HIGH PERFORMANCE & PARALLEL COMPUTING

OCTOBER 27, 2018

DOUGLAS BRENNAN – N7326645
CAB401 - QUT

Overview

This report details my conducted parallelisation of a C++ application which compares bacteria files against one another.

2.0 – Application Summary & Analysis

This section details what the program computes and how it processes the information.

2.1 – Overview:

The program compares each bacteria file (.faa) against one another, each comparison value is then printed to the console. A time elapsed value is printed to the console for how long the process took in seconds.

2.2 – Description:

The program reads the provided 'list.txt' file which contains a list of bacteria file names to be compared, this is achieved through use of the ReadInputFile() method (see Figure 1).

```
//Reads the .txt file and adds names of bacterias in the text file to a string array bacteria_name[]
void ReadInputFile(char* input_name)
{
    FILE* input_file = fopen(input_name, "r");
    fscanf(input_file, "%d", &number_bacteria);
    bacteria_name = new char*[number_bacteria]; //Create a list of bacteria names

    for (long i = 0; i<number_bacteria; i++) // For # of bacteria'
    {
        bacteria_name[i] = new char[20]; //add a new char[20] field to bacteria_name[i]
        fscanf(input_file, "%s", bacteria_name[i]); //Reads .txt and adds string of bacteria name to bacteria_name[i]
        strcat(bacteria_name[i], ".faa"); //Concatenates parameters
    }
    fclose(input_file);
}
```

Figure 1 – ReadInputFile()

The CompareAllBacteria() method uses this file to create and instantiate an array of the class type Bacteria() for each bacteria specified (see Figure 2). The Bacteria() constructor uses the name specified in the '.txt' file to locate the corresponding '.faa' file, the information stored in '.faa' file is then used to generate the Bacteria() object.

```
void CompareAllBacteria()
{
    Bacteria** b = new Bacteria*[number_bacteria]; //creates an array of the class Bacteria b[]
    for (int i = 0; i<number_bacteria; i++) //for the number of bacteria recognised in the text file...
    {
        printf("load %d of %d\n", i + 1, number_bacteria); //print to the console a loading message for the next line
        b[i] = new Bacteria(bacteria_name[i]); //create a new instance of the Bacteria class and add it to the previosuly created array b[]
    }
```

Figure 2 – CompareAllBacteria() – pt.1

A loading message is outputted to the console to represent when each instantiation of the Bacteria class is being processed (see Figure 3).

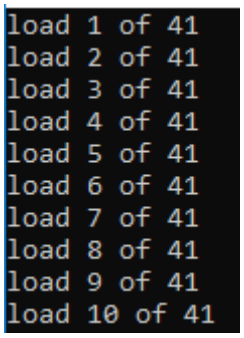


Figure 3 – Loading Message

Additionally the CompareAllBacteria() method proceeds to compare each created instantiation of the Bacteria() class against each other (see Figure 4). This comparison is computed using the CompareBacteria() method which returns a correlation/comparison value of type 'double'. The result of each comparison is then printed to the console (see Figure 5).

```
for (int i = 0; i<number_bacteria - 1; i++) //for the number of bacteria...
    for (int j = i + 1; j<number_bacteria; j++) //assess each bacteria against one another
    {
        printf("%2d %2d -> ", i, j);
        double correlation = CompareBacteria(b[i], b[j]); //compares the provided bacteria' and assigns the comparison value to 'correlation'
        printf("%.20lf\n", correlation); //prints the value of 'correlation' to the screen
    }
```

Figure 4 – CompareAllBacteria() – pt.2

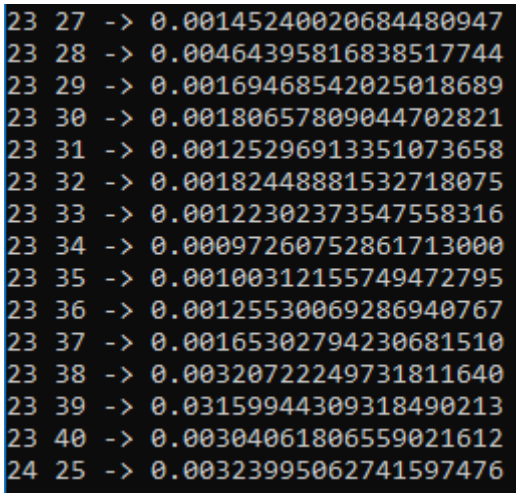


Figure 5 – Comparison Results

Once the results have been outputted to the console a timer value for how long the execution took is displayed to the user (see Figure 6 for application, and Figure 7 for output).

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

    Init(); //intialises parameters
    ReadInputFile(argv[1]); //reads the text file and adds bacteria names to an array []
    CompareAllBacteria(); //Creates and instantiates Bacteria() class instances and adds them to the array,
                          //Then compares all instances against each other and ouputs the results

    time_t t2 = time(NULL); //stops application timer
    printf("time elapsed: %d seconds\n", t2 - t1); //outputs elapsed time to user

    system("pause");

    return 0;
}
```

Figure 6 – Main Method

```
time elapsed: 41 seconds
Press any key to continue . . .
```

Figure 7 – Time Elapsed

2.3 – Granularity

The following table displays a chosen selection of loops and control flow constructs which have the potential for parallelisation. The information was collected through use of a performance profiler, the outputted report shows which functions and code examples are contributing the most towards overall CPU processing time.

Loop / Control Flow Construct	Called Within	Called @ Line	Granularity

`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]);`
`}`
	CompareAllBacteria()	268 - 272	77.10%

`for (int i = 0; i<number_bacteria - 1; i++)`
`  for (int j = i + 1; j<number_bacteria; j++)`
`  {`
`  printf("%2d %2d -> ", i, j);`
`  double correlation = CompareBacteria(b[i], b[j]);`
`  printf("%.20lf\n", correlation);`
`  }`
	CompareAllBacteria()	274-280	22.04%

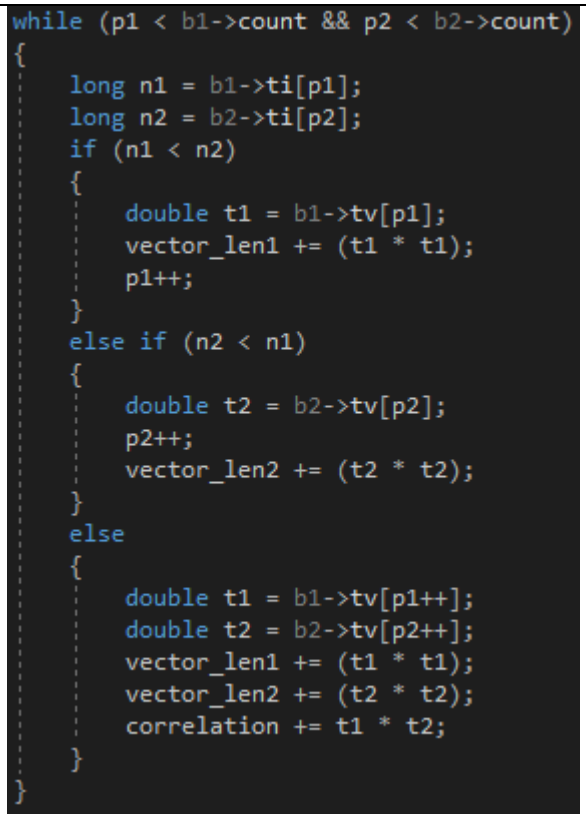
	
	`while (p1 < b1->count && p2 < b2->count)`
	`{`
	`  long n1 = b1->ti[p1];`
	`  long n2 = b2->ti[p2];`
	`  if (n1 < n2)`
	`  {`
	`  double t1 = b1->tv[p1];`
	`  vector_len1 += (t1 * t1);`
	`  p1++;`
	`  }`
	`  else if (n2 < n1)`
	`  {`
	`  double t2 = b2->tv[p2];`
	`  p2++;`
	`  vector_len2 += (t2 * t2);`
	`  }`
	`  else`
	`  {`
	`  double t1 = b1->tv[p1++];`
	`  double t2 = b2->tv[p2++];`
	`  vector_len1 += (t1 * t1);`
	`  vector_len2 += (t2 * t2);`
	`  correlation += t1 * t2;`
	`  }`
	`}`
		CompareBacteria()	223 - 247	23.43% (Collectively)

Figure 8 – Granularity Table

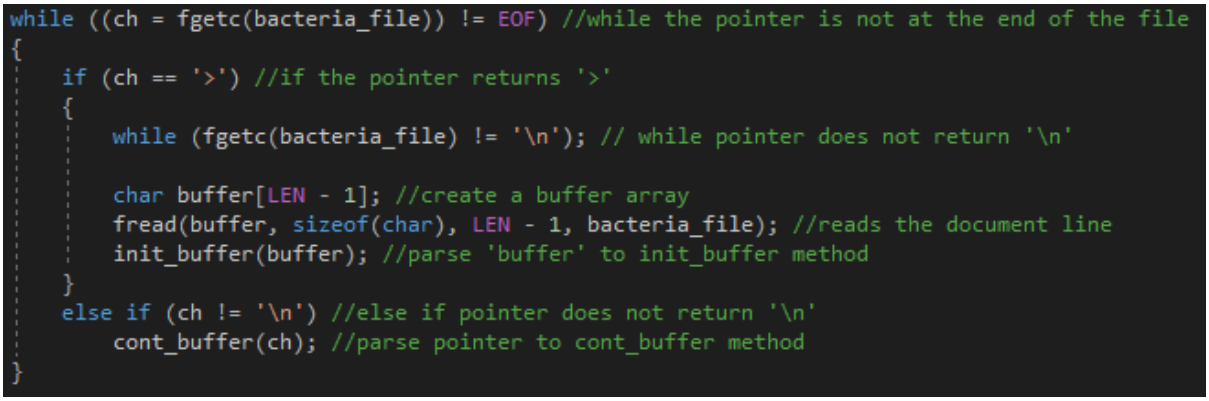
2.4 – Dependencies

This section details the occurring dependencies which exist in the application.

Control Dependencies:

A control dependency occurs when a statement can affect whether another statement executes (Queensland University of Technology, 2018). The following table shows the control dependencies which exist throughout the application.

Control Dependencies	Details


`while ((ch = fgetc(bacteria_file)) != EOF) //while the pointer is not at the end of the file`
`{`
`  if (ch == '>') //if the pointer returns '>'`
`  {`
`  while (fgetc(bacteria_file) != '\n'); // while pointer does not return '\n'`
`  char buffer[LEN - 1]; //create a buffer array`
`  fread(buffer, sizeof(char), LEN - 1, bacteria_file); //reads the document line`
`  init_buffer(buffer); //parse 'buffer' to init_buffer method`
`  }`
`  else if (ch != '\n') //else if pointer does not return '\n'`
`  cont_buffer(ch); //parse pointer to cont_buffer method`
`}`
	Function: Reads through parsed file
	Location: 112 – 124
	Parent: Bacteria()
	Dependence Impact on Runtime: Minimal
	Change? No

```
for (long i = 0; i<M; i++)
{
    double p1 = second_div_total[i_div_aa_number];
    double p2 = one_l_div_total[i_mod_aa_number];
    double p3 = second_div_total[i_mod_M1];
    double p4 = one_l_div_total[i_div_M1];
    double stochastic = (p1 * p2 + p3 * p4) * total_div_2;

    if (i_mod_aa_number == AA_NUMBER - 1)
    {
        i_mod_aa_number = 0;
        i_div_aa_number++;
    }
    else
        i_mod_aa_number++;

    if (i_mod_M1 == M1 - 1)
    {
        i_mod_M1 = 0;
        i_div_M1++;
    }
    else
        i_mod_M1++;

    if (stochastic > EPSILON)
    {
        t[i] = (vector[i] - stochastic) / stochastic;
        count++;
    }
    else
        t[i] = 0;
}
```

Function: Helps to create an instance of the Bacteria class from the parsed file

Location: 144 – 175

Parent: Bacteria()

Dependence Impact on Runtime: Moderate

Change? No

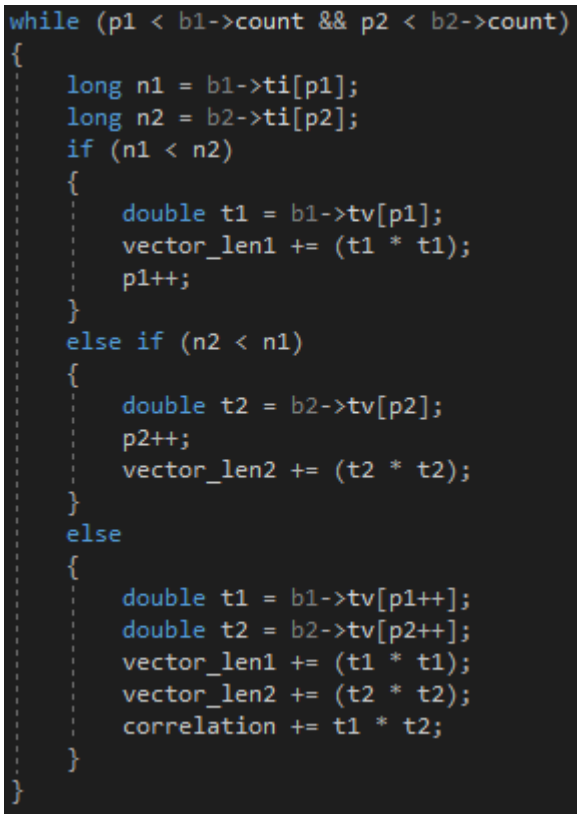
 	**Function:** Helps to conduct a comparison between the two parsed Bacteria() instances, b1 & b2. **Location:** 223 – 247 **Parent:** CompareBacteria() **Dependence Impact on Runtime:** Minimal **Change?** No

Figure 9 – Control Dependencies

Data Dependencies:

A data dependency occurs where one statement refers to the same data as some other statement (Queensland University of Technology, 2018). The following table shows data dependencies which occur throughout the application,

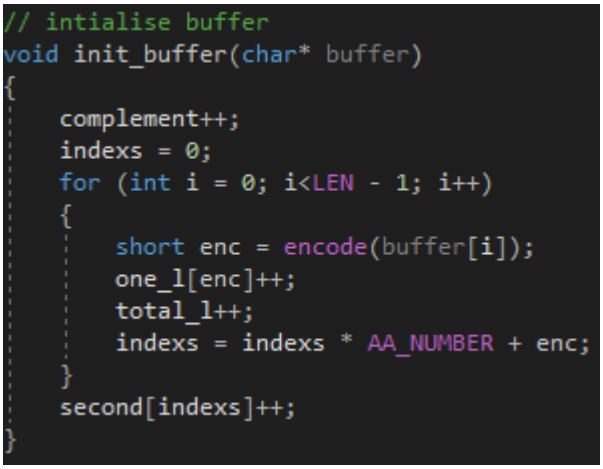
Data Dependencies	Details
 	**Function:** Initialises the buffer that assists in reading the bacteria (.faa) files **Dependency:** (`indexs = indexs * AA_NUMBER + enc;`) **Dependency Type:** Output Dependence **Location:** 75 - 87 **Parent:** Bacteria() **Dependence Impact on Runtime:** Moderate

Figure 10 – Data Dependencies

2.5 – Bottlenecks

When executed the applications average run time on the testing machine was 41 seconds. After the implemented parallelisation (see section 3.2) this runtime was reduced to 16 seconds. The sequential applications major bottlenecks are recognised as the two for-loops existing within the CompareAllBacteria() method (see section 2.3).

3.0 – Parallelisation

This section details how the above-mentioned sequential application was parallelised and the results of parallelisation.

3.1 – Tools & Techniques

This section showcases the tools and techniques used to parallelise the application.

Software

Visual Studio 2017 – Integrated Development Environment (IDE)

Compiler

Visual Studio 2017 C++ Compiler

Parallelisation Tools

OpenMP (Open Multi-Processing) – Multi-platform shared memory multiprocessing API

Techniques

Parallel-For – OpenMP function

Code modification

3.2 – Effective Parallelisation

Sections 2.5 (bottlenecks) and 2.3 (granularity) highlighted the areas where parallelisation would have most effect. These areas are recognised as the two for-loops existing in the CompareAllBacteria() method. The following shows how these loops were parallelised.

```
#pragma omp parallel
    {
#pragma omp 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]);
        }
    }
```

Figure 11 – Parallel Implementation pt1

```
#pragma omp parallel
    {
#pragma omp for
        for (int i = 0; i < number_bacteria - 1; i++)
            for (int j = i + 1; j < number_bacteria; j++)
            {
                printf("%2d %2d -> ", i, j);
                double correlation = CompareBacteria(b[i], b[j]);

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

Figure 12 – Parallel Implementation pt2

3.3 – Safe Parallelisation

To ensure that the parallelisation of the second for-loop (figure 12) did not affect the outputted data, code to check that the data created from the parallel version matched the sequential version was conducted. The following are excerpts from the parallel version showing how this was accomplished.

Step 1: Arrays were initialised before the loops to store the output for each.

```
double* storage_sequel = new double[number_bacteria*number_bacteria];
double* storage_parallel = new double[number_bacteria*number_bacteria];
```

Figure 13 – Storage Array Initialisation

Step 2: Storage was achieved by assigning the outputted data to the arrays.

```
storage_sequel[j-1] = CompareBacteria(b[i], b[j]);
```

Figure 14 – Storage Sequential

```
storage_parallel[j - 1] = CompareBacteria(b[i], b[j]);
```

Figure 15 – Storage Parallel

Step 3: The arrays were compared against one another to determine if they contained the same data outputs, the results of this test were displayed to the console.

```
for (int i = 0; i < (number_bacteria*number_bacteria); i++)
{
    if (storage_parallel[i] == storage_parallel[i]) {
        printf("Id %2d results match\n", i);
    }
    else
    {
        printf("Id %2d results do not match\n", i);
    }
}
```

Figure 16 – Results Comparison

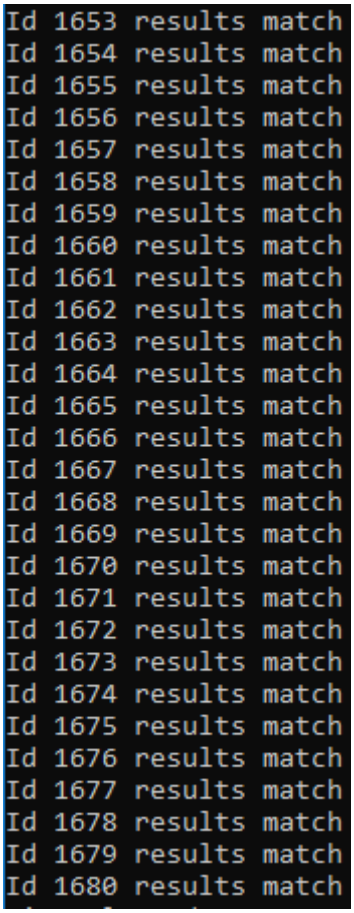


Figure 17 – Displayed Results

3.4 – Ineffective Parallelisation

The following for-loop was parallelised to determine if it would have an impact on runtime, although it was not successful for the application ran no faster. The code was changed back to the sequential version.

```
186   #pragma omp parallel
187           {
188   #pragma omp for
189               for (long i = 0; i < M; i++)
190               {
191                   if (t[i] != 0)
192                   {
193                       tv[pos] = t[i];
194                       ti[pos] = i;
195                       pos++;
196                   }
197               }
198           }
```

Figure 18 – Ineffective Parallelisation

3.5 – Application Amendment

The second for-loop as part of the CompareAllBacteria() method contained two printf statements when only one is necessary. This was changed to use only one printf statement, although the applications runtime showed no noticeable speedup. The change is seen below.

```
#pragma omp parallel
    {
#pragma omp for
        for (int i = 0; i < number_bacteria - 1; i++)
            for (int j = i + 1; j < number_bacteria; j++)
            {
                double correlation = CompareBacteria(b[i], b[j]);
                printf("%2d %2d -> %.20lf\n",i, j, correlation);
            }
    }
```

Figure 19 – Application Amendment

3.6 – Timing & Profiling

The following are the results of the timing and profiling for both the sequential program and the parallelised version.

Sequential

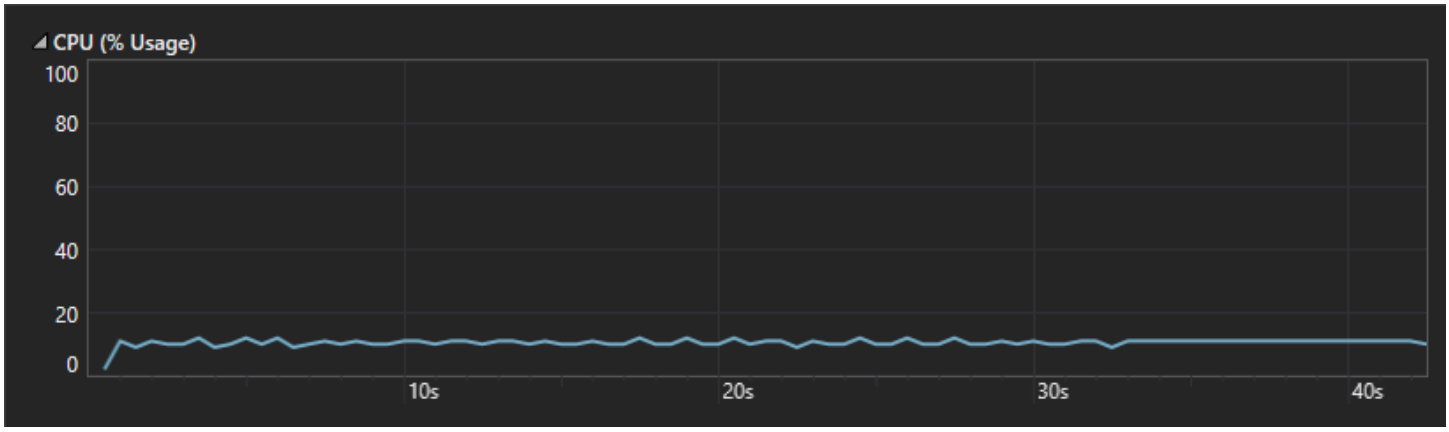


Figure 20 – Sequential Profiling

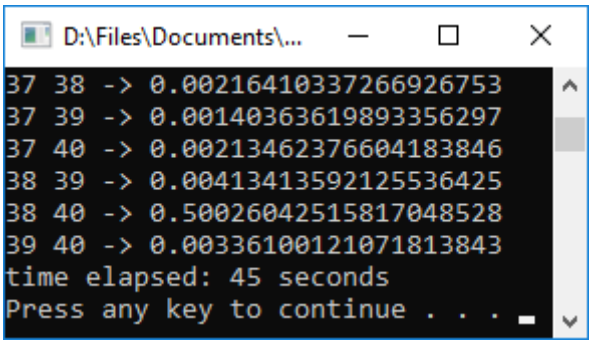


Figure 21 – Sequential Timing

Parallel

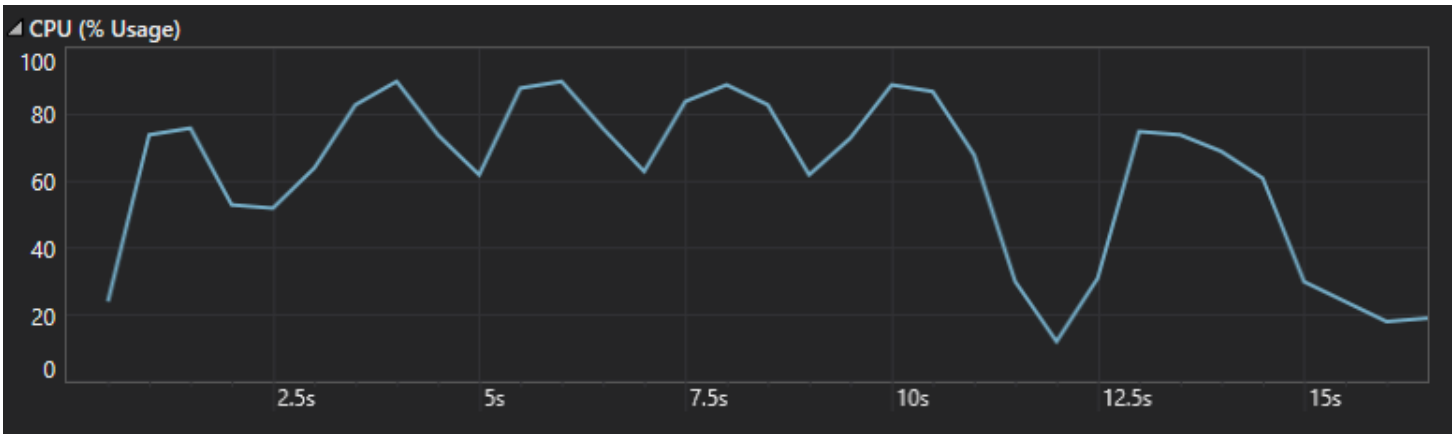


Figure 22 – Parallel Profiling

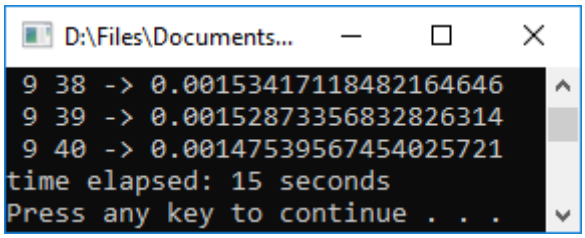


Figure 23 – Parallel Timing

Figure 22 clearly shows that the parallel program utilises a much greater use of CPU's available cores, and figure 23 shows a speedup difference of 30 seconds when using the parallel version.

3.7 – Speedup

Speedup is defined as the execution time of best sequential program divided by the execution time of the parallel program (Queensland University of Technology, 2018). As explained in section 3.6 the time taken for the sequential program was recognised as 45 seconds and the parallel time taken was 15 seconds. This means that the parallel version ran at 3 times faster than the sequential version.

To generate data for the speed up graph the parallel programs' runtime was tested when assigned an increasing number of threads. It was discovered that when assigned with more threads than the maximum amount of threads number (generated by the 'omp_get_max_threads()' function), faster results were achieved in some cases. The 'omp_get_max_threads()' function returned that the max number of threads was 8, although when the max number of threads was set to 9 the runtime performed at its best value of 14 seconds. Below are the results of this test.

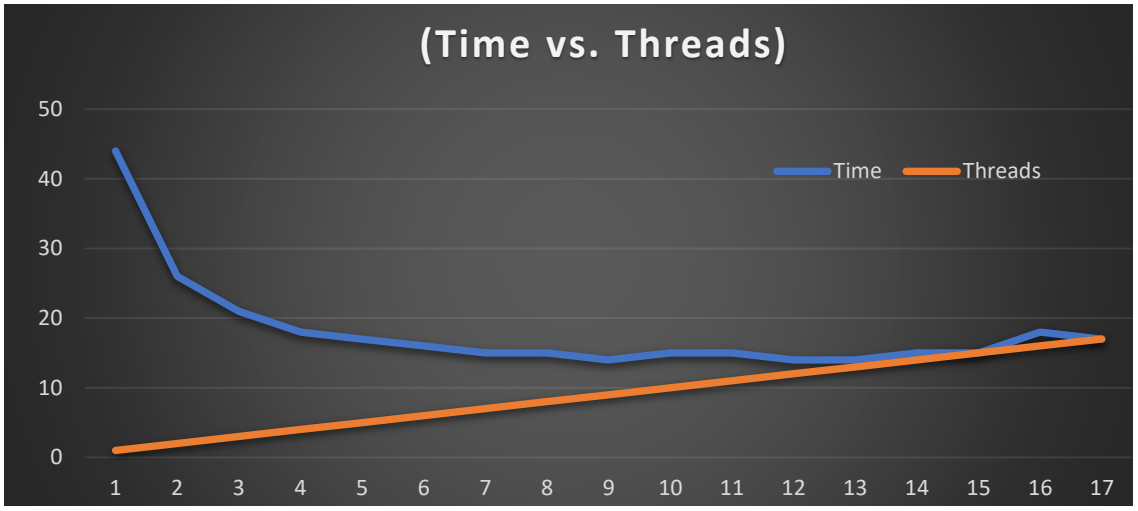


Figure 24 – Time vs. Threads

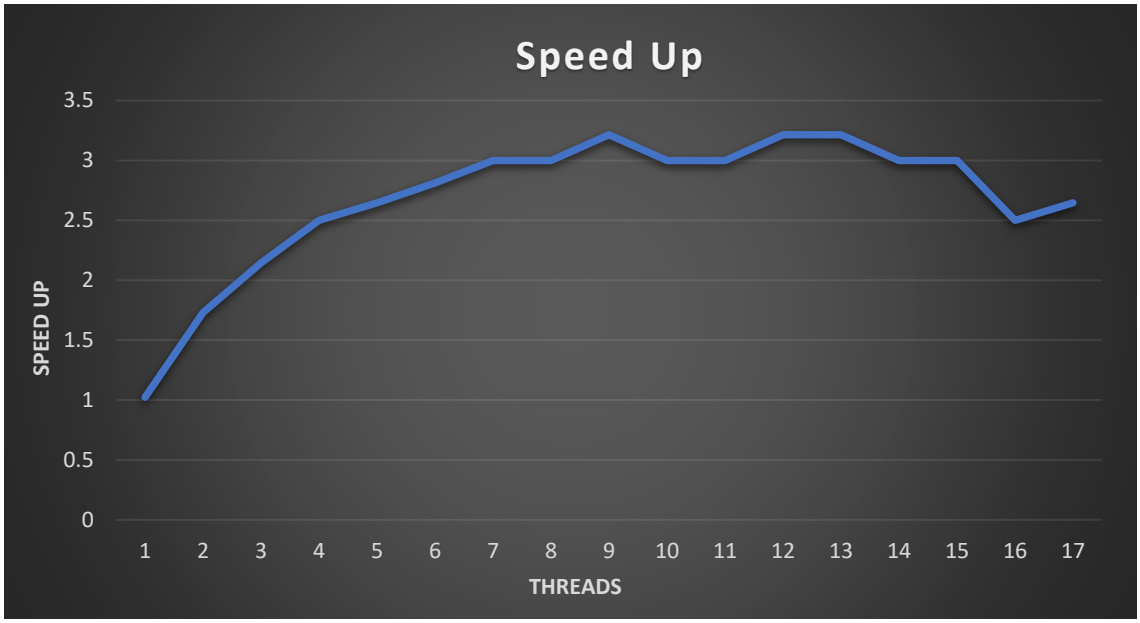


Figure 25 – Speed Up

As can been seen above the speed up test shows the parallelisation of the application has achieved a sub-linear speed up which is typical (Queensland University of Technology, 2018).

4.0 – Overcoming Barriers & Conclusion

Barriers

The challenges I faced when tackling this task was not due to not knowing how to parallelise the application, but rather getting the application to initially compile and run. It took me close to three 5-hour days of troubleshooting to get the project up and running. This was overcome by consulting with the lecturer and contacting my friends for assistance. I initially chose to attempt at parallelising the Digital Music Analysis project for it suited my musical aptitudes, although without many comments on what each section was achieving it was extremely difficult to understand.

The major barrier I faced when working on the Bacteria task was when working out how to compare the results of the sequential version against the parallel version (previously mentioned in section 3.3). I initially tried to export the data to a .csv file so that I could

import it into Microsoft Excel and use its' inbuilt 'compare cells' functionality. I then realised I could use the programs CompareBacteria() method and output the results to arrays. Which then made it easy to compare the results.

Another notable barrier was when the application performed better when assigning it more than it's 'maximum amount of threads', respective to the 'omp_get_num_threads()' function. I wasn't sure whether to start over the testing and make it part of the final parallelisation code, or to add it to the results. I chose the latter for it better describes the process of how I tackled the project.

Conclusion

This report has detailed my conducted parallelisation of an application which compares bacteria files against one another. A speed up of 3.214 times the sequential runtime was achieved through parallelisation of code existing in the CompareAllBacteria() function. I believe this is a significant improvement to the original sequential application. I believe there may be ways to either parallelise or change code which will improve runtime further, especially in the Bacteria() class and the CompareBacteria() method, although I did not tamper with these for I was afraid of disrupting the dependencies listed in section 2.4. I have learnt that parallelisation is easier than once thought and is an extremely useful tool that I will need to better grasp to obtain a leading industry edge over the ever-increasing body of developers.

4.0 – Bibliography

Queensland University of Technology. (2018, October). *Week 2 - Different Forms of Parallel Computing and the Process of Parallelization.* Retrieved from Blackboard: https://blackboard.qut.edu.au/bbcswebdav/pid-7470127-dt-content-rid-16734246_1/xid-16734246_1

Queensland University of Technology. (2018, October). *Week 3 - Methodology for Scalable Parallelization.* Retrieved from Blackboard: https://blackboard.qut.edu.au/bbcswebdav/pid-7470130-dt-content-rid-16734276_1/xid-16734276_1

5.0 – Parallel Hardware & Framework

Processor: Intel Core i7-7700HQ @ 2.80GHz,

Multicore: Quad Core

Age: Q1'17

Specification: 8 Threads, DDR4 Memory,

Levels of Cache: 6mb

Hyperthreading: Yes

GPU: Nvidia GeForce GTX 1050Ti

Language/Framework: C++ - Microsoft Visual Studio – Visual Studio Performance Profiler