

# **Parallelization of CV Tree**

In Partial Fulfilment of the Requirements  
for the Parallelization Project of (CAB401)  
High Performance and Parallel Computing

**By**  
**Chu Weng Law (n10116877)**

16<sup>th</sup> of September 2021

## Table of Contents

1.0	Introduction of CV Tree.....	3
2.0	Potential Improvement Available from Parallelism.....	5
2.1	Visual Studio 2019 Performance Profiler .....	5
2.2	Intel VTune Profiler.....	6
2.3	Bacteria constructor function .....	6
2.4	CompareBacteria() function.....	8
3.0	Optimising CV Tree.....	10
3.1	Replacing redundant code in the original program .....	10
3.2	Loop transformation.....	11
4.0	Parallelisation Design and Application.....	12
4.1	OpenMP .....	12
4.2	Parallel construct directives .....	12
	<b>4.1.1 parallel for clause</b> .....	12
	<b>4.1.2 schedule clause</b> .....	12
	<b>4.1.3 num_threads clause</b> .....	13
4.3	CPU time after parallelism.....	14
5.0	Analysis of the Results.....	16
6.0	Software and Tools .....	18
7.0	Limitations and Difficulties .....	19
7.1	Fine grain parallelism.....	19
7.2	schedule clause.....	19
7.3	Compilation error.....	19
8.0	Conclusion .....	21
9.0	Appendix.....	22
9.1	CPU Time (Bacteria constructor function) .....	22
9.2	CPU Time (CompareBacteria() function).....	23
9.3	CPU Utilization (Original CV Tree).....	23
9.4	CPU Utilization (Parallelized CV Tree) .....	24
9.5	Full test results on QUT HPC .....	26

## 1.0 Introduction of CV Tree

CV Tree is a sequential program that is available on QUT Blackboard that is built with C++ tool and can be compiled with Microsoft Visual Studio 2019. The purpose of the program is to compute genome similarity using Frequency Vectors. It consists of two C++ files which are tidy.cpp and improved.cpp. The later file is the improved version of tidy file which gives a more human readable and better sequential computation of the CV Tree. Since we need to compare the parallelization performance with best sequential program when developing parallel computing, we will investigate the improved.cpp as our main source file and implement parallelism into it then report on the improvement as a result.

CV Tree has a required argument which reads a given text file named “list.txt” that stores the numeric number of text-based format for representing genomic sequences, also known as FASTA format files and then one genome name per line as the data file respectively. The FASTA files must have a “.faa” extension and the program will automatically search through the working directory for all the listed files.

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

    Init(); 1
    ReadInputFile("list.txt"); 2
    CompareAllBacteria(); 3

    time_t t2 = time(NULL);
    printf("time elapsed: %lld seconds\n", t2 - t1);
    return 0;
}
```

Figure 1 CV tree main function

Referring to the figure above, CV Tree consists of three major components. In State 1, it initializes variables for the protein polymers which are k-mers, (k-1)-mers, and (k-2)-mers. The k-value is hardcoded in the program as six representing the length of the polymer and the number of all possible combination of k-mers is the number of amino acids to the power of k-value which is  $20^6$  and  $20^5$  for (k-1)-mers and so on.

For State 2, the program reads an input text file named “list.txt” and then initialize new string array of bacteria names.

Most of the expensive calculations and overheads happens in State 3. It creates all the bacteria class objects which is initialized and stored in a one-dimensional array and then computes the comparison between each bacterium. The comparison between two bacteria is calculated in a nested for loop and then the index of the compared bacteria along with their correlation value is printed onto the console screen. In CompareAllBacteria() function, the comparison is repeated with two for loops where the outer loop represents all bacterium but itself and the inner loop

represents the exclusion of inverse comparison from two bacterium. In short, for N number of bacteria objects,  $N(N-1)/2$  comparisons are made because a bacterium excludes self-comparison and the reverse comparison with other bacteria that has already been made.

As for the genome similarity between two bacteria, it is represented by the correlation between the bacteria using T-score with stochastic method. After the creation of a bacteria, the frequency count of (k-2)-mers, (k-1)-mers and k-mers are stored in the arrays respectively and the variables required for calculation of the correlation are also initialized.

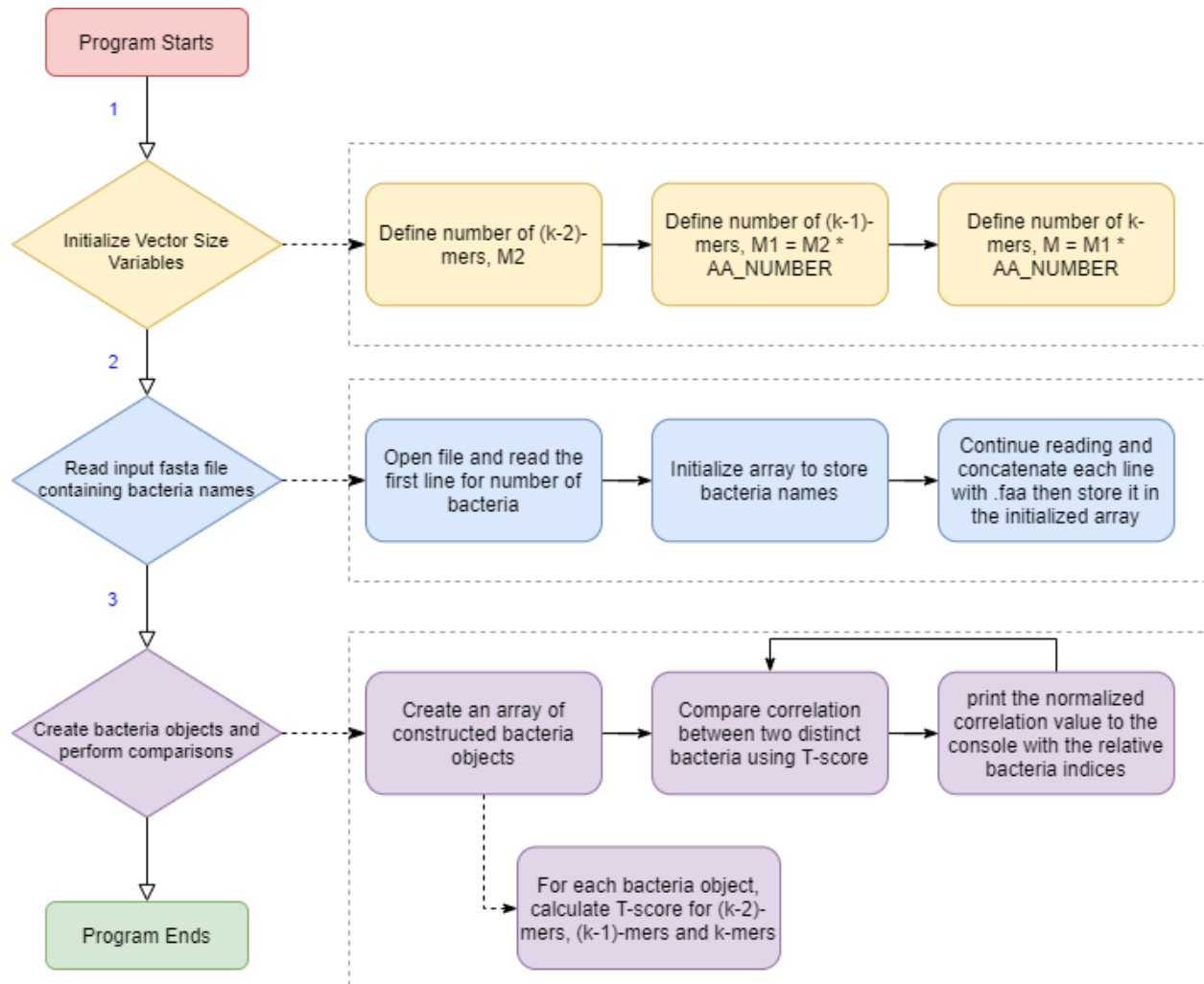
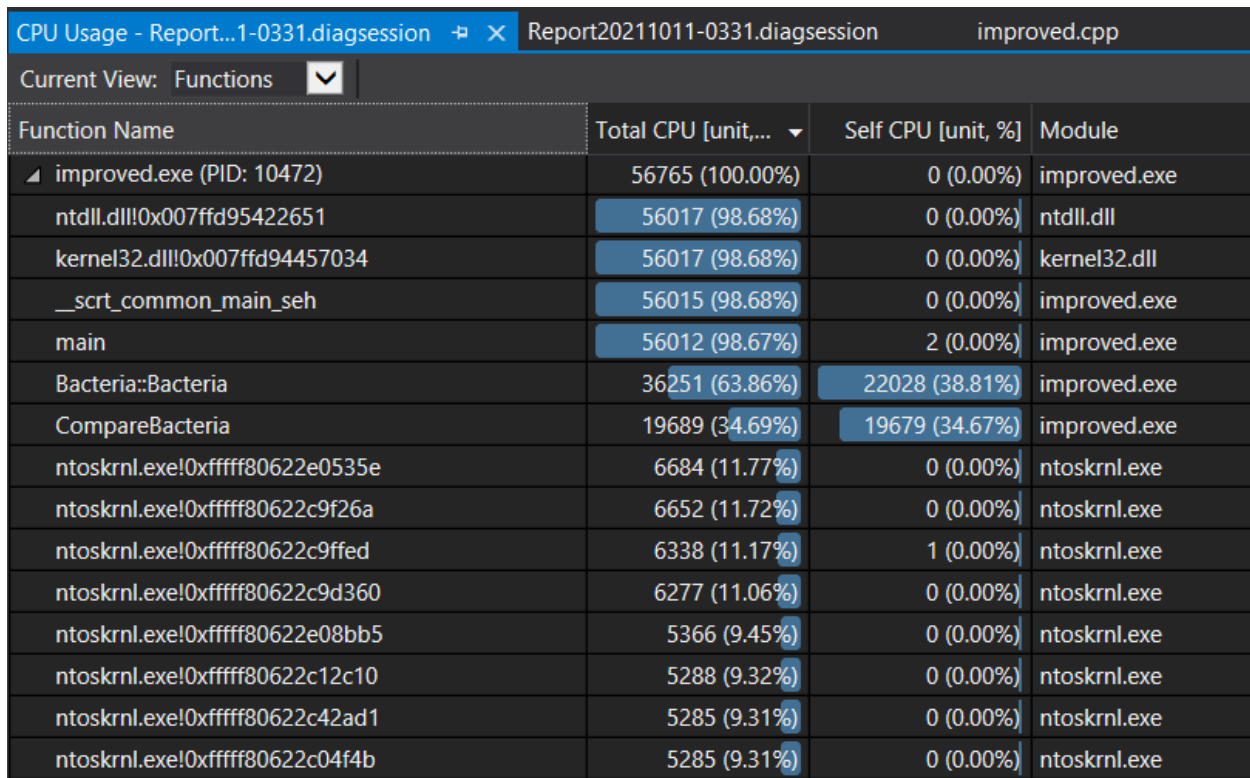


Figure 2 Flow diagram of CV Tree program

## 2.0 Potential Improvement Available from Parallelism

### 2.1 Visual Studio 2019 Performance Profiler

Before we begin to implement parallelism to the program, first we need to identify which section or function of it consumes the most CPU usage. Then we can check if that particular section is independent and parallelizable as well as comparing the performance improvement at the end. The performance of CV Tree program is analyzed using Visual Studio 2019 Performance Profiler. It provides basic performance information including CPU usage, GPU usage, Memory usage etc. The CPU Usage performance tool shows the CPU time and percentage spent executing code in CV Tree.



CPU Usage - Report...1-0331.diagsession Report20211011-0331.diagsession improved.cpp			
Current View: Functions			
Function Name	Total CPU [unit,...	Self CPU [unit, %]	Module
improved.exe (PID: 10472)	56765 (100.00%)	0 (0.00%)	improved.exe
ntdll.dll!0x007ffd95422651	56017 (98.68%)	0 (0.00%)	ntdll.dll
kernel32.dll!0x007ffd94457034	56017 (98.68%)	0 (0.00%)	kernel32.dll
__scrt_common_main_seh	56015 (98.68%)	0 (0.00%)	improved.exe
main	56012 (98.67%)	2 (0.00%)	improved.exe
Bacteria::Bacteria	36251 (63.86%)	22028 (38.81%)	improved.exe
CompareBacteria	19689 (34.69%)	19679 (34.67%)	improved.exe
ntoskrnl.exe!0xfffff80622e0535e	6684 (11.77%)	0 (0.00%)	ntoskrnl.exe
ntoskrnl.exe!0xfffff80622c9f26a	6652 (11.72%)	0 (0.00%)	ntoskrnl.exe
ntoskrnl.exe!0xfffff80622c9ffed	6338 (11.17%)	1 (0.00%)	ntoskrnl.exe
ntoskrnl.exe!0xfffff80622c9d360	6277 (11.06%)	0 (0.00%)	ntoskrnl.exe
ntoskrnl.exe!0xfffff80622e08bb5	5366 (9.45%)	0 (0.00%)	ntoskrnl.exe
ntoskrnl.exe!0xfffff80622c12c10	5288 (9.32%)	0 (0.00%)	ntoskrnl.exe
ntoskrnl.exe!0xfffff80622c42ad1	5285 (9.31%)	0 (0.00%)	ntoskrnl.exe
ntoskrnl.exe!0xfffff80622c04f4b	5285 (9.31%)	0 (0.00%)	ntoskrnl.exe

Figure 3 improved.cpp CPU Usage diagnostic session using Visual Studio 2019 Performance Profiler

Figure 3 shows the diagnostic report of improved version of the CV Tree application, which is sorted by Total CPU from highest to lowest. Total CPU indicates how much work was done by the function and any functions called by it whereas Self CPU indicates how much work was done by the code in the function body, excluding the work done by functions that were called by it and here we focus more on self-CPU. Based on the figure above, the Bacteria constructor function consumes the highest amount of CPU time in the entire application with 38.81% and the CompareBacteria() function consumes the second most with 34.67%.

## 2.2 Intel VTune Profiler

We can also analyze the program with Visual Studio integrated Intel VTune Profiler to ensure the performance analysis is accurate and reliable. Intel VTune Profiler provides information such as the CPU time in percentage utilised per function and that per thread to identify the most time-consuming functions and drill down to see time spent on each line of source code using its hotspots analysis. Intel VTune Profiler will be used as the main CPU Time analysis in the rest of the report.

Function / Call Stack	CPU Time	Module	Function (Full)
Bacteria::Bacteria	57.8%	improved.exe	Bacteria::Bacteria(char *)
CompareBacteria	20.6%	improved.exe	CompareBacteria(class Bacteria *, class Bacteria *)
free_base	12.3%	ucrtbase.dll	free_base
func@0x180001980	7.7%	VCRUNTIME140.dll	func@0x180001980
_stdio_common_vfprintf	1.1%	ucrtbase.dll	_stdio_common_vfprintf
fgetc	0.4%	ucrtbase.dll	fgetc
exit	0.0%	ucrtbase.dll	exit
fclose	0.0%	ucrtbase.dll	fclose
fopen_s	0.0%	ucrtbase.dll	fopen_s

Figure 4 Intel VTune Profiler in VS 2019 on improved.cpp using Basic Hotspots Analysis

Intel VTune Profiler Hotspots shows that CompareBacteria() function yields the most CPU time with 57.8% and Bacteria constructor function yields the second most CPU time with 20.6%. Both of these functions utilizes almost 80% of the CPU time of the entire application so we can focus more on them when implementing parallelism in the program.

## 2.3 Bacteria constructor function

Source Line	Source	CPU Time: Total	CPU Time: Self
115	for (int i=0; i<AA_NUMBER; i++)		
116	one_l_div_total[i] = (double)one_l[i] / total_l;		
117			
118	double* second_div_total = new double[M1];	0.3%	0s
119	for (int i=0; i<M1; i++)	0.0%	0.062s
120	second_div_total[i] = (double)second[i] / total_plus_complement;	1.1%	1.549s

Figure 5 First two for-loops example code in the Bacteria constructor function body

The Bacteria constructor function consists of four for-loops in total. The first 2 for loops at line 115 and 119 do not have any dependency hence they are safe to be parallelized, but they are not worth enough to parallelize as they only consume low amount of CPU time. It may create unnecessary additional overhead for the loops if we parallelize them.

Iteration	Read	Write
0	i_mod_aa_number, i_div_aa_number, i_mod_M1, i_div_M1, count	(i_mod_aa_number and/or i_div_aa_number), (i_mod_M1 and/or i_div_M1), count
1	i_mod_aa_number, i_div_aa_number, i_mod_M1, i_div_M1, count	(i_mod_aa_number and/or i_div_aa_number), (i_mod_M1 and/or i_div_M1), count
2	i_mod_aa_number, i_div_aa_number, i_mod_M1, i_div_M1, count	(i_mod_aa_number and/or i_div_aa_number), (i_mod_M1 and/or i_div_M1), count
...		

Table 1 Iteration dependency flow of 3rd for-loop of Bacteria constructor function

The CPU consumptions of the function body are more significant at the latter two for-loops. The CPU time figure for this part is attached in the Appendix 9.1 for the ease of user readability. There are flow dependencies between the variables in the for-loops (Refer Table 1) thus they are not safe to be parallelized. Table above shows all the variables with dependency in third for-loop and their iteration flow. There is also a pos variable that has dependency in the fourth for-loop.

```

125  for(long i=0; i<M; i++)
126  {
127      ...
148
149      if (stochastic > EPSILON)
150      {
151          t[i] = (vector[i] - stochastic) / stochastic;
152          count++;
153      }
154      else
155          t[i] = 0;
156      }
157
158      ...
164
165      int pos = 0;
166      for (long i=0; i<M; i++)
167      {
168          if (t[i] != 0)
169          {
170              tv[pos] = t[i];
171              ti[pos] = i;
172              pos++;
173          }
174      }

```

Figure 6 3rd and 4th for-loop of Bacteria constructor function

There is a redundant code that can be removed in this constructor function which could possibly further optimizing the code. The code snippet in Figure 7 demonstrates the example of Bacteria constructor stores the T-score of each k-mers in an array with size of  $20^k$  in a for-loop at line 125. When the stochastic value is lesser than epsilon, its associated T-score is set to zero. However, the constructor also filters the initialized array in another for-loop. Since only the non-zero T-scores can be accessed by other functions, it is unnecessary complicated and redundant to have a filtering process. It can be removed with a condition that ensures that only non-zero T-scores are stored in a vector in the for-loop which stores the T-score.

The Bacteria constructor objects are stored in a one-dimensional array in the for-loop of CompareAllBacteria() function. They are also safe to be parallelized since they do not have any dependency on the bacteria array variable.

## 2.4 CompareBacteria() function

```

253 void CompareAllBacteria()
254 {
255     Bacteria** b = new Bacteria*[number_bacteria];
256     for(int i=0; i<number_bacteria; i++)
257     {
258         printf("load %d of %d\n", i+1, number_bacteria);
259         b[i] = new Bacteria(bacteria_name[i]);
260     }
261
262     for(int i=0; i<number_bacteria-1; i++)
263     {
264         for(int j=i+1; j<number_bacteria; j++)
265         {
266             printf("%2d %2d -> ", i, j);
267             double correlation = CompareBacteria(b[i], b[j]);
268             printf("%.20lf\n", correlation);
269         }
270     }
271 }

```

Figure 7 CompareAllBacteria() function code snippet

CompareBacteria() function is called by the CompareAllBacteria() function in a nested for-loop at line 266 for a correlation value between two distinct Bacteria objects. The nested for-loop has no dependency issue and therefore it is safe to parallelize both inner and outer loop. If parallelism were to be implemented in this section of the code, the order that this code section executes matters. Since the nested for-loop is only printing the result of the correlation, the order of execution does not matter because the correlation value is printed along with the bacteria indices, hence it will not affect the accuracy of the program.

CompareBacteria() function calculates the correlation between two distinct bacteria objects. The correlation value is the normalised sum of all the T-scores multiplied between the two distinct bacteria objects for each type of k-mers as defined in the code with **correlation** +=  $t_{b1,k_i} * t_{b2,k_i}$ , where  $i = \{0 \dots 20^k - 1\}$  and the value is defined as data type of double.

The T-scores of a Bacteria object of different type of k-mers is calculated and the values are stored in an array if they are larger than epsilon during initialization of the Bacteria object. The size of the T-scores array depends on their respective bacteria objects and the T-scores array between two distinct Bacteria objects may not align. For example,  $T_{b1,i}$  and  $T_{b2,i}$  may not correspond to the same type of k-mers where  $i$  is the index of the T-score. The original application uses three while-loops to align the T-scores together for calculating correlation value.

The while-loops in the CompareBacteria() function spent 20.6% of CPU time as shown in Figure 4, so it will be beneficial to parallelize it, refer Appendix 9.2 for more details of CPU time in the



function. Again, we need to ensure that there is not dependency on the variables in the loops before we implement parallelism to it. There is a flow dependency on the variables **vector\_len1**, **vector\_len2** and **correlation** in the first loop as well as a control dependency on the variables **p1** and **p2** for all three while-loops hence the three while-loops are not safe to be parallelized.

```
205 double CompareBacteria(Bacteria* b1, Bacteria* b2)
206 {
207     ...
237     while (p1 < b1->count)
238     {
239         long n1 = b1->ti[p1];
240         double t1 = b1->tv[p1++];
241         vector_len1 += (t1 * t1);
242     }
243     while (p2 < b2->count)
244     {
245         long n2 = b2->ti[p2];
246         double t2 = b2->tv[p2++];
247         vector_len2 += (t2 * t2);
248     }
249
250     return correlation / (sqrt(vector_len1) * sqrt(vector_len2));
251 }
```

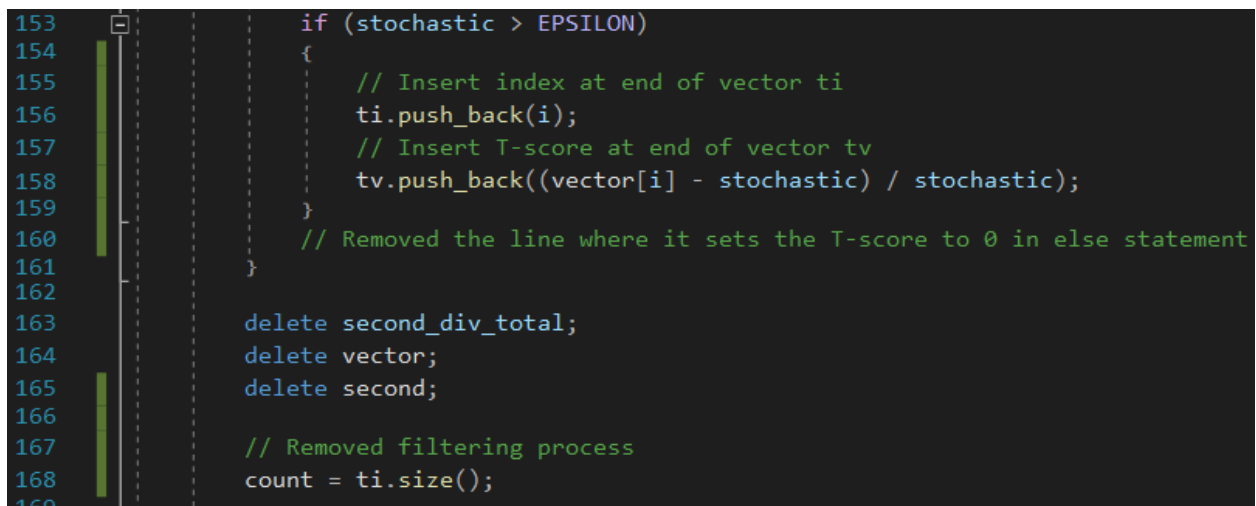
Figure 8 2nd and 3rd while-loops in CompareBacteria() function

There are 2 variables initialised, **n1** and **n2**, but were not used in CompareBacteria() function at line 239 and 245 so these two lines will also be removed to preserve computational power.

### 3.0 Optimising CV Tree

The original application completes in approximately 50.5 seconds. This section discussed the actions that I have taken to reduce the execution time of the original sequential program to 21.7 seconds using optimization. Note that the time elapsed does not include the time taken to load the input files.

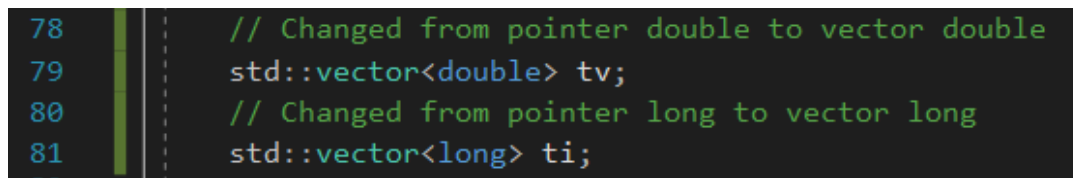
#### 3.1 Replacing redundant code in the original program

A screenshot of a code editor showing a C++ code snippet. The code is in a dark-themed editor with line numbers on the left. The code is as follows:

```
153     if (stochastic > EPSILON)
154     {
155         // Insert index at end of vector ti
156         ti.push_back(i);
157         // Insert T-score at end of vector tv
158         tv.push_back((vector[i] - stochastic) / stochastic);
159     }
160     // Removed the line where it sets the T-score to 0 in else statement
161 }
162
163 delete second_div_total;
164 delete vector;
165 delete second;
166
167 // Removed filtering process
168 count = ti.size();
169
```

Figure 9 Code snippet of the changes made in Bacteria constructor function

The original program has some redundant code in Bacteria constructor function as discussed in Section 2.3 and can be replaced with a slightly more efficient method of implementing it. The temporary array to store the T-scores increases overhead and wasting resources by occupying large memory space and also causes an increase in computational time due to the filtering process. It can be removed and stores the T-scores in a vector directly to eliminate the process of iterating through  $20^k$  of array size for filtering.

A screenshot of a code editor showing a C++ code snippet. The code is in a dark-themed editor with line numbers on the left. The code is as follows:

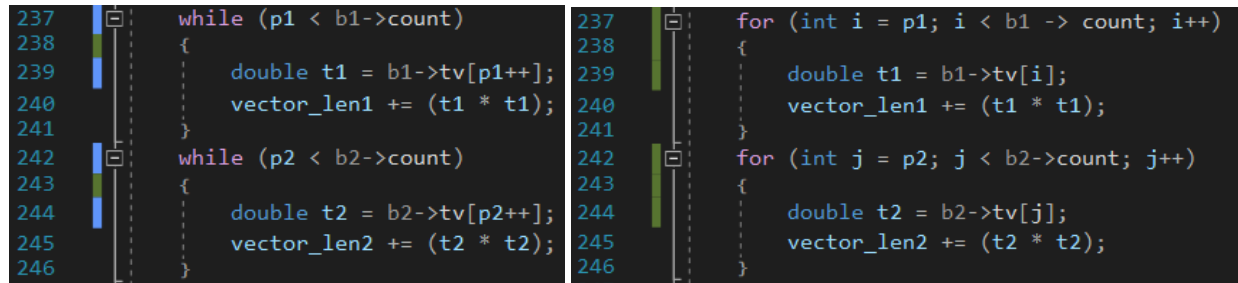
```
78 // Changed from pointer double to vector double
79 std::vector<double> tv;
80 // Changed from pointer long to vector long
81 std::vector<long> ti;
```

Figure 10 Example of vector arrays

Here we use vector instead of pointer arrays because vector has better cache locality when it comes to accessing elements in an array.

### 3.2 Loop transformation

The Auto-Vectorizer in Visual Studio 2019 analyses loops in the code and uses the vector registers and instructions on the target computer to execute them to improve the performance of the program. The Auto-Vectorizer is enabled by default. Loop unrolling is a space-time trade-off technique of optimizing a program's execution speed at the expense of its binary size and can be performed manually.



```
237 while (p1 < b1->count)
238 {
239     double t1 = b1->tv[p1++];
240     vector_len1 += (t1 * t1);
241 }
242 while (p2 < b2->count)
243 {
244     double t2 = b2->tv[p2++];
245     vector_len2 += (t2 * t2);
246 }
```

```
237 for (int i = p1; i < b1->count; i++)
238 {
239     double t1 = b1->tv[i];
240     vector_len1 += (t1 * t1);
241 }
242 for (int j = p2; j < b2->count; j++)
243 {
244     double t2 = b2->tv[j];
245     vector_len2 += (t2 * t2);
246 }
```

Figure 11 Before and After loop transformation in CompareBacteria() function

The two while-loops in the CompareBacteria() function are transformed into for-loops for auto vectorization while preserving the temporal sequence of all dependencies at the same time. The goal is to increase the program speed by reducing or eliminating instructions that control the while loop such as the "end of loop" **t1**, **t2**, **vector\_len1** and **vector\_len2** operations on each iteration. It may seem to be unnecessary since it only consumes small amount of CPU time in exchange for small amount of speedup, but it does not create extra overhead for it, so it is a good practice to do so.

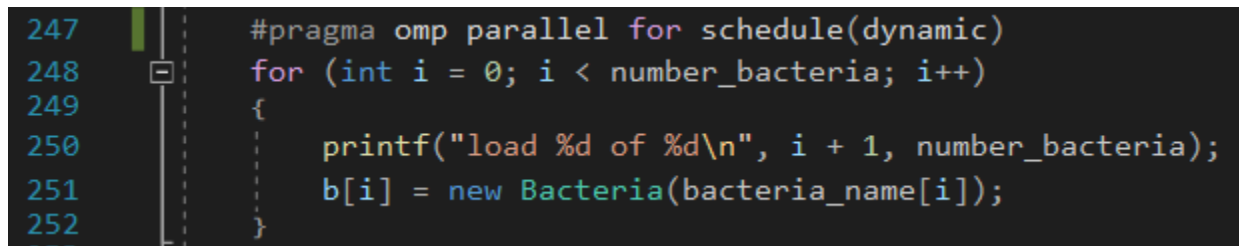
## 4.0 Parallelisation Design and Application

### 4.1 OpenMP

In parallel computing, in order to achieve the best parallel performance, the best balance between load and communication overhead needs to be found. For the optimized version of the application, I have only implemented coarse grain parallelism which provides low communication and synchronization overhead as fine grain parallelism has been tested that it is unable to provide more speedup due to the increase in overhead. The parallelism is only implemented in the CompareAllBacteria() function which initializes Bacteria objects and calls for the CompareBacteria() function as discussed in the earlier section. The OpenMP library will be used for this implementation.

### 4.2 Parallel construct directives

OpenMP provides a “#pragma omp parallel for” directive which is used to fork additional threads to carry out the work enclosed in the construct in parallel. For this case, it is used to divide the loop iterations’ tasks among some number of threads which we can manually set so that they can run in parallel.



```
247 #pragma omp parallel for schedule(dynamic)
248 for (int i = 0; i < number_bacteria; i++)
249 {
250     printf("load %d of %d\n", i + 1, number_bacteria);
251     b[i] = new Bacteria(bacteria_name[i]);
252 }
```

Figure 12 Parallelizing first for-loop in CompareAllBacteria() function

#### 4.1.1 parallel for clause

The relationship between the size of input files for the application and the time taken to initialize and create the bacteria object is linear. If it has a longer sequence, the longer the time it takes to create a bacteria object. The “parallel for” clause creates a new group of threads, splits that group to handle different portions of the loop.

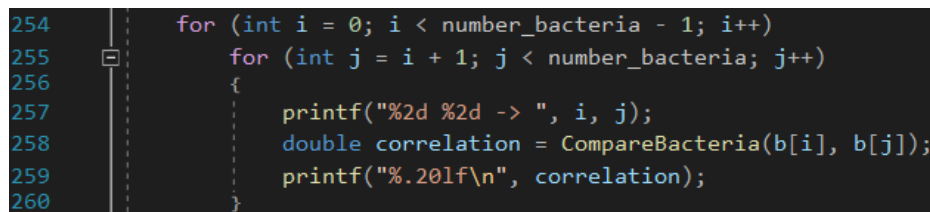
#### 4.1.2 schedule clause

The workload in the thread for each iteration is inconsistent because all the fasta files have different sizes and they are unordered when reading them from the list.txt in terms of file size. The dynamic scheduler clause is chosen for this case of the for-loop with the iterations requiring varying and unpredictable amounts of work for load balancing so that no thread waits at the parallel region

barrier for longer than it takes another thread to execute its final iteration as shown in Figure 12 at line 247.

#### 4.1.3 num\_threads clause

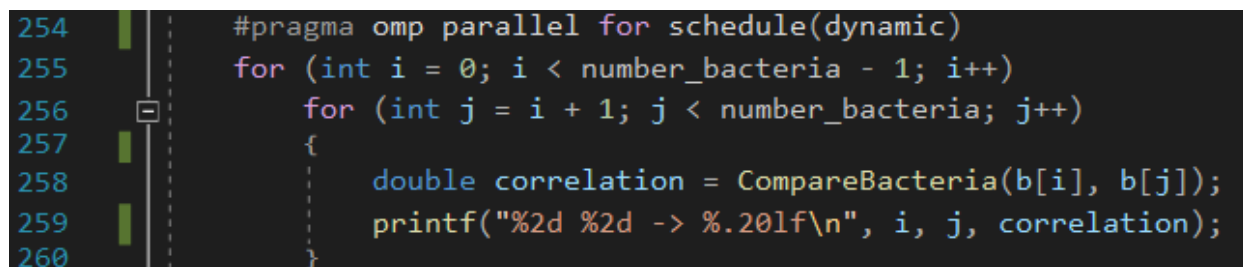
The num\_threads clause has the same functionality as the omp\_set\_num\_threads function and it can be added to a parallel directive to specify the number of threads that should execute the following block. However, there may be system-defined limitations on the number of threads that a program can start so we should carefully select the number of threads before utilizing the clause. If num\_threads clause was not specified, the number of threads requested is implementation-defined, using the maximum number of threads that is equal to the number of logical cores in the system. The number of threads for the CompareAllBacteria() function is left unspecified to minimize parallelism overhead by utilizing only one thread per logical core and we can define the number of logical cores we wish to utilize in the QUT HPC when running for the final testing later.



```
254     for (int i = 0; i < number_bacteria - 1; i++)
255     {
256         for (int j = i + 1; j < number_bacteria; j++)
257         {
258             printf("%2d %2d -> ", i, j);
259             double correlation = CompareBacteria(b[i], b[j]);
260             printf("%.20lf\n", correlation);
261         }
262     }
```

Figure 13 Before parallelization of the nested for-loop in CompareAllBacteria() function

As shown in Figure 13, the original program uses a nested for-loop for accessing two distinct bacteria objects and compare them for a correlation value then print it to the console along with their respective indices. Despite having both the inner and outer loop to be safe to parallelize, only the outer loop is parallelized following the coarse grain parallelism technique.



```
254     #pragma omp parallel for schedule(dynamic)
255     for (int i = 0; i < number_bacteria - 1; i++)
256     {
257         for (int j = i + 1; j < number_bacteria; j++)
258         {
259             double correlation = CompareBacteria(b[i], b[j]);
260             printf("%2d %2d -> %.20lf\n", i, j, correlation);
261         }
262     }
```

Figure 14 After parallelization of the nested for-loop in CompareAllBacteria() function

The parallelism is achieved by using the “#pragma omp parallel for” directive with “schedule(dynamic)” clause on top of the loop. Similar to the previous case, dynamic scheduler is chosen because the T-score vector is unpredictable and has various sizes for each of the bacteria object. The printf() call is shorten from two to one to avoid synchronization issues so there will be no need for extra synchronization implementation such as mutex locks.

Now, the order of the output printed to the console is now dependent on thread execution. For example, when Thread A is writing the correlation value to the console, Thread B will be waiting until the end of the task of Thread A before it can perform the writing. This is a critical issue because it will cause a delay in execution time. However, it can be overcome by storing the correlation value in a row major one-dimensional array for minimal cache misses and wait time while maintaining the order of the printing.

```

254 // Fixed size array of N(N-1)/2
255 double* correlation = new double[number_bacteria * (number_bacteria - 1) / 2];
256
257 // Row major conversion
258 #define correlation(i,j) correlation[number_bacteria * i + j - (i + 1)*(i + 2) / 2]
259
260 #pragma omp parallel for schedule(dynamic)
261 for (int i = 0; i < number_bacteria - 1; i++)
262     for (int j = i + 1; j < number_bacteria; j++)
263     {
264         correlation(i, j) = CompareBacteria(b[i], b[j]);
265     }
266
267 for (int i = 0; i < number_bacteria - 1; i++)
268     for (int j = i + 1; j < number_bacteria; j++)
269     {
270         printf("%2d %2d -> %.20lf\n", i, j, correlation(i, j));
271     }

```

Figure 15 Conversion to row major array to store the results

Since we knew the total number of comparisons that will be made by the program on the Bacteria objects is  $N(N-1)/2$  from Section 1.0, the correlation values will be stored in a new row major fixed size array at line 264 in Figure 15 and we can use another nested for-loop to iterate through the stored values and then print each of the result to the console between line 267 and 271.

There is a slight disadvantage in the conversion of memory array to row major array. The increase in number of Bacteria objects due to the allocation of memory will lead to a decrease in efficiency of the application. When it comes to dealing with big data especially involving millions of numbers of sequences fasta file as well as the order of the comparisons between Bacteria objects is not concerned, it may be better to keep the original implementation of storing the correlation vectors.

### 4.3 CPU time after parallelism

The parallelism achieved for the application reaches 74.8%. Since only the redundant code region is modified in the Bacteria constructor function and filtering process is removed, the average effective time by utilization was improved from poor to ok and ideal. As for the CompareBacteria() function where parallelism pragma omp is implemented, it has an ideal CPU time utilization. The application is running at the maximum number of threads, giving an ideal efficiency during most of the CPU time. Despite having a small percentage of time when the application has poor efficiency, there is only small amount of spin time and overhead time. The full detail of effective CPU time by utilization is included in Appendix 9.5.

CompareBacteria() function still has the second highest CPU utilization, 33.0% of CPU time compared to 20.6% in the original application. Small amount of CPU utilization is poor or ok may possibly because of the imbalance in load as the size of the bacteria sequences varies so the inconsistency in computation and printing of the result will causes additional overhead. Since dynamic scheduler clause is used in pragma omp directive, the unpredicted and poorly distributed size of the Bacteria input files affects the overall performance of the program. The largest input file (416 KB) is at least 34 times larger than the smallest file (12 KB).

Bacteria constructor function has reduced CPU time from 57.8% to 44.0% but still has 1/3 of poor CPU utilization due to most of the variables in the loops from the original program has dependencies between each other and kept as it was before.

The performance profile of the application is recorded using the Visual Studio 2019 integrated Intel VTune Performance Compiler. The final version of the parallelized program will be compiled using GNU g++ compiler with OpenMP support and run on QUT HPC platform using Secure Shell (SSH). We can type "lscpu" command in Linux terminal that gives the number of CPU threads per core and cores per socket available. We can define the number of threads for the jobs that we can run in parallel as stated in Section 4.1.3 to perform the testing.

## 5.0 Analysis of the Results

```
// Replaced time to chrono clock for more accurate timing
auto start = std::chrono::high_resolution_clock::now();

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

auto finish = std::chrono::high_resolution_clock::now();
std::chrono::duration<double> elapsed = finish - start;
printf("time elapsed: %f seconds\n", elapsed.count());
```

Figure 16 Replaced time with chrono time clock

For consistency, chrono library is used for high-resolution clock which provides more accurate timing for the tests. The original program takes approximately 50.5 seconds to finish while the optimized version (the best sequential application) takes approximately 21.7 seconds to complete which is more than double of that time taken for the original program to finish. The best sequential program can be built by removing the -fopenmp flag which enables supports for OpenMP during the compilation process of the application.

The test is performed in the QUT HPC platform where I have no full control of the background processes, hence the tests are repeated for 10 times to obtain an average value for the final analysis of the results. The complete test table is included in Appendix 9.6. We can also set the number of threads to be used dynamically in the number of physical cores when running the CV Tree program by declaring the OMP\_NUM\_THREADS environment variable. The complete guide is written in README.md file in the submission folder which also includes full tutorial on how to compile and run the program on Linux. The original, best sequential, and parallelized version of the program output the same correlation values to ensure that all of them are working correctly.

Best Sequential Time = 21.729439		
Parallel:		
Core	Elapsed Time	Speed Up
1	24.0078459	0.905097404
2	12.2918326	1.767794901
4	6.5151291	3.335227693
6	4.4636226	4.868117434
8	3.7349587	5.817852551
12	3.0939619	7.023176013
16	3.0678271	7.083006405
20	3.1418296	6.916173621

Figure 17 Table of results of elapsed time and speedup



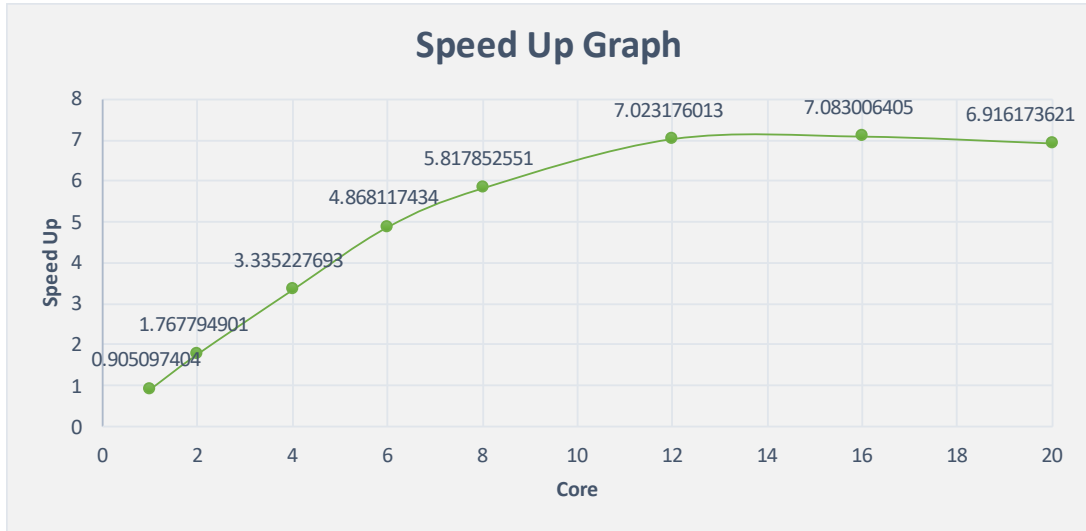


Figure 18 Speed up graph

The speedup of the parallelized version of the program is calculated by using the formula  $Speedup = \frac{Best\ Sequential\ Time}{Parallel\ Time}$ . The speedup graph shows a sub-linear pattern which is expected. The parallelized version using 1 thread is slower than the best sequential version (0.9 speedup) due to excessive overhead in thread creation. The speedup of the parallelized program reaches plateau at 12 threads. More than 12 threads in use will not give more speedup but slowing it down instead or the speedup obtain from it is not worth it than using less threads. This is because too many threads will drain the CPU time by giving the kernel more work to do and causes performance retrograde.

## 6.0 Software and Tools

<b>Visual Studio 2019</b>	Used for main development for this major assignment as well as debugging and view the memory and CPU usage of the original CV Tree improved.cpp program. The original and final version of the program is built in Release mode, 64-bit platform.
<b>High resolution clock</b>	A high-resolution clock from the chrono library is used for timing of all the tests. This clock has the shortest tick period and more timing accuracy compared to others.
<b>Intel VTune Profiler</b>	Used for profiling the CPU performance of the program.
<b>OpenMP Library</b>	Used for implementing parallelism in the program with pragma omp parallel for directive.
<b>QUT HPC</b>	Used for final testing of the parallelized version of CV Tree for a consistent result. QUT HPC has 24 processing units available that can be used for testing. Up to 20 threads are used for final testing.
<b>SSH</b>	Used for remotely connecting to account associated to QUT HPC platform.
<b>Cisco Connect</b>	Used for remotely connecting to QUT Wi-Fi for authorized access to the QUT HPC platform.
<b>GDB</b>	Used for debugging compilation error.
<b>GNU g++</b>	Used for compiling the CV Tree program.

*Table 2 List of software and tools*

## 7.0 Limitations and Difficulties

### 7.1 Fine grain parallelism

The original CV Tree program can also be parallelized using fine grain parallelism technique, breaking down the application to a large number of small tasks and then balancing the loads by assigning the tasks individually to many processors.

I tried to parallelize every loop which is safe from data dependency in the program, but it slows down the application instead of speeding it up. Since many loops only consume few CPU time so the additional overheads and computation power from thread creations are more expensive than the original computation of the loops themselves.

### 7.2 schedule clause

The other possible method to utilize “#pragma omp parallel for” directive from OpenMP library is by using the other type of scheduling clause. There are static, dynamic, and guided scheduling clause that I have used when comparing the parallelism performance between them.

In Section 4.1.2, we have discussed over the reason of choosing dynamic scheduler clause. A static scheduler means that each thread gets approximately the same number of iterations as any other thread, each iteration requires the same amount of work, and all threads should finish at about the same time. In this case we have unpredictable amount of work and the time taken for each thread execution does not guarantee to be the same, so it is not suitable.

On the other hand, a guided scheduler is more suitable for the case in which the threads may arrive at varying times, with each iteration requiring about the same amount of work. It has higher overhead in general due to extra computation from decreasing the chunk size. If the number of the input sequence file or the file size is a lot larger, using guided scheduler should perform better. For this application, using dynamic scheduler is better and safe overall because the workload per iteration is unpredictable.

### 7.3 Compilation error

GNU g++ compiler fails to identify and compile declaration for `errno_t`, `fscanf_s` and `sprintf_s` which can only be supported in Visual Studio 2019 and causes compilation error in QUT HPC platform. Thus, I modified the program which can be supported by GNU compiler just to perform testing on HPC. The versions of the program compiled by g++ and Visual Studio 2019 are both included in the submission folder so there is a total of three cpp files in the folder which includes `improved-original.cpp` (original version), `improved.cpp` (parallelized, Visual Studio 2019 version) and `improved-g++.cpp` (parallelized, g++ version) of the program.

```

[n10116877@sef-bigdata-01 example]$ g++ -o improved improved.cpp -g
improved.cpp: In constructor 'Bacteria::Bacteria(char*)':
improved.cpp:86:17: error: 'errno_t' was not declared in this scope; did you mean 'ino_t'?
   86 |         errno_t OK = fopen_s(&bacteria_file, filename, "r");
      |         ^~~~~~
      |         ino_t
improved.cpp:88:21: error: 'OK' was not declared in this scope
   88 |         if (OK != 0)
      |         ^~
improved.cpp: In function 'void ReadInputFile(const char*)':
improved.cpp:176:9: error: 'errno_t' was not declared in this scope; did you mean 'ino_t'?
   176 |         errno_t OK = fopen_s(&input_file, input_name, "r");
      |         ^~~~~~
      |         ino_t
improved.cpp:178:13: error: 'OK' was not declared in this scope
   178 |         if (OK != 0)
      |         ^~
improved.cpp:184:9: error: 'fscanf_s' was not declared in this scope; did you mean 'fscanf'?
   184 |         fscanf_s(input_file, "%d", &number_bacteria);
      |         ^~~~~~
      |         fscanf
improved.cpp:192:17: error: 'sprintf_s' was not declared in this scope; did you mean 'sprintf'?
   192 |         sprintf_s(bacteria_name[i], 20, "data/%s.faa", name);
      |         ^~~~~~
      |         sprintf

```

Figure 19 Error when compiling the program with g++

There is also a segmentation fault (core dumped) error during the first run of the g++ compiled program. I had to use GDB debugger with -g flag to debug the issue. It was due to the end of line issue with the input fasta files as some of the file has carriage return and the others are new line character. Even though the Visual Studio 2019 automatically ignore this issue, the G++ compiler does not. So, I had to edit the code in Bacteria constructor function so that it checks for the carriage return (see Figure 20).

```

// Modified for G++
FILE* bacteria_file = fopen(filename, "r");

InitVectors();

char ch;
while ((ch = fgetc(bacteria_file)) != EOF)
{
    if (ch == '>')
    {
        while (fgetc(bacteria_file) != '\n'); // skip rest of line

        char buffer[LEN - 1];
        fread(buffer, sizeof(char), LEN - 1, bacteria_file);
        init_buffer(buffer);
    }
    else if (ch != '\n' && ch != '\r')
        cont_buffer(ch);
}

```

Figure 20 Modified else if statement to check for carriage return

## 8.0 Conclusion

The final version of the parallelized CV Tree achieved 74.8% of parallelism and takes only 3 seconds to complete execution using appropriate optimizations and OpenMP multithreading library from the original application which takes 50.5 seconds, giving a speedup ratio of 7. Using more than 12 threads will cause excessive overhead in thread creation, slowing down the application instead of speeding it up. Overall, I am satisfied with the performance of the parallelized application with the CPU utilization graph included in Appendix 9.4 achieved even and wide spread of task distribution.

Deep understanding of the sequential application and analysis of data dependencies are necessary process to ensure that we can modify the code and algorithm in the program for parallelism safely. It is also important to investigate whether the CPU usage of a loop that is safe to be parallelized worth for the parallelism.

Even though there are some difficulties during the development of this project such as compilation error, segmentation error and missing necessary libraries. There are all able to be solved in the end.

During the early stages of development, I investigated the program in local machine with Windows environment which I have limited access to GPU parallelism, and I am unfamiliar with GPU parallelism techniques. Thus, I did not opt to implement GPU parallelism in the program, and it may not provide much more speedup than the final parallelized version with OpenMP of the program already has. I will attempt GPU parallelization to prove that it does not worth for the implementation in exchange for limited speedup if I have more time.

## 9.0 Appendix

### 9.1 CPU Time (Bacteria constructor function)

Source	Assembly	=	🔍 ⬆ ⬇ ⬇ ⬇
Source Line ▲	Source	🔥 CPU Time: Total ⌵	CPU Time: Self ⌵
125	for(long i=0; i<M; i++)	1.3%	1.797s
126	{		
127	double p1 = second_div_total[i_div_aa_number];	1.5%	2.046s
128	double p2 = one_l_div_total[i_mod_aa_number];	1.1%	1.551s
129	double p3 = second_div_total[i_mod_M1];	1.6%	2.126s
130	double p4 = one_l_div_total[i_div_M1];	0.3%	0.438s
131	double stochastic = (p1 * p2 + p3 * p4) * total_div_2;	8.2%	11.077s
132			
133	if (i_mod_aa_number == AA_NUMBER-1)	1.0%	1.393s
134	{		
135	i_mod_aa_number = 0;		
136	i_div_aa_number++;	0.1%	0.095s
137	}	0.0%	0.031s
138	else		
139	i_mod_aa_number++;	1.0%	1.314s
140			

Source	Assembly	=	🔍 ⬆ ⬇ ⬇ ⬇
Source Line ▲	Source	🔥 CPU Time: Total ⌵	CPU Time: Self ⌵
141	if (i_mod_M1 == M1-1)	0.2%	0.266s
142	{		
143	i_mod_M1 = 0;		
144	i_div_M1++;		
145	}		
146	else		
147	i_mod_M1++;	1.3%	1.797s
148			
149	if (stochastic > EPSILON)	4.6%	6.285s
150	{		
151	t[i] = (vector[i] - stochastic) / stochastic;	2.0%	2.723s
152	count++;	0.1%	0.110s
153	}		
154	else		
155	t[i] = 0;	1.3%	1.749s
156	}	1.4%	1.907s

## 9.2 CPU Time (CompareBacteria() function)

VT

Hotspots

Hotspots by CPU Utilization

Analysis Configuration

Collection Log

Summary

Bottom-up

Caller/Callee

Top-down Tree

Platform

improved.cpp

Source

Assembly

Source Line

Source

🔥 CPU Time: Total

CPU Time: Self

212

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

2.5%

1.238s

213

{

214

long n1 = b1->ti[p1];

0.8%

0.419s

215

long n2 = b2->ti[p2];

1.2%

0.584s

216

if (n1 < n2)

0.5%

0.263s

217

{

218

double t1 = b1->tv[p1];

3.1%

1.522s

219

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

1.2%

0.616s

220

p1++;

221

}

2.0%

0.999s

222

else if (n2 < n1)

2.3%

1.130s

223

{

224

double t2 = b2->tv[p2];

1.0%

0.513s

225

p2++;

226

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

2.4%

1.188s

227

}

1.9%

0.929s

228

else

229

{

230

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

0.9%

0.461s

231

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

0.2%

0.102s

232

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

0.2%

0.105s

233

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

0.0%

0.015s

234

correlation += t1 \* t2;

0.3%

0.149s

235

}

236

}

## 9.3 CPU Utilization (Original CV Tree)

Hotspots Hotspots by CPU Utilization

Analysis Configuration Collection Log Summary Bottom-up Caller/Callee Top-down Tree Platform

Elapsed Time: 51.077s

CPU Time: 49.738s

Total Thread Count: 1

Paused Time: 0s

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time
Bacteria::Bacteria	improved.exe	28.769s
CompareBacteria	improved.exe	10.232s
free_base	ucrtbase.dll	6.119s
func@0x180001980	VCRUNTIME140.dll	3.821s
_stdio_common_vfprintf	ucrtbase.dll	0.558s
[Others]	N/A*	0.239s

\*N/A is applied to non-summable metrics.

Hotspots Insights

If you see significant hotspots in the Top Hotspots list, switch to the Bottom-up view for in-depth analysis per function. Otherwise, use the Caller/Callee view to track critical paths for these hotspots.

Explore Additional Insights

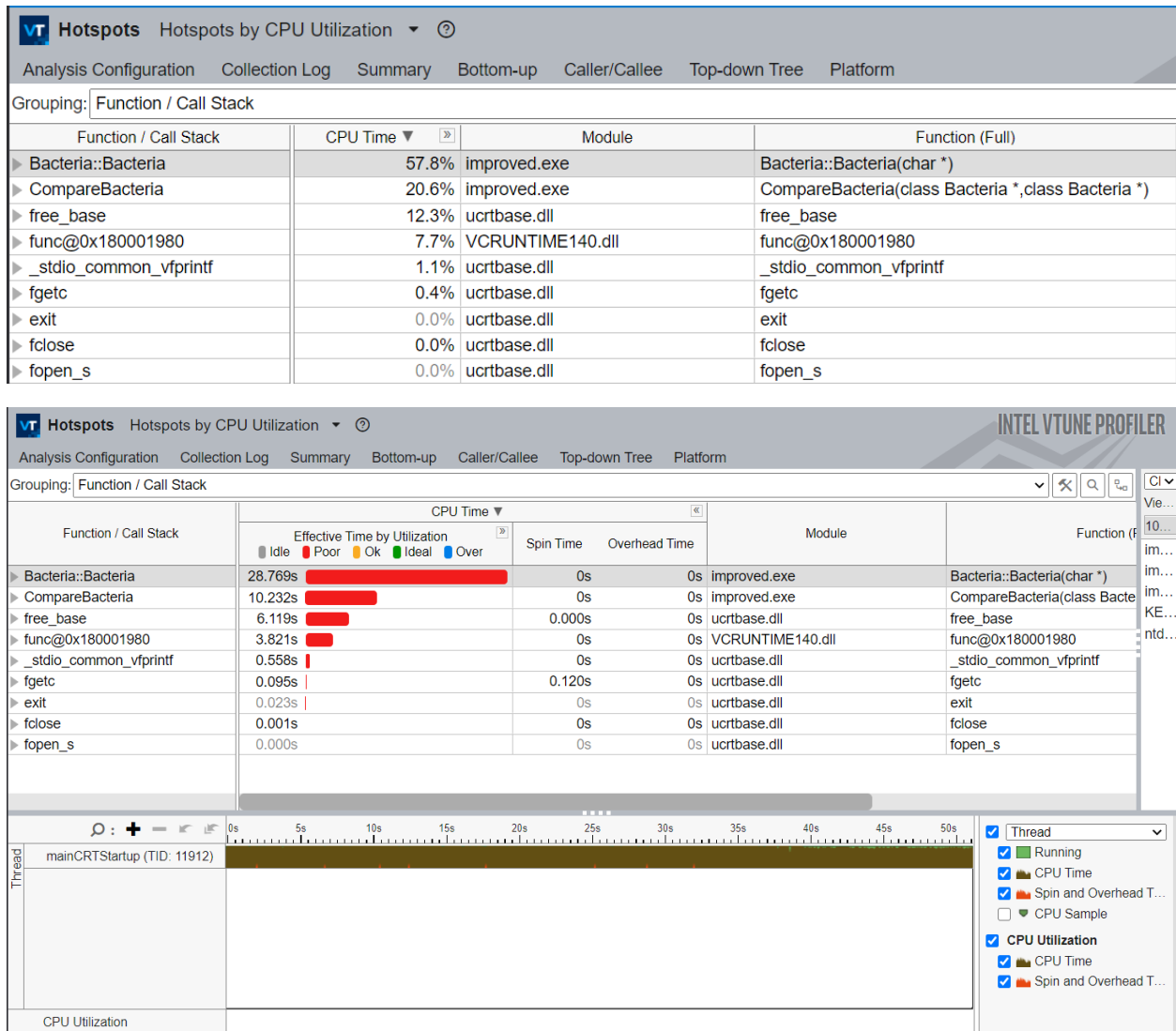
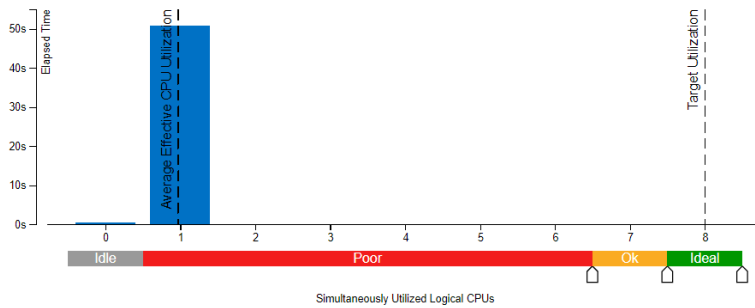
Parallelism: 12.1%

Use Threading to explore more opportunities to increase parallelism in your application.

INSIGHTS

## Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.



## 9.4 CPU Utilization (Parallelized CV Tree)



vt

Hotspots

Hotspots by CPU Utilization

Analysis Configuration

Collection Log

Summary

Bottom-up

Caller/Callee

Top-down Tree

Platform

Elapsed Time: 8.915s

CPU Time: 54.249s

Total Thread Count: 8

Paused Time: 0s

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time
Bacteria::Bacteria	improved.exe	23.859s
CompareBacteria	improved.exe	17.924s
func@0x180001980	VCRUNTIME140.dll	7.577s
free_base	ucrtbase.dll	2.181s
std::vector<double,class std::allocator<double> >::_Emplace_reallocate<double>	improved.exe	0.985s
[Others]	N/A*	1.722s

\*N/A is applied to non-summable metrics.

Hotspots Insights

If you see significant hotspots in the Top Hotspots list, switch to the Bottom-up view for in-depth analysis per function. Otherwise, use the Caller/Callee view to track critical paths for these hotspots.

Explore Additional Insights

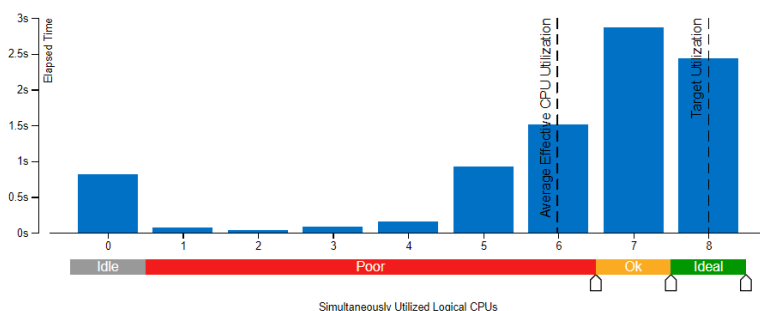
Parallelism: 74.8%

Use Threading to explore more opportunities to increase parallelism in your application.

INSIGHTS

Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.



vt

Hotspots

Hotspots by CPU Utilization

Analysis Configuration

Collection Log

Summary

Bottom-up

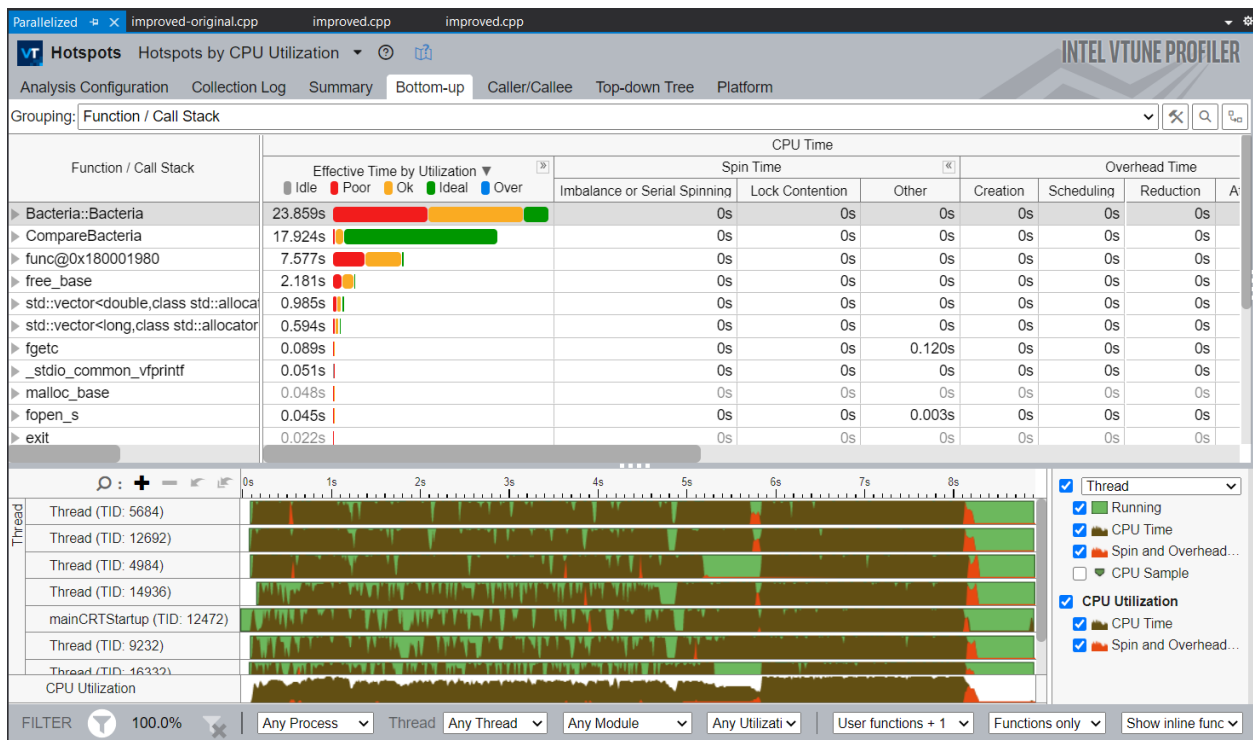
Caller/Callee

Top-down Tree

Platform

Grouping: Function / Call Stack

Function / Call Stack	CPU Time	Module	Function (Full)
Bacteria::Bacteria	44.0%	improved.exe	Bacteria::Bacteria(char *)
CompareBacteria	33.0%	improved.exe	CompareBacteria(class Bacteria *,class Bacteria *)
func@0x180001980	14.0%	VCRUNTIME140.dll	func@0x180001980
free_base	4.0%	ucrtbase.dll	free_base
std::vector<double,class std::allocator<double> >::_Emplace_reallocate<double>(	1.8%	improved.exe	std::vector<double,class std::allocator<double> >::_Emplace_reallocate<double>(
std::vector<long,class std::allocator<long> >::_Emplace_reallocate<long const &>(	1.1%	improved.exe	std::vector<long,class std::allocator<long> >::_Emplace_reallocate<long const &>(
fgetc	0.4%	ucrtbase.dll	fgetc
_stdio_common_vfprintf	0.1%	ucrtbase.dll	_stdio_common_vfprintf
malloc_base	0.1%	ucrtbase.dll	malloc_base
fopen_s	0.1%	ucrtbase.dll	fopen_s
exit	0.0%	ucrtbase.dll	exit
	0.0%	VCRUNTIME140.dll	memset
	0.0%	ucrtbase.dll	fclose



## 9.5 Full test results on QUT HPC

	Core	1	2	4	6	8	12	16	20
Test Num									
1		23.89095	12.3049	6.499332	4.446641	3.764435	3.076717	2.970818	3.274244
2		23.95791	12.39325	6.474997	4.46866	3.765433	3.260613	3.169832	3.087116
3		23.8768	12.29525	6.546957	4.478192	3.73121	3.096043	2.976293	3.123963
4		24.10852	12.2173	6.556766	4.475639	3.745191	3.090735	3.116533	3.09056
5		24.18048	12.32883	6.536932	4.479428	3.714036	3.083632	2.993883	3.179891
6		24.05462	12.38061	6.459095	4.479222	3.737787	3.053646	3.097139	3.071302
7		24.02878	12.36375	6.454204	4.460134	3.746032	3.057563	3.04406	3.167074
8		24.00619	12.19182	6.541837	4.433841	3.699128	3.067852	3.169436	3.055269
9		23.88588	12.18424	6.559285	4.453277	3.705434	3.08428	3.139366	3.305488
10		24.08833	12.25838	6.521886	4.461192	3.740901	3.068538	3.000911	3.063389
Average		24.00785	12.29183	6.515129	4.463623	3.734959	3.093962	3.067827	3.14183