number\_bacteria];

239 for (int i = 0; i<number\_bacteria; i++)

240 {

241 printf("load %d of %d\n", i + 1, number\_bacteria);

242 b[i] = new Bacteria(bacteria\_name[i]);

243 }

244

245 for (int i = 0; i<number\_bacteria - 1; i++)

246 for (int j = i + 1; j<number\_bacteria; j++)

247 {

248 printf("%2d %2d -> ", i, j);

249 double correlation = CompareBacteria(b[i], b[j]);

250 printf("%.20lf\n", correlation);

251 }

252 }

253

254 int main(int argc, char \* argv[])

255 {

256 time\_t t1 = time(NULL);

257

258 Init();

259 ReadInputFile(argv[1]);

260 CompareAllBacteria();

261

262 time\_t t2 = time(NULL);

263 printf("time elapsed: %d seconds\n", t2 - t1);

264

265 system("pause");

266

267 return 0;

277 }

**6.2 High Performance and Parallel Code**

This code for the Bioinformatics – Genome similarity using Frequency Vectors (C++) application was analysed and tested to provide the correct results in a short time as per the requirement for the CAB401 High Performance and Parallel Computing Assignment. This code can be seen below. All bacteria files, notes taken during testing, code and the Visual Studio 2017 project can be found at the link here <https://github.com/Starwolf-001/CAB401/tree/master/Bioinformatics_c_assignment_high_performance>. There are lines missing in this code. These missing lines of code are commented notes based on the testing, and analysis when parallelising the original Bioinformatics – Genome similarity using Frequency Vectors application.

1 #include <stdio.h>

2 #include <string.h>

3 #include <time.h>

4 #include <math.h>

5 #include <iostream>

6 #include <omp.h>

7

8 int number\_bacteria;

9 char\*\* bacteria\_name;

10 long M, M1, M2;

11 short code[27] = {0, 2, 1, 2, 3, 4, 5, 6, 7, -1, 8, 9, 10, 11, -1, 12, 13, 14, 15, 16, 1, 17, 18, 5, 19, 3};

12 #define encode(ch) code[ch-'A']

13 #define LEN 6

14 #define AA\_NUMBER 20

15 #define EPSILON 1e-010

16

17 void Init() {

18 M2 = 1;

24 for (int i = 0; i < LEN - 2; i++) { // M2 = AA\_NUMBER ^ (LEN-2);

25 M2 \*= AA\_NUMBER;

26 }

27 M1 = M2 \* AA\_NUMBER; // M1 = AA\_NUMBER ^ (LEN-1);

28 M = M1 \*AA\_NUMBER; // M = AA\_NUMBER ^ (LEN);

29 }

30

31 class Bacteria {

32 private:

33 long\* vector;

34 long\* second;

35 long one\_l[AA\_NUMBER];

36 long indexs;

37 long total;

38 long total\_l;

39 long complement;

40

41 void InitVectors() {

42 vector = new long[M];

43 second = new long[M1];

44 memset(vector, 0, M \* sizeof(long));

45 memset(second, 0, M1 \* sizeof(long));

46 memset(one\_l, 0, AA\_NUMBER \* sizeof(long));

47 total = 0;

48 total\_l = 0;

49 complement = 0;

50 }

51

52 void init\_buffer(char\* buffer) {

53 complement++;

54 indexs = 0;

59 for (int i = 0; i < LEN - 1; i++) {

60 short enc = encode(buffer[i]);

61 one\_l[enc]++;

62 total\_l++;

63 indexs = indexs \* AA\_NUMBER + enc;

64 }

65 second[indexs]++;

66 }

67

68 void cont\_buffer(char ch) {

69 short enc = encode(ch);

70 one\_l[enc]++;

71 total\_l++;

72 long index = indexs \* AA\_NUMBER + enc;

73 vector[index]++;

74 total++;

75 indexs = (indexs % M2) \* AA\_NUMBER + enc;

76 second[indexs]++;

77 }

78

79 public:

80 long count;

81 double\* tv;

82 long \*ti;

83

84 Bacteria(char\* filename) {

92 FILE \* bacteria\_file = fopen(filename, "r");

93 InitVectors();

94

95 char ch;

96 while ((ch = fgetc(bacteria\_file)) != EOF) {

97 if (ch == '>') {

98 while (fgetc(bacteria\_file) != '\n'); // skip rest of line

99

100 char buffer[LEN - 1];

101 fread(buffer, sizeof(char), LEN - 1, bacteria\_file);

102 init\_buffer(buffer);

103 }

104 else if (ch != '\n')

105 cont\_buffer(ch);

106 }

107 fclose(bacteria\_file);

108

109 long total\_plus\_complement = total + complement;

110 double total\_div\_2 = total \* 0.5;

111 int i\_mod\_aa\_number = 0;

112 int i\_div\_aa\_number = 0;

113 long i\_mod\_M1 = 0;

114 long i\_div\_M1 = 0;

115

116 double one\_l\_div\_total[AA\_NUMBER];

117 double\* second\_div\_total = new double[M1];

130 for (int i = 0; i < AA\_NUMBER; i++) {

131 one\_l\_div\_total[i] = (double)one\_l[i] / total\_l;

132 }

138 for (int i = 0; i < M1; i++) {

139 second\_div\_total[i] = (double)second[i] / total\_plus\_complement;

140 }

141

142 count = 0;

143 double\* t = new double[M];

144

156 for (long i = 0; i < M; i++) {

157 double p1 = second\_div\_total[i\_div\_aa\_number];

158 double p2 = one\_l\_div\_total[i\_mod\_aa\_number];

159 double p3 = second\_div\_total[i\_mod\_M1];

160 double p4 = one\_l\_div\_total[i\_div\_M1];

161 double stochastic = (p1 \* p2 + p3 \* p4) \* total\_div\_2;

162

163 if (i\_mod\_aa\_number == AA\_NUMBER - 1) {

164 i\_mod\_aa\_number = 0;

165 i\_div\_aa\_number++;

166 }

167 else {

168 i\_mod\_aa\_number++;

169 }

170

171 if (i\_mod\_M1 == M1 - 1) {

172 i\_mod\_M1 = 0;

173 i\_div\_M1++;

174 }

175 else {

176 i\_mod\_M1++;

177 }

178

179 if (stochastic > EPSILON) {

180 t[i] = (vector[i] - stochastic) / stochastic;

181 count++;

182 }

183 else {

184 t[i] = 0;

185 }

186 }

187

188 delete second\_div\_total;

189 delete vector;

190 delete second;

191

192 tv = new double[count];

193 ti = new long[count];

194

195 int pos = 0;

201 for (long i = 0; i < M; i++) {

202 if (t[i] != 0) {

203 tv[pos] = t[i];

204 ti[pos] = i;

205 pos++;

206 }

207 }

208 delete t;

209 }

210 };

211

212 void ReadInputFile(char\* input\_name) {

213 FILE\* input\_file = fopen(input\_name, "r");

214 fscanf(input\_file, "%d", &number\_bacteria);

215 bacteria\_name = new char\*[number\_bacteria];

220 for (long i = 0; i < number\_bacteria; i++) {

221 bacteria\_name[i] = new char[20];

222 fscanf(input\_file, "%s", bacteria\_name[i]);

223 strcat(bacteria\_name[i], ".faa");

224 }

225 fclose(input\_file);

226 }

227

228 double CompareBacteria(Bacteria\* b1, Bacteria\* b2) {

229 double correlation = 0;

230 double vector\_len1 = 0;

231 double vector\_len2 = 0;

232 long p1 = 0;

233 long p2 = 0;

237 while (p1 < b1->count && p2 < b2->count) {

238 long n1 = b1->ti[p1];

239 long n2 = b2->ti[p2];

240 if (n1 < n2) {

241 double t1 = b1->tv[p1];

242 vector\_len1 += (t1 \* t1);

243 p1++;

244 }

245 else if (n2 < n1) {

246 double t2 = b2->tv[p2];

247 p2++;

248 vector\_len2 += (t2 \* t2);

249 }

250 else {

251 double t1 = b1->tv[p1++];

252 double t2 = b2->tv[p2++];

253 vector\_len1 += (t1 \* t1);

254 vector\_len2 += (t2 \* t2);

255 correlation += t1 \* t2;

256 }

257 }

264 while (p1 < b1->count) {

265 long n1 = b1->ti[p1];

266 double t1 = b1->tv[p1++];

267 vector\_len1 += (t1 \* t1);

268 }

274 while (p2 < b2->count) {

275 long n2 = b2->ti[p2];

276 double t2 = b2->tv[p2++];

277 vector\_len2 += (t2 \* t2);

278 }

279 return correlation / (sqrt(vector\_len1) \* sqrt(vector\_len2));

280 }

281

282 void CompareAllBacteria() {

283 Bacteria\*\* b = new Bacteria\*[number\_bacteria];

291 #pragma omp parallel for

292 for (int i = 0; i < number\_bacteria; i++) {

293 printf("load %d of %d\n", i + 1, number\_bacteria);

294 b[i] = new Bacteria(bacteria\_name[i]);

295 }

296

342 #pragma omp parallel for

343 for (int i = 0; i < number\_bacteria - 1; i++) {

347 for (int j = i + 1; j < number\_bacteria; j++) {

348 double correlation = CompareBacteria(b[i], b[j]);

349 printf("%2d %2d -> %.20lf\n", i, j, correlation);

350 }

351 }

352 }

353

354 int main(int argc, char \* argv[]) {

355 time\_t t1 = time(NULL);

364 Init();

365 ReadInputFile(argv[1]);

366 CompareAllBacteria();

367

368 time\_t t2 = time(NULL);

369 printf("time elapsed: %d seconds\n", t2 - t1);

370

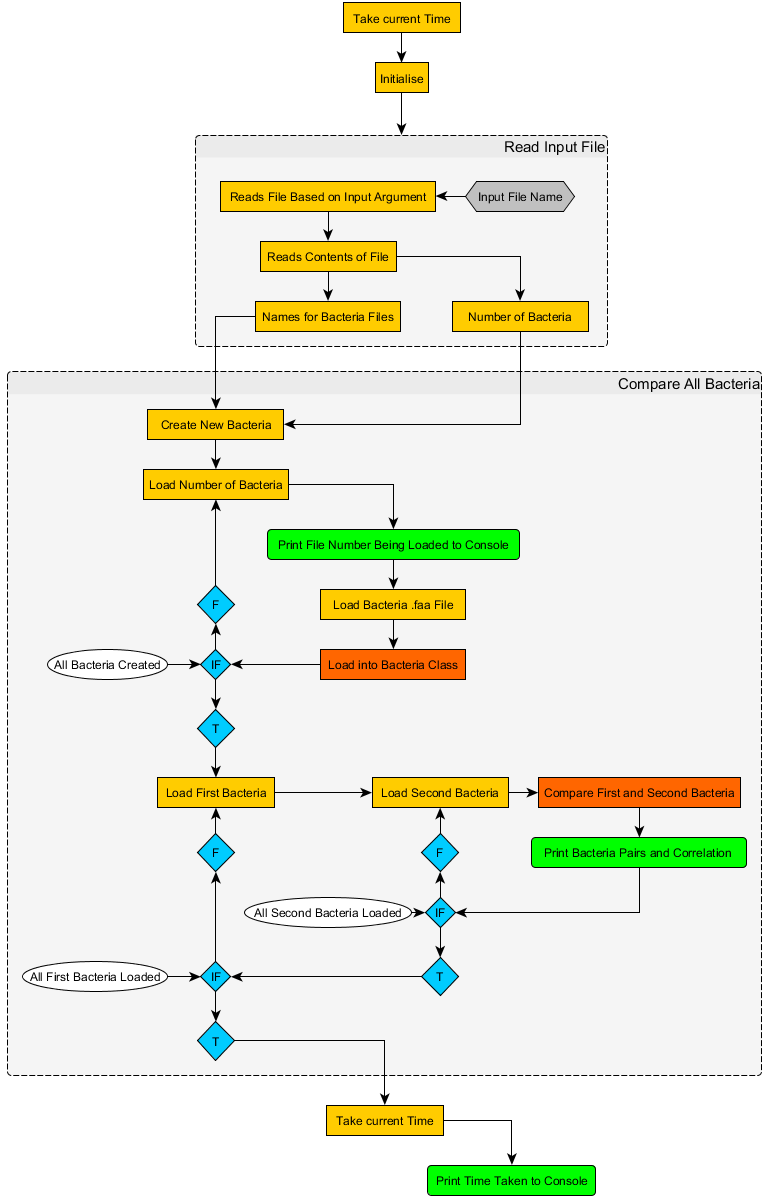
371 system("pause");

372

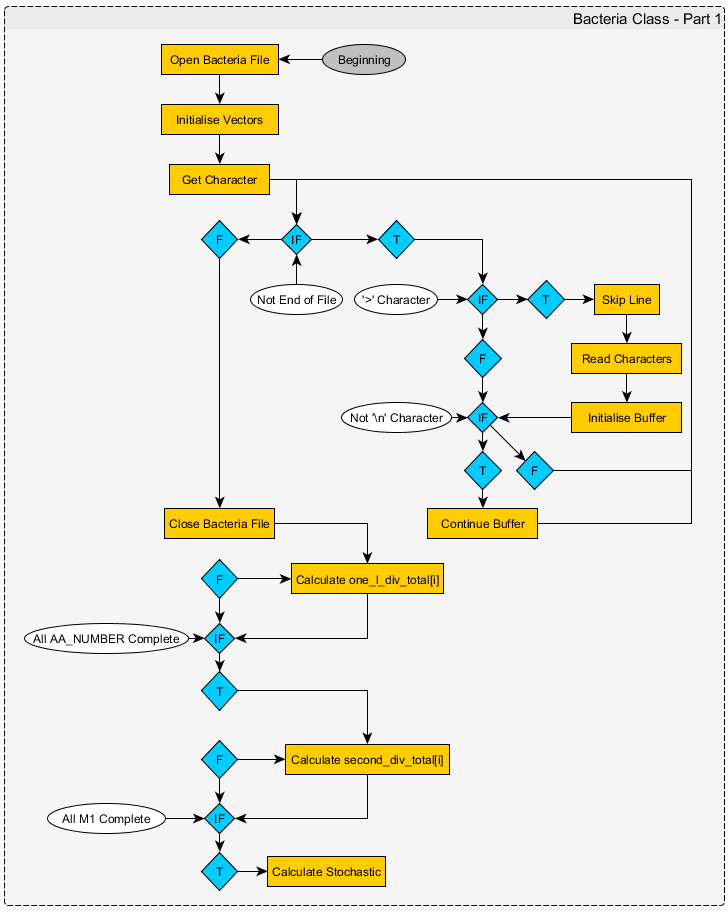
373 return 0;

374 }

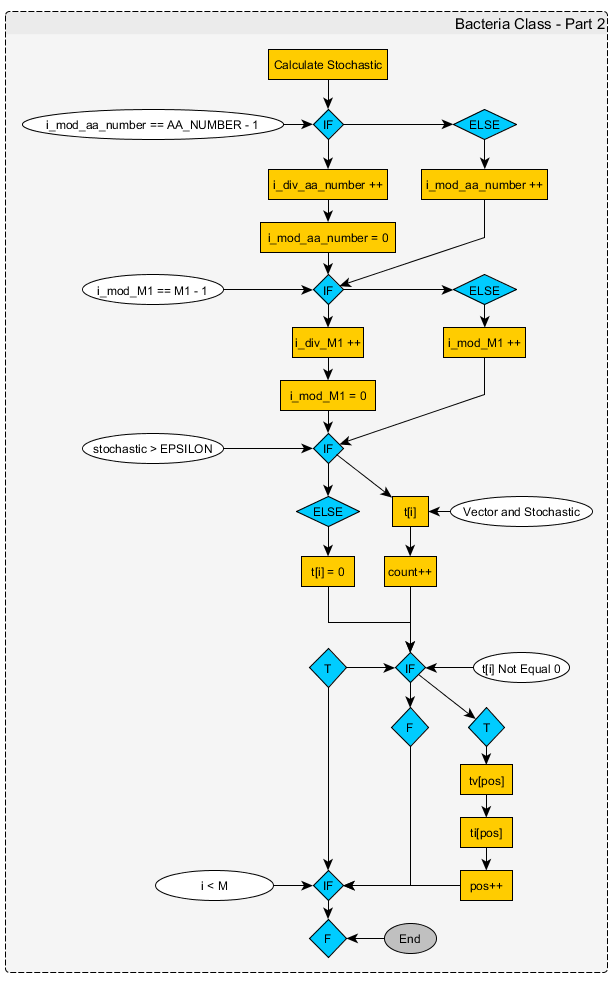
**6.3 Detailed Software Architecture**



**6.4 Bacteria Class – Part 1 Software Architecture**



**6.5 Bacteria Class – Part 2 Software Architecture**



**6.6 Compare Bacteria Software Architecture**

