CV Tree parallelisation

CAB401

Sam Hutton n10206655

Contents

[What Does CV Tree Do? 2](#_Toc86645666)

[Analysis of Potential Parallelism 4](#_Toc86645667)

[Computation Mapping and Synchronisation 5](#_Toc86645668)

[Timing and Profiling Results 6](#_Toc86645669)

[How Were Program Outputs Tested 8](#_Toc86645670)

[Software and Tools Utilised 9](#_Toc86645671)

[Problems / Barriers 9](#_Toc86645672)

[Code Modifications 10](#_Toc86645673)

[Reflection 12](#_Toc86645674)

# What Does CV Tree Do?

Diagram

Description automatically generatedCV Tree is a program that takes a text file that describes the quantity and names of a group of bacteria contained in Fasta files located inside another folder in the project. The program enters each of these Fasta files one by one, systematically iterating through each of the genes that make up the bacteria and the amino acids that make up these genes. It then creates a bacteria object in the program and compares each bacteria object to the others and outputs the correlation or the value for how similar each bacteria is to all the other bacteria.

**Figure 1. High level overview of CV Tree application**

This is a very simplified overview of the program and the operations it performs. Of these tasks, almost the entirety of the program is spent within the create bacteria objects and compare all bacteria functions. These correlate to the Bacteria constructor, and the CompareBacteria function. This is detailed more heavily in figure 2.

A screenshot of a computer

Description automatically generated with medium confidence

**Figure 2. Caller / Callee function diagram**

Figure 2 depicts the two key functions within the program, and the structure of this follows a logical order: Main calls CompareAllBacteria, CompareAllBacteria calls Bacteria::Bacteria (Bacteria constructor) and then calls CompareBacteria. As depicted above, this is where the program spends over 99% of its time

The main purpose of the Bacteria Constructor is to calculate a tv, ti, and count value variable for each Bacteria object and the main purpose of the CompareBacteria method is to calculate a correlation value between the two bacteria.

Diagram

Description automatically generatedDiagram

Description automatically generated

**Figure 3. Bacteria Constructor Overview Figure 4. CompareBacteria Method Overview**

Figure 3 and Figure 4 above give us a slightly more detailed look into the operations that happen in the two significant methods. Inside the Bacteria Constructor there is the opening of a file followed by a number of for loops to calculate different values and initialise arrays. Whilst in CompareBacteria, it takes two Bacteria objects and perform a number of calculations in order to return a correlation at the end.

# Analysis of Potential Parallelism

A screenshot of a computer

Description automatically generated with medium confidence

**Figure 5. Sequential CPU Profiler**

Firstly, looking at Figure 5, we can see very little computing power is actually used by the CPU during the runtime of the program yet it still takes approximately 30 seconds to complete. This is the first indication that the program may benefit from parallelism.

Diving deeper into the program, we can see the two most significant functions; the Bacteria constructor, and CompareBacteria – are both called within the CompareAllBacteria function inside a for-loop. As explained above, the Bacteria constructor and CompareBacteria are the two most significant functions in the program, so that is a good place to start.

void CompareAllBacteria()

{

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

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]);

}

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 6. CompareAllBacteria Function – Potential For Parallelism**

Looking inside the Bacteria constructor in Figure 4, there are 4 for-loops and a while loop. It would be possible to run all of these for loops in parallel, but this is extremely fine grain parallelism and is not likely to not result in significant performance increase. If we however look at the loop inside of CompareAllBacteria in Figure 6, we can see that repeated calls to the constructor are simply used to fill an array (b) with Bacteria objects. There are no dependencies in this loop as each iteration is only writing to the i value of b. This provides us with much more coarse grain parallelism as we are able to call the constructor multiple times at once allowing us to create many Bacteria objects at the same time. Although it would not be parallelism, there are improvements that can be made inside the constructor in regards to several of the arrays, particularly how they are created, destroyed, and the size of them

Onto CompareBacteria, there are 3 while loops contained within the function, none of which can be parallelised. However, similarly to the previous example CompareBacteria is called in a loop which indicates there may be exploitable parallelism. There is a flow dependency that must be controlled to maintain program correctness. If the loop was simply just run in parallel, the correlation and the bacteria currently being compared would not be outputted correctly to the console. In order to overcome this, we can create an array to store the correlations that are calculated in the parallel loop and then print those values in a sequential loop following it.

Finally, it is important to remember that there is an inherent dependency that all Bacteria objects must be loaded into the program before the correlation calculations can begin.

These proposed changes are scalable parallelism, and the programs run time will continue to scale as the hardware its run on improves. Ultimately, this program is definitely parallelisable but may be limited by some IO constraints. The main culprit of this is the Bacteria constructor which needs to access 41 different Fasta files and read through in some cases thousands of lines of amino acids. The actual speed of this process is limited, but it is still possible for the process to run multiple times in parallel. Additionally, print is called 820 times each time the program runs, and this cannot be made to run in parallel.

# Computation Mapping and Synchronisation

I made used of the OpenMP library to expose the parallelism available within the aforementioned for-loops. This made the process of mapping computations to different processors extremely simple and efficient.

#pragma omp parallel for

**Figure 7. Computational Mapping with OpenMP**

Figure 7 details how simple it can be in some cases to map computational data to threads using the OpenMP library. It isn’t always as simple as this however, and there are many keywords that can be placed after the “for” that result in different operations taking place.

OpenMP has implicit synchronisation built in that means at the end of any parallel process, the threads created to run and compute that region of code are dissolved and only the master thread will continue. This is called an implicit barrier and happens at the end of each of the parallelised loops. However, some of the aforementioned keywords that can change this behaviour include the critical, atomic, barrier, or nowait keywords that all change how synchronisation is performed.

# Timing and Profiling Results

A screenshot of a computer

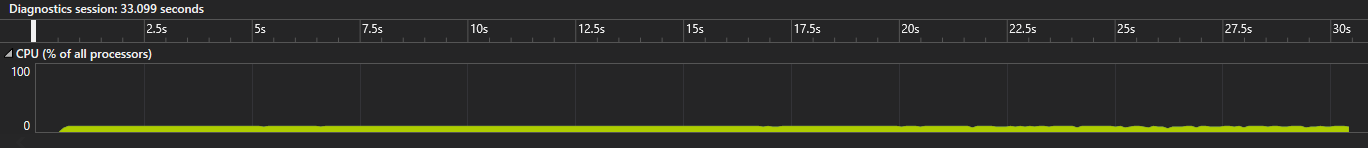
Description automatically generated with low confidence

**Figure 8. Result Chart of 10 Runs Between 1 and 12 Cores**

This table details 10 runs of the sequential program followed by 10 runs of the parallel program using between 1 and 12 cores. Of note should be that running on 8 cores produced a worse result than on 7. It is quite likely that this was just by chance, and would not happen if the test were repeated multiple times.

**Figure 9. Speedup Curve**

These results describe that the speedup was sublinear. While a speedup was clearly achieved, I don’t believe it was the most optimal speedup for this program.

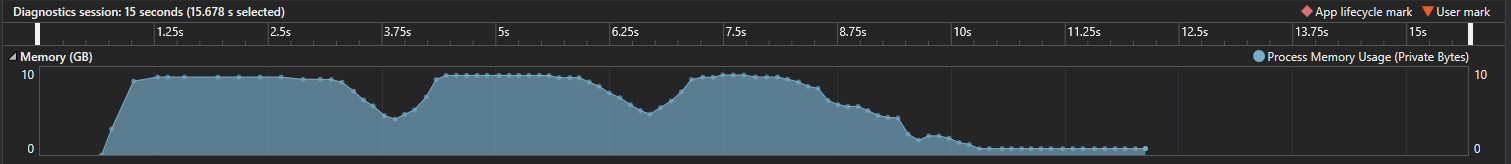
A screenshot of a computer

Description automatically generated with low confidence

**Figure 10. Sequential Profiler**

A screenshot of a computer

Description automatically generated with medium confidence



**Figure 11. Parallel Profiler**

The sequential memory profiler makes it very clear what is happening throughout the runtime of the program. It clearly depicts accessing the 41 files sequentially and creating a bacteria object each time follow by approximately 8 seconds of bacteria comparison and outputting to console.

The parallel profiler however completes many of these tasks in parallel so there is much higher memory usage (2GB -> 10GB) and CPU usage (10% -> 100%). There are significantly more computing resources being utilised by the program, engaging all 12 virtual cores accessible on the CPU, and using up to 10GB of memory when rapidly accessing the Fasta files in parallel.

Interestingly, there are significant drop offs in the amount of memory being utilised during the project at the 3 second and 6 second marks. I believe this to be when the 12 threads have completed their first Fasta file and are about to enter into the second.

# How Were Program Outputs Tested

Initially, I was copying and pasting results to DiffChecker to see whether the results were the same, but this was very tedious and rather inefficient.

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

{

if (myFile.is\_open())

{

myFile << i << " " << j << " -> " << std::setprecision(20) << storage[j] << "\n";

}

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

}

**Figure 12. Writing Program Output to “myFile”**

Instead, I modified the program slightly so that it would write to a file at the same time as it would be writing to console, as shown in the code above. Firstly, I ran this code for the sequential version to generate a correct text file and then every time I ran the parallel version it would create a new text file with the values from that run inside it and I could compare these.

while (!Sequential.eof())

{

Sequential.getline(string1, 256);

Parallel.getline(string2, 256);

j++;

if (strcmp(string1, string2) != 0)

{

areSame = false;

}

}

if (areSame)

{

printf("Files are Identical");

}

else

{

printf("Invalid Solution");

}

**Figure 13. Code Within Main to Check Sequential Against Most Recent Output**

I put this code into the Main, and after every run of the program it would print whether the two text files were identical. This allowed me to instantly check whether or not the changes I had made to the program produced incorrect results or not.

For the final submission however, I did remove these lines of code as they slightly slowed the program down.

# Software and Tools Utilised

For writing and running my code, I made use of the IDE and compiler available through Visual Studio 2019. This was extremely useful and very easy to write, run and test my code all in the same place. A tool inside Visual Studio that was extremely helpful in finding out what the program was doing is the performance profiler and the outputs it produces. This was an extremely useful starting point and something I referred to constantly as I continued to make changes to the program. The hot path and the caller tree showed exactly where the program was spending the most amount of time which was obviously where I needed to look to exploit potential parallelism.

In terms of software libraries, I made use of the OpenMP library as throughout this unit, I found this to be the easiest way to parallelise for-loops and that was where most of the parallelism in this program could be exploited. Additionally, I utilised the chrono library to time and record my code, so I knew how long the program took to run.

Regarding actual parallelisation techniques, I had to make use of loop splitting. This will be detailed further on, but it was utilised to ensure program correctness whilst gaining a speedup within a for-loop.

# Problems / Barriers

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 14. Loop Splitting to Ensure Program Correctness**

The largest problem I had was ensuring program correctness when the parallel changes were implemented. As can be seen in figure X, whilst this loop was safe to parallelise, and the values would be calculated correctly but would be outputted incorrectly. In order to overcome this, I had to use the loop splitting technique, and make use of an array to store the correlation values. This resulted in a program that always output the correlation values correctly.

Secondly, I noticed that memory was used very inefficiently through the Bacteria class constructor and spent a significant amount of time testing and trying methods to improve this. The closest I came to success was changing the ti, tv, and t values to a vector. I feel that my implementation was very close to working but I was unable to get a run that produced correct values. In my testing I found that this significantly sped up the program as it ignored many of the 0 values produced through the current arrays however I can’t confirm whether this speed up was real or just because mistakes I made reduced the total number of calculations.

Additionally, I spent time trying to optimise the loops within this constructor and even trying to parallelise them. However, after safely parallelising many of the loops, and running time tests I found there to be no improvement. I spent time researching this and found that constructors can only be accessed by one thread so any parallelised loops inside this were simply ignored, and no improvement found.

# Code Modifications

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 15. Sequential Bacteria Creation Loop**

#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]);

}

**Figure 16. Parallelised Bacteria Creation Loop**

There is no dependency in the order of which bacteria is loaded into the program so this loop was completely safe to parallelise and was done so utilising OpenMP.

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 17. Sequential Correlation Calculation**

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

{

#pragma omp parallel for

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

{

storage[x] = CompareBacteria(b[i], b[x]);

}

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

{

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

}

}

**Figure 18. Parallelised Correlation Calculation**

There was a flow dependency in this loop, which if ignored resulting in incorrect values and orders of correlations being printed to the screen. In order to combat this, I utilised a loop splitting technique and moved the printing to another loop and left the calculations in the parallelised loop.

if (stochastic > EPSILON)

{

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

count++;

}

else

t[i] = 0;

**Figure 19. Original If Statement**

if (stochastic > EPSILON)

{

t[i] = (vector[i] – stochastic) / stochastic;

count++;

}

**Figure 20. Updated If Statement**

I also made this slight code modification within the Bacteria constructor which stopped assigning a value of 0 to parts of the t array. This is because in the next part of the code, it ignores all 0 values within the t array, so this modification was safe to do.

Final Source Code Changes

|  |  |
| --- | --- |
| Lines Added | 6 |
| Lines Removed | 2 |
| Total Line Count | 302 -> 306 |

# Reflection

Throughout this process, I learnt a lot about C++, performance profiling, and determining whether it’s safe to parallelise certain functions or loops.

C++ is an extremely interesting language and through the amount of time I spent in it for this assignment I learnt a lot about memory management, data structures and the available libraries associated with the program. The performance profiler was a tool available to me through the Visual Studio 2019 program which was extremely useful in understanding the program and visualising what the CPU and memory were doing during run time. Finally, the experience of learning whether a loop has dependencies and whether or not it can be parallelised was extremely interesting and seeing the results of what happens when this is done incorrectly or cannot be done has taught me a lot.

The attempt was to the best of my ability, but I can’t help to feel that I may have missed something. I exposed a fair amount of coarse grain parallelism and attempted many different ways to speedup the program. There is a fair amount of bottlenecking firstly when reading the files, which after much research and effort I don’t believe there is any way to speed up this process. Secondly, I feel that there is a possibility that the Bacteria constructor is slowing down the program, but I was unable to find anything that could improve the performance further.

Looking solely at the most computationally heavy functions, the Bacteria constructor and CompareBacteria, without significant changes to the code I don’t believe much more parallelism can be exploited, but I believe that there are many improvements that could be made. Constructors are called by a single thread, so unless there is an alternative way to run this program without utilising the Bacteria class, I don’t think more parallelism can be exploited. Obviously as mentioned previously, I spent a considerable amount of time trying to improve the memory usage of the program to potentially exploit more parallelism but was unable to do so within this constructor. Secondly, the CompareBacteria makes use of many while loops that cannot be converted to for-loops so once again, I don’t believe there is any way to further parallelise this code unless there is a more optimal way to calculate correlations than the current function.

Ultimately, whilst parallelism could not be exploited, I am certain there are inefficiencies that exist in regard to a number of the arrays and the way they’re handled by memory. I spent a significant amount of time attempting to improve this, but I could not manage to preserve program correctness whilst doing so. A considerable speedup was achieved whilst testing but the results were completely inaccurate so I cannot verify whether the changes I attempted caused a speedup or just made a mistake.

Ultimately, the speedup achieved was much lower than a linear speedup or even the most optimal speedup possible for this program. There are certainly some optimisations that could be made around the Bacteria constructor in terms of memory efficiency and data creation that could possibly expose more parallelism, but I was unable to exploit this.