CAB401 Parallelisation Project

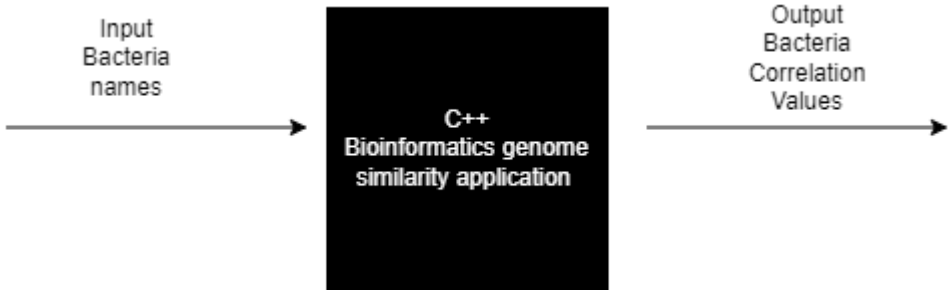
Hari Markonda Patnaikuni n10789511

The Application:

The sequential application being parallelised is a C++ Bioinformatics genome similarity program which uses frequency vectors to find these genome similarities and then outputs each bacteria's correlation score. This application utilises two external files for the calculation of correlation scores for each bacteria. The first file is a '.faa' dictionary of genes. These genes are used to calculate the correlation scores between the current bacteria and others.

For this to occur the current bacteria's name must be concatenated with a '.faa' so that the name of the bacteria can be pointed back to the dictionary of genes. The process of calculating correlation scores involves the use of a biological principle known as k-mers. Also, the correlation score is out of 1. Hence, the black box diagram for this application can be described as an input of bacteria's specifically their genes and then an output of all the bacteria's similarities to one another. This is illustrated below.

Figure 1: Application Black Box Diagram



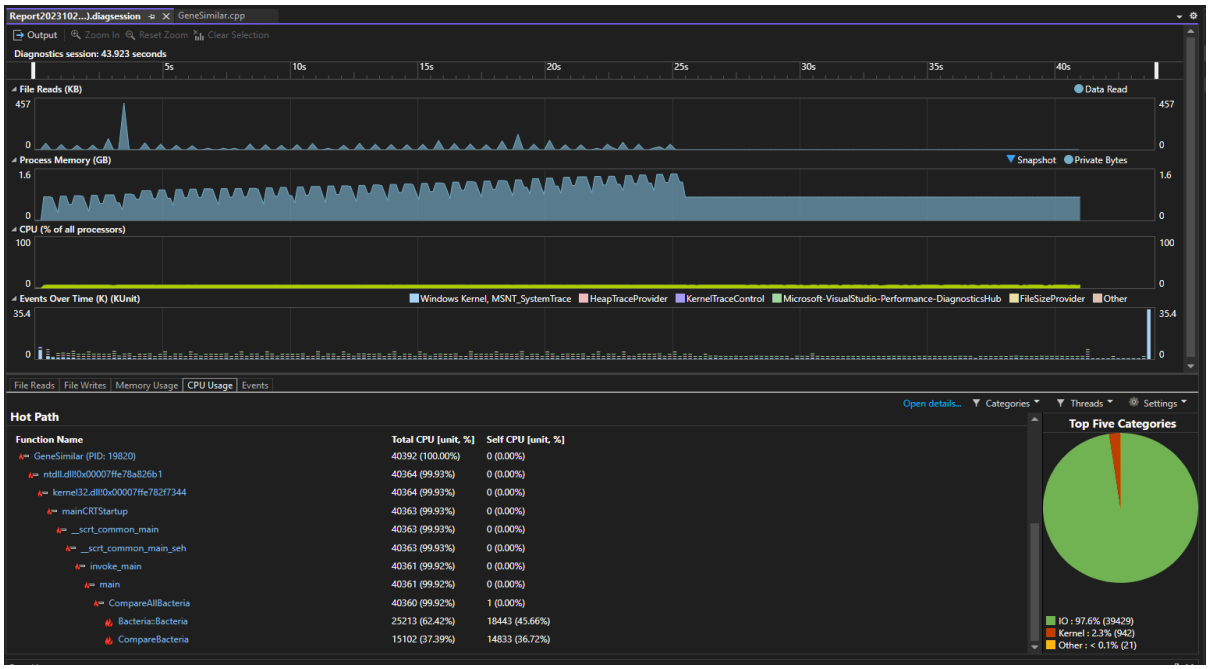
The structure of the original sequential application involves several functions within a main source file known as 'GeneSimilar.cpp'. It is important to note that for this project the improved version of this application is used which has more optimised code and runs faster. This main source file contains 9 functions and one main function. In addition to this, there are many initialisations of definitions and variables used by the first few functions.

The first few functions deal with initialising the frequency vectors, buffers, and loading the bacteria input file. The two major functions that deal with using the frequency vectors and calculating correlation scores are the 'CompareBacteria' and 'CompareAllBacteria' functions. The 'CompareBacteria' function is used to find the correlation score for one bacteria within the input list of bacteria names. Hence, the 'CompareAllBacteria' function is used to find all of the bacteria's correlation scores via the use of a nested for loop and the 'CompareBacteria' function. Finally, the main function calls the initialisation, read input file, and 'CompareAllBacteria' functions to begin the calculation of correlation scores.

Analysis of Potential Parallelism:

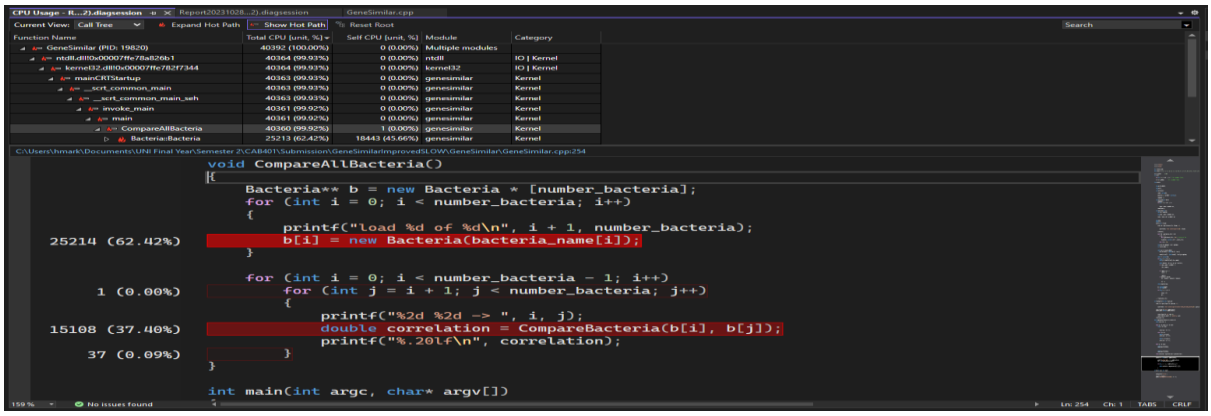
Before any analysis of potential parallelism was done the application was first profiled using Visual Studio Profiler. The below hot path hierarchy figure shows that the 'CompareAllBacteria' function uses 99.92% of total CPU resources or 40360 CPU units out of a total of 40392. Hence, this function has been identified as a major hotspot for this application.

Figure 2: Sequential Application Visual Studio Profiler Hot Path Hierarchy



In addition to this, it is shown in Figure 2 that the percentage of processors used throughout the runtime of the application is around 6%. As a result, it was decided to see if the 'CompareAllBacteria' function could be parallelised. Hence, the hot path link for the 'CompareAllBacteria' function was selected to see the hotspots within the function that are causing the application to run for a longer time. These hotspots with the function are shown in the below figure.

Figure 3: 'CompareAllBacteria' Function Visual Studio Profiler Hotspots



As shown in the above figure, the loading of the bacteria names and pointing them to their specific dictionaries takes up 64.42% of the CPU usage. Also, the calculation of each bacteria's correlation score

takes up 37.40% of the CPU usage. Hence, it was decided to attempt to parallelise this function with the use of an implicit parallelisation API known as OpenMP.

However, before this can be done the 'CompareAllBacteria' and any other functions it calls must be analysed for data dependencies. Data dependencies were found in the 'Bacteria' class which initialises a bacteria object that is used in the 'CompareBacteria' and 'CompareAllBacteria' functions. In addition to this, the use of comparing pairs of bacteria is done in the 'CompareAllBacteria' function's nested for loop which calls the 'CompareBacteria' function. This function in turn uses 'Bacteria' objects to find their correlation scores. Hence, these are the main data dependencies found in the sequential program.

To map computation to multiple processors hence, parallelising the program, the use of the below line of code is used at the start of the 'CompareAllBacteria' function.

```
omp_set_num_threads(16);
```

This code tells the program's 'CompareAllBacteria' function to use all 16 virtual threads. After this, the below line of OpenMP parallelisation code is added to the outside of the load bacteria for loop within the 'CompareAllBacteria' function.

```
#pragma omp parallel for
```

In addition to this, the for loop which calculates each bacteria's correlation score is restructured as shown below with more OpenMP parallelisation code.

Figure 4: Correlation score OpenMP parallelisation

```
291     //ADDED BELOW LINE OF SAFE CODE
292     #pragma omp parallel for
293     for (int i = 0; i < number_bacteria - 1; i++) {
294         //ADDED BELOW LINE OF SAFE CODE
295         #pragma omp parallel for
296         for (int j = i + 1; j < number_bacteria; j++) {
297
298
299             double correlation = CompareBacteria(b[i], b[j]);
300
301             //ADDED BELOW LINE OF SAFE CODE
302             #pragma omp critical
303             {
304                 fprintf(outputFile, "%d %d -> %.20lf\n", i, j, correlation);
305
306                 totalCorrelation += correlation; // Add the correlation value
307             }
308
309         }
310
311     }
312     //ADDED BELOW LINE OF SAFE CODE
313     #pragma omp critical
314     fprintf(outputFile, "Total Correlation: %.20lf\n", totalCorrelation);
315     fclose(outputFile);
316
317 }
```

Parallelisation Results:

The below table shows the change in execution time as the number of virtual cores is increased for both the debug and release versions of the parallelised application. For the sequential application, it takes 41 and 19 seconds for the debug and release versions respectively to run. Finally, the parallelised versions of the application take 11 and 6 seconds while utilising all 16 virtual threads of the CPU. It should be noted that all tests conducted with this application sequential and parallel were with the x64-bit configuration.

Table 1: Debug and Release Execution Times

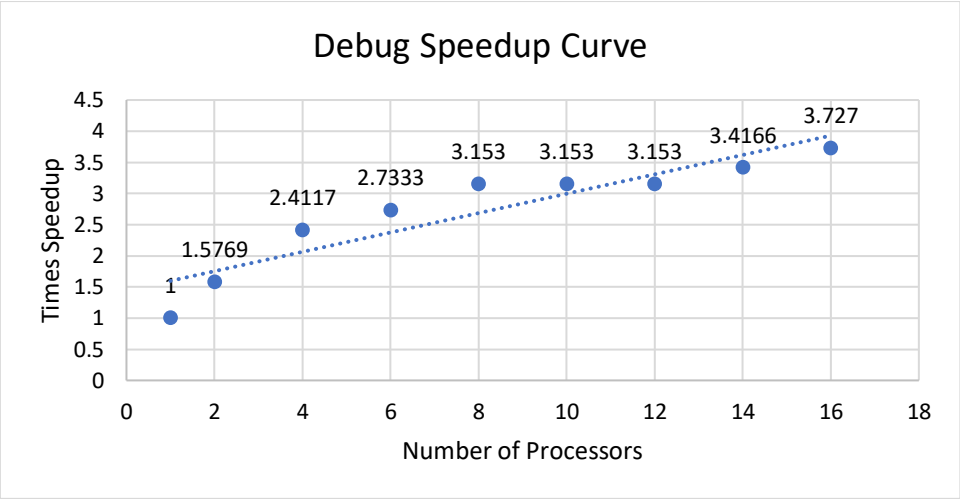
Number of Virtual Cores	Debug Time	Release Time
1	41	19
2	26	11
4	17	8
6	15	7
8	13	7
10	13	7
12	13	7
14	12	7
16	11	6

This information was then used to calculate speedup values for each version of the application. The formula used to calculate speedup is shown below.

$$Speedup = \frac{Best\ Sequential\ Time\ in\ Seconds}{Best\ Parallel\ Time\ in\ Seconds}$$

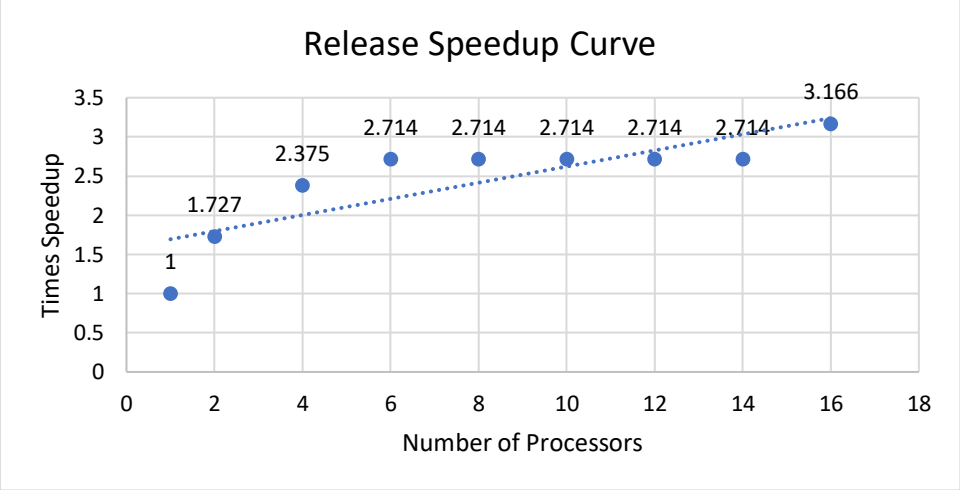
The results of the applied implicit parallelisation code added to the 'CompareAllBacteria' function had a substantial increase in speedup. Figure 5 shows the speedup curve for the Debug version of the application. This figure shows that the implementation of the OpenMP parallelisation code was successful in providing a relatively linear speedup. The final speedup while using all 16 virtual threads resulted in a speedup value of 3.727 times. This is very close to the best possible speedup due to the use of more than 4 virtual cores.

Figure 5: Debug Speedup Curve



The Release version of this application also has a relatively linear increase in performance as the number of virtual cores is increased. For this version of the application, the final speedup value is 3.166 times which is also very close to the best possible speedup.

Figure 6: Release Speedup Curve



The profiling results for the parallelised version of this application have also improved. Figure 7 shows that the application is now using 100% of all processors most of the time during the runtime of the application. This is a significant increase from the previously observed 6% found in Figure 2. In addition to this, the Total CPU % for the 'CompareAllBacteria' function has decreased from 99.92% to 78.75%. However, the function is also using more CPU units now when compared to the sequential application as the function is using more virtual threads. Also, the process memory used has increased from a peak of 1.6 for the sequential program to 13.5 for the parallelised program.

Figure 7: Parallelised Application Visual Studio Profiler Results

When parallelising an application, the changes made can lead to incorrect results hence, a test needs to be conducted to monitor this. For this application, the correlation values of all bacteria were accumulated and stored in a variable called 'totalCorrelation'. A reference sequential 'totalCorrelation' value was recorded and referenced whenever making changes to the application. This value is 9.84325092573049964528. The value was calculated with the sequential application with the below define statements at the start of the program.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <math.h>

int number_bacteria;
char** bacteria_name;
long M, M1, M2;
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 };
#define encode(ch)              code[ch-'A']
#define LEN                             6
#define AA_NUMBER               20
#define   EPSILON                       1e-010
```

The same definitions were used when parallelising the application as well. It was found that the 'totalCorrelation' value does slightly vary sometimes for the final parallelised program. This is shown in the console output screenshots found in Appendix A when the parallelised program was run multiple times with the same define statements.

As shown in Appendix A, the difference between the correct total correlation value of 9.84325092573049964528 and the parallelised total correlation values is very small. In two out of 5 test cases, the parallelised total correlation values are the same as the sequential total correlation value. However, there are still slight differences with the three other tests which have different decimal values after about 12 decimal places.

Parallel Programming Software, Tools and Technologies Used

There are many software and hardware technologies used to enable the parallelisation process of this application. The computer used to run this application is a custom-built PC with the latest version of Windows 10 installed. This computer has an AMD Ryzen 7 3700X 8 core CPU paired with an ASUS RTX 2070 graphics card and 32GB of RAM at a speed of 3200MHz. Mainly the CPU and RAM are the most important pieces of hardware due to the inherent increase in performance of using an 8-core CPU and 32GB of RAM. The CPU is especially useful since the use of implicit parallelisation via OpenMP means that the application can use 16 virtual threads in some parts of the application.

Hence, increasing the optimal speedup capability. Another technology used is the IDE known as Visual Studio 2022. This IDE is extensively used in the industry and has an in-built application profiling software known as Visual Studio Profiler. Visual Studio Profiler was used to find the hotspots of code within the application source code. These hotspots are sections of code that use more computing resources when compared to other parts of the code. In addition to this, the profiler has many other profiling capabilities such as CPU usage, GPU usage, memory usage and events monitoring.

OpenMP was used to implicitly parallelise the 'CompareAllBacteria' function. This parallel programming API is commonly used to parallelise C/C++ applications. For this application, the use of OpenMP #pragma code and setting the number of virtual threads to be used was exploited to increase the speedup of the application.

The C++ programming language is also another software technology used primarily for its capability of saving the correlation values for each bacteria into a text file locally and then retrieving that text file and outputting the correlation values to the console. This method is used to bypass the printing errors that have been observed when using OpenMP parallelisation code in the 'CompareAllBacteria' function's nested for loop.

Problems Encountered

There were two major problems encountered while trying to parallelise this application. Firstly, after adding the necessary OpenMP code to the load bacteria for loop the correlation values were still printing the same as the sequential program and with the same total correlation value. However, after adding OpenMP code to the calculate bacteria correlation value nested for loop it was apparent that the printed correlation values were not printing correctly as some had more than 1 correlation value.

The attempt was made to use more OpenMP constructs such as ordered and shared. However, this lost the performance increase that was observed without these constructs. Hence, the decision was made to save the correlation values in a locally stored text file and then read that text file and print the correlation values to the console. This both solved the printing issue and kept the performance gain.

Finally, the code added to the final parallelised program is shown below.

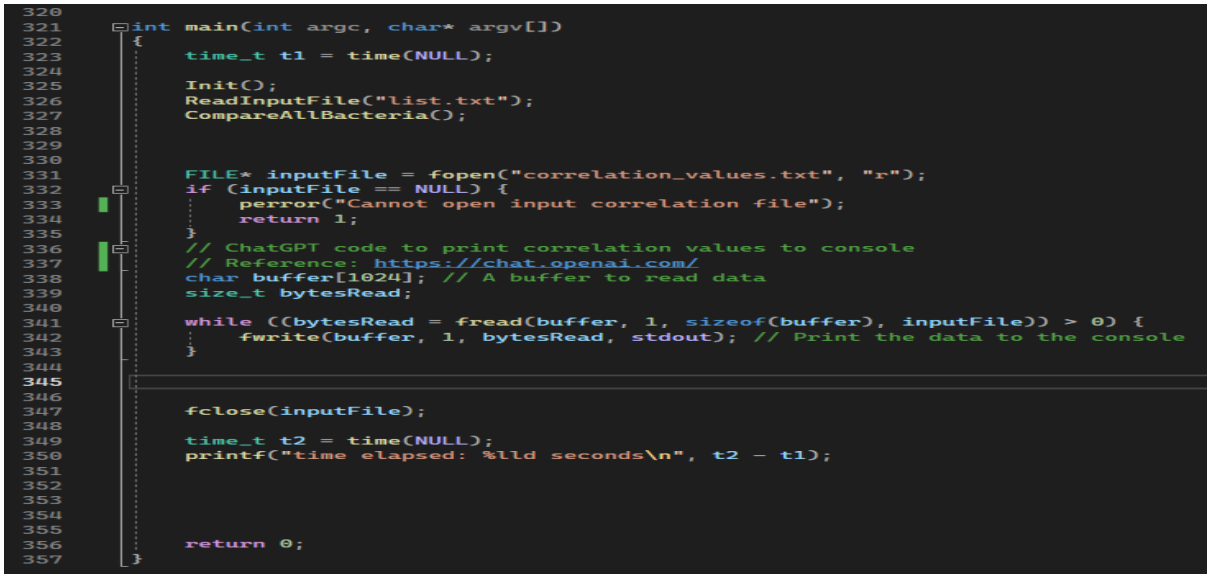
Figure 8: Final Code added

```
void CompareAllBacteria()
{
    double totalCorrelation = 0.0; // Store the total correlation
    Bacteria** b = new Bacteria * [number_bacteria];

    //ADDED BELOW LINE OF SAFE CODE
    omp_set_num_threads(16);

    //ADDED BELOW LINE OF SAFE CODE

    #pragma omp parallel for
    for (int i = 0; i < number_bacteria; i++)

    {
        printf("load %d of %d\n", i + 1, number_bacteria);
        b[i] = new Bacteria(bacteria_name[i]);
    }

    FILE* outputFile = fopen("correlation_values.txt", "w");
    if (outputFile == NULL) {
        perror("Cannot open correlation file");

    }
    //ADDED BELOW LINE OF SAFE CODE
    #pragma omp parallel for
    for (int i = 0; i < number_bacteria - 1; i++) {
        //ADDED BELOW LINE OF SAFE CODE
        #pragma omp parallel for
        for (int j = i + 1; j < number_bacteria; j++) {

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

            //ADDED BELOW LINE OF SAFE CODE
            #pragma omp critical
            {
                fprintf(outputFile, "%d %d -> %.20lf\n", i, j, correlation);

                totalCorrelation += correlation; // Add the correlation value
            }

        }
    }
    //ADDED BELOW LINE OF SAFE CODE
    #pragma omp critical
    fprintf(outputFile, "Total Correlation: %.20lf\n", totalCorrelation);
    fclose(outputFile);

}
```

As shown in the above figure, OpenMP code is added to the top of the load bacteria for loop and to the nested for loop for the calculation of correlation values. The line of code written as “omp_set_number_threads(16);” is used to set the number of threads that can be used within the ‘CompareAllBacteria’ function. In addition to this, the instances where “#pragma omp parallel for” are used are for running those for loops in parallel. It also should be noted that if only the load bacteria for loop is parallelised then the fastest execution time is 24 seconds in debug mode instead of 11 seconds with both the load bacteria and nested for loops parallelised.

“#pragma omp critical” is used for critical sections of code that need to be protected in multiple threads to reduce the chance of the total correlation values and correlation values being incorrect. In addition to this, there is code added to create a locally saved text file that contains the correlation values for all bacteria and the total correlation value. These values are then called from this text file in the main function and printed to the console as shown below.

Figure 9: Main Function



```
int main(int argc, char* argv[])
{
    time_t t1 = time(NULL);

    Init();
    ReadInputFile("list.txt");
    CompareAllBacteria();



    FILE* inputFile = fopen("correlation_values.txt", "r");
    if (inputFile == NULL) {
        perror("Cannot open input correlation file");
        return 1;
    }
    // ChatGPT code to print correlation values to console
    // Reference: https://chat.openai.com/
    char buffer[1024]; // A buffer to read data
    size_t bytesRead;

    while ((bytesRead = fread(buffer, 1, sizeof(buffer), inputFile)) > 0) {
        fwrite(buffer, 1, bytesRead, stdout); // Print the data to the console
    }


    fclose(inputFile);

    time_t t2 = time(NULL);
    printf("time elapsed: %lld seconds\n", t2 - t1);




    return 0;
}
```

As a result, there are 25 lines of new code added to the parallelised application which excludes new comments added.

Conclusion and Recommendations

As a result, the parallelisation of the Bioinformatics genome similarity program can be considered a success. This is due to the use of OpenMP parallelisation code and how it contributes to a speedup of 3.727 and 3.166 times when compared to the sequential program. This results in a program that used to run for 41 seconds to 11 seconds for the debug version and 19 to 6 seconds for the release version. However, there are slight discrepancies in the total correlation value which occurs after the first 12 decimal places of the total correlation value.

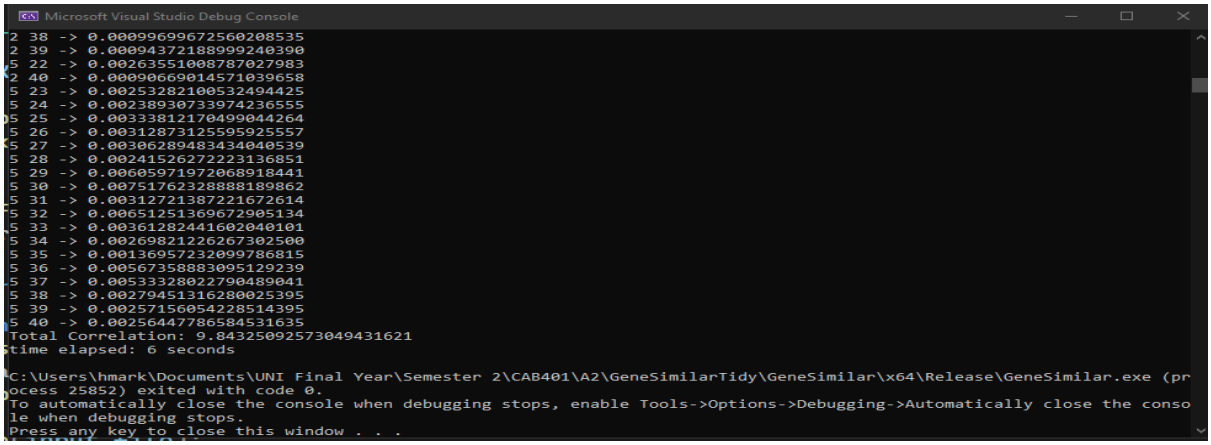
Since this occurs after so many decimal places it can be said that this is a minor issue that does not impact the accuracy of the overall program too dramatically. I do believe that I could have exposed more parallelism in other functions within this program. This may have led to a higher overall speedup for the application. Hence, in the future, I would also like to attempt to parallelise the 'CompareBacteria' function.

Appendix:

Appendix A:

Varying Correlation Values:

Test 1:



Test 2:

Test 3:

Test 4:


Microsoft Visual Studio Debug Console
2 40 -> 0.00090669014571039658
5 20 -> 0.00321748506167430625
5 21 -> 0.00217349185433886186
5 22 -> 0.00263551008787027983
5 23 -> 0.00253282100532494425
5 24 -> 0.00238930733974236555
5 25 -> 0.00333812170499044264
5 26 -> 0.00312873125595925557
5 27 -> 0.00306289483434040539
5 28 -> 0.00241526272223136851
5 29 -> 0.00605971972068918441
5 30 -> 0.00751762328888189862
5 31 -> 0.00312721387221672614
5 32 -> 0.00651251369672905134
5 33 -> 0.00361282441602040101
5 34 -> 0.00269821226267302500
5 35 -> 0.00136957232099786815
5 36 -> 0.00567358883095129239
5 37 -> 0.00533328022790489041
5 38 -> 0.00279451316280025395
5 39 -> 0.00257156054228514395
5 40 -> 0.00256447786584531635
Total Correlation: 9.84325092573049786893
time elapsed: 7 seconds
C:\Users\hmark\Documents\UNI Final Year\Semester 2\CAB401\A2\GeneSimilarTidy\GeneSimilar\x64\Release\GeneSimilar.exe (process 9040) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .

Test 5:


Microsoft Visual Studio Debug Console
5 21 -> 0.00217349185433886186
2 39 -> 0.00094372188999240390
2 40 -> 0.00090669014571039658
5 22 -> 0.00263551008787027983
5 23 -> 0.00253282100532494425
5 24 -> 0.00238930733974236555
5 25 -> 0.00333812170499044264
5 26 -> 0.00312873125595925557
5 27 -> 0.00306289483434040539
5 28 -> 0.00241526272223136851
5 29 -> 0.00605971972068918441
5 30 -> 0.00751762328888189862
5 31 -> 0.00312721387221672614
5 32 -> 0.00651251369672905134
5 33 -> 0.00361282441602040101
5 34 -> 0.00269821226267302500
5 35 -> 0.00136957232099786815
5 36 -> 0.00567358883095129239
5 37 -> 0.00533328022790489041
5 38 -> 0.00279451316280025395
5 39 -> 0.00257156054228514395
5 40 -> 0.00256447786584531635
Total Correlation: 9.84325092573050142164
time elapsed: 7 seconds
C:\Users\hmark\Documents\UNI Final Year\Semester 2\CAB401\A2\GeneSimilarTidy\GeneSimilar\x64\Release\GeneSimilar.exe (process 9688) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .