CAB401

High Performance and Parallel Computing

Fredrik Forsell - N10297057

Semester 2, 2018

Explanation of sequential code:

Program Details:

Name: CV-TREE / Bioinformatics - Genome similarity using Frequency Vectors (C++)

Language: C++

Development Framework: Microsoft Visual Studio Community 2017

Original Source Code: Example Applications to Parallelize:

Bioinformatics - Genome similarity using Frequency Vectors (C++) / cvtree

Brief Description: CV-Tree is an application made for Mac OS X, Linux and

Windows (C++) and works by comparing 6-mers. A k-mer is all the substrings in a text that contains 6 characters. In this case, these 6-mers are small subsequences from a gene. This data will be processed to decide which bacterias that are most closely related.

Development Machine Specifications:

Processor: Intel Core i7-6700

Number of Cores: 4 cores

Threads: 8

Socket: FCLGA1151

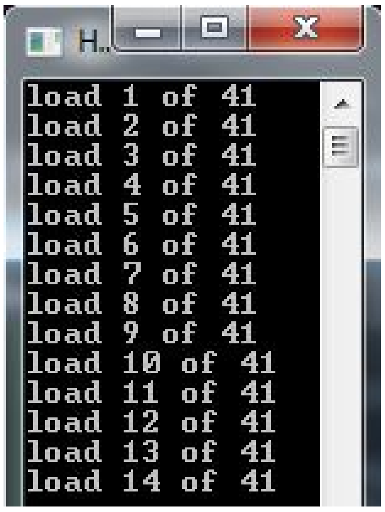
Cache: 8MB

Clock Speed: 3.40 GHz

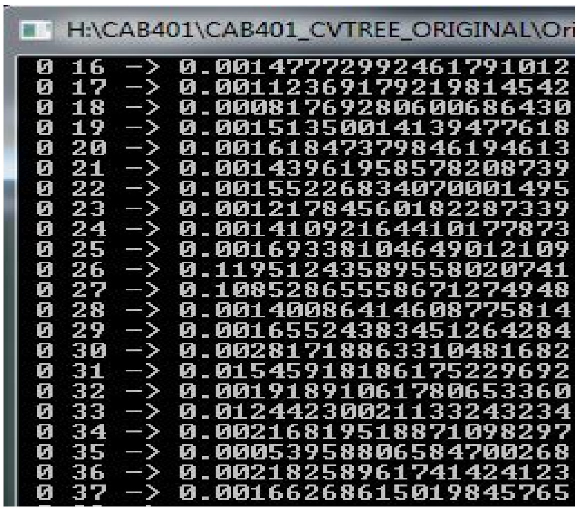
Hyper-Threading: Yes

Memory: 16GB

Screenshots:



```
load 1 of 41
load 2 of 41
load 3 of 41
load 4 of 41
load 5 of 41
load 6 of 41
load 7 of 41
load 8 of 41
load 9 of 41
load 10 of 41
load 11 of 41
load 12 of 41
load 13 of 41
load 14 of 41
```

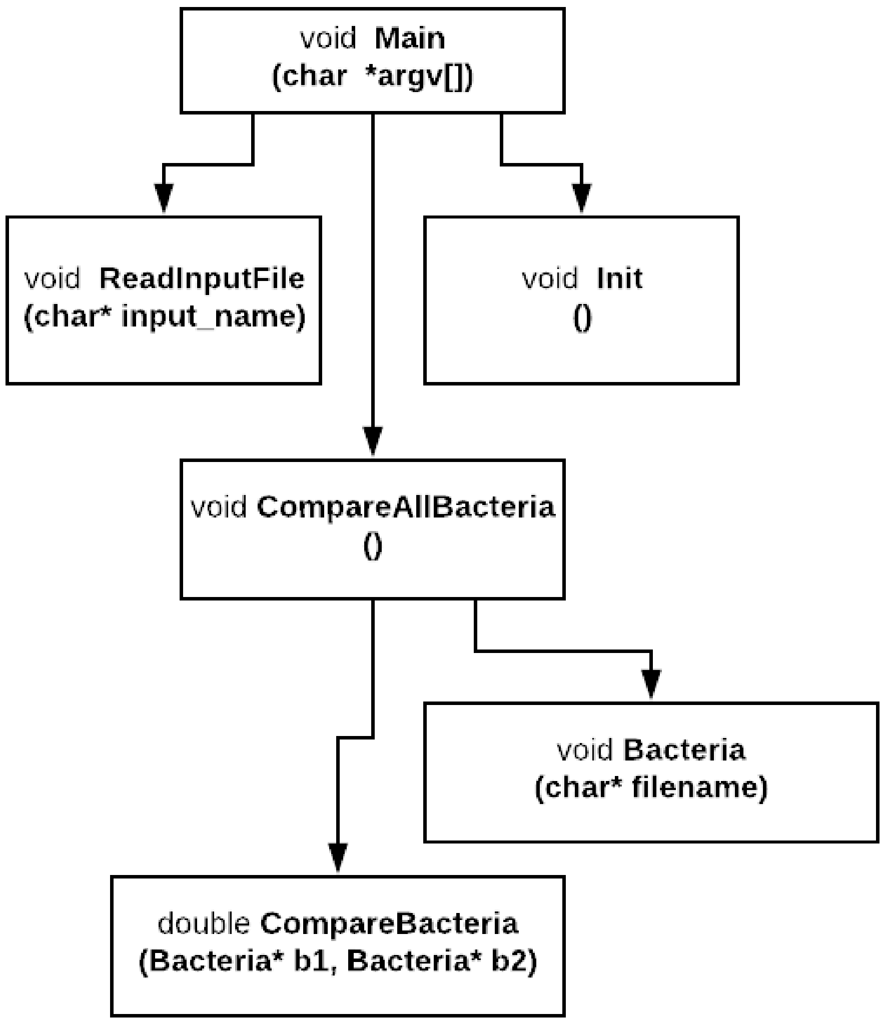


H:\CAB401\CAB401_CVTREE_ORIGINAL\Or

```
0 16 -> 0.0014777299246179101 2
0 17 -> 0.001123691792198145 42
0 18 -> 0.000817692806006864 30
0 19 -> 0.001513500141394776 18
0 20 -> 0.001618473798461946 13
0 21 -> 0.001439619585782087 39
0 22 -> 0.001552268340700014 95
0 23 -> 0.001217845601822873 39
0 24 -> 0.001410921644101778 73
0 25 -> 0.001693381046490121 09
0 26 -> 0.119512435895580207 41
0 27 -> 0.108528655586712749 48
0 28 -> 0.001400864146087758 14
0 29 -> 0.001655243834512642 84
0 30 -> 0.002817188633104816 82
0 31 -> 0.015459181861752296 92
0 32 -> 0.001918910617806533 60
0 33 -> 0.012442300211332432 34
0 34 -> 0.002168195188710982 97
0 35 -> 0.000539588065847002 68
0 36 -> 0.002182589617414241 23
0 37 -> 0.001662686150198457 65
```

Function Call Graph:

The function call graph below contains all of the functions that the CV-Tree application reads and processes while running the application. It contains the return type of the functions and also the input types that are necessary for the application to run through the functions. The vertical distances between the functions gives a simple representation of the run order they are processed in and their dependencies. This process is more thoroughly explained during the high level breakdown below the graph. I chose to not include the functions stored in the bacteria class in the diagram due to the small work they do; however, I will mention it while explaining the Bacteria() constructor in step 5.


void Main (char *argv[])
void ReadInputFile (char* input_name)
void Init ()
void CompareAllBacteria ()
void Bacteria (char* filename)
double CompareBacteria (Bacteria* b1, Bacteria* b2)

High level breakdown of the program:

This is a high level breakdown of the steps the program goes through while reading and processing the Gene data (every step covers 1 function of the application):

1. The Main() function is where we choose and direct all of the data that we want to process. It takes in one parameter (directory of a list file) and this parameter is used to send a text list file to the ReadInputFile() function. The Main() function calls 3 different functions in the following order:
    a. Init()
    b. ReadInputFile()
    c. CompareAllBacteria()

    The first 2 functions (a,b) are both made to prepare the necessary information for running the last function CompareAllBacteria (c).

    (A) and (b) are not dependant on each other to be able to run the application. Therefore I have placed them at the same height in the Call Diagram above.
2. The Init() function is used to set up variables for the k-mers as type long. M2 is the size of k-mer that the user wants to calculate, M1 is the K-mer -1 in length and M is the K-mer -2 in length.

    The reason this information is calculated in a function, and not stored as premade variables, is because if the user wants to search through the values with a longer k-mer, all the user has to do is change the globally defined LEN value.
3. The ReadInputFile() takes in one parameter char* that stores the link to a text document. This text document contains the name of 41 different bacterias. All of these bacterias have individual files with more data stored in them. These '.faa' files contain multiple genes that the bacteria is made up with.

    This function reads the .txt file and creates an array that stores all of the filenames and concatenates with the file ending in .faa. Now we have a complete list stored in the array bactaria_name[].

    Now that all of the necessary setup functions has been run, we can now move to the next step in the picture (CompareAllBacteria()).
4. The function CompareAllBacteria() is where most of the computation is initiated.

    It starts with running through a 'for' loop that runs through all of the bacteria_names[] that was set up in step 3. All of these bacterias is sent in to the function Bacteria(), where it generates a new object for every bacteria (explained further in step 5). All of these objects are stored in the array b[].

After the Bacteria files are stored in objects then we can compare them using the compareBacteria(b[], b[]) function. The compareBacteria() function will be explained in step 6.

This comparison is done through a double ‘for’ loop which compares by sending in two parameters (bacteria object and the next bacteria object in the b[] object list). The return value of this function is the correlation value that is printed (as you can see in the second picture in program details -> Screenshots page 1).

5. The function Bacteria() is a constructor for the Bacteria class where the bacteria objects are created.

   The objects keep track of how many instances there are of K-mers. This works by opening the bacterias .faa file where all of the genes are stored.

   a. Every line in this file that starts with ‘>’ are skipped as this is only metadata about the gene.
   b. It then stores the 5 first characters of the first 6-mer in a new buffer that it’s creating (init_buffer()).
   c. The reason why it only uses the first 5 characters of the 6-mer is because it will reuse the 5 characters and append the next character.
   d. As long as the gene contains more characters it will run the cont_buffer() where it will add a character and increment a counter for the k-mers.
   e. The InitVector() Functions keeps track of the amount of 6-mers, 5-mers and 4-mers the bacterias have in base 20. The reason it is in base 20, is because 20 is the amount of different amino acid genes a bacteria has.
   f. The class also stores more specific information about the bacterias, such as the count of k-mers, total amount, etc.
6. The CompareBacteria() function is where Bacteria objects gets compared. If the pattern between the amount of 6-mers are similar, it will get a high score. It takes in two parameters which are the two bacteria objects that are being compared.

   The function returns a value that is calculated in 3 steps. These steps are run the same amount of times as they have k-mers (b1->count and b2->count):

   1. If bacteria 1’s k-mer is smaller than the second one (n1 < n2) the value of the number ‘p1‘ increments by one and also adds the value of (t1 * t1) to vector_len1.
   2. If bacteria 2’s k-mer is smaller than the first one (n2 < n1) the value of the number ‘p2‘ increments by one and also adds the value of (t2 * t2) to vector_len2.

3. If they are the same value (else statement -> n1 == n2), both of the vector_len value gets incremented as in step 1 and 2. t1 * t2 are also added to the correlation value.

Both of the vector_len will also increment with (t1*t1) or (t2*t2) depending on how many k-mers exist.

It will now return the correlation divided by (sqrt(vector_len1) * sqrt(vector_len2).

Description of compilers, software, tools and techniques used in the app:

All of the code was compiled using Visual Studio 2017. I also used the built-in compiler to profile the CPU and memory consumption of the application. I used the instrumental profiler along with the CPU profiler to detect the runtime and CPU processing power.

To ensure that the program worked after every change I made, I saved the original output of the program to a txt file. I then saved every new output into another file that was compared to the original output after runtime. This was done by adding the command system(String) where string was the CMD FC (file compare) command.

The timing of the runtime was done through the application itself. In the main method I added a timer that checked the current time before and after all of the functions had been processed. I then calculated the time in seconds and printed it out to the console.

I used the OpenMP language for all of the code that was parallelised. Alongside this, I used arrays to store values so that I could keep all the data in the sorting that I wanted.

Profiling the sequential program:

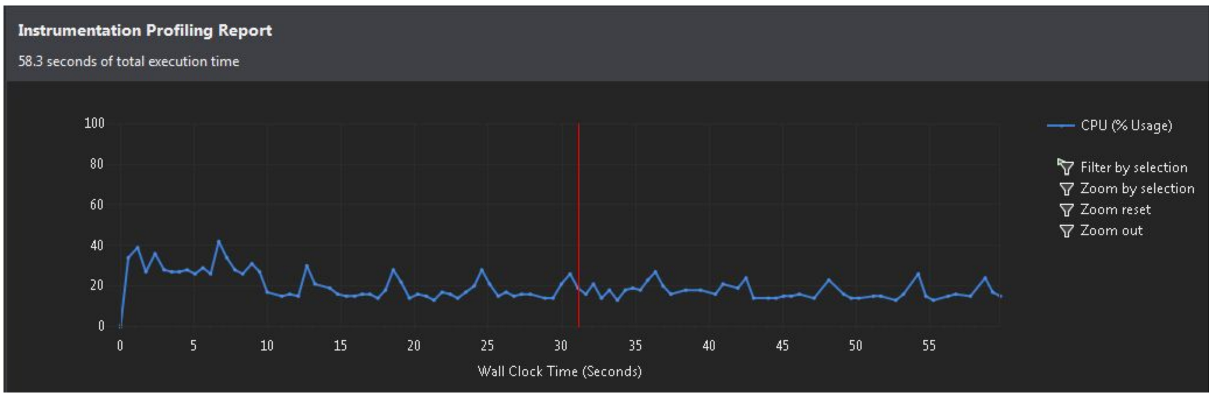
Profiling values:

As mentioned earlier, the ReadInputFile() is where the program reads a list which contains all the bacteria files that are going to be stored and calculated. The bacteria file contains data of all the genes it is made up with.

Profiling using Visual Studio:

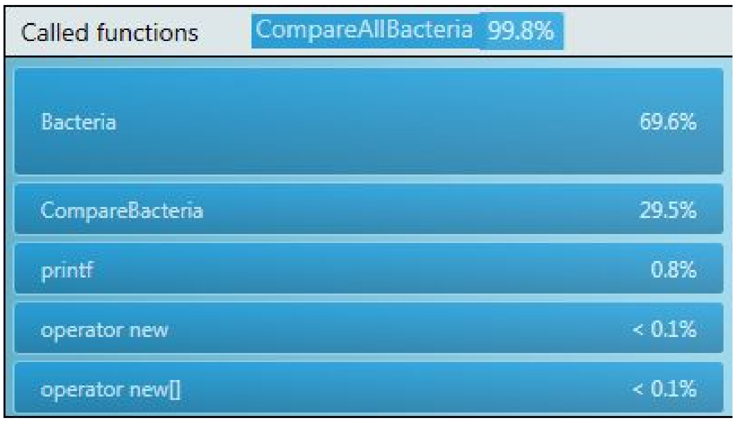
I used the Visual Studio - ‘Instrumental Profile’ tool to gather information about how much work the CPU cores are doing during the runtime of the application. It also presented me with useful information about which sections of the code do the most work.

Picture - 1: CPU Consumption



Hyperthreading and boost combined

Picture - 2: CompareAllBacteria()

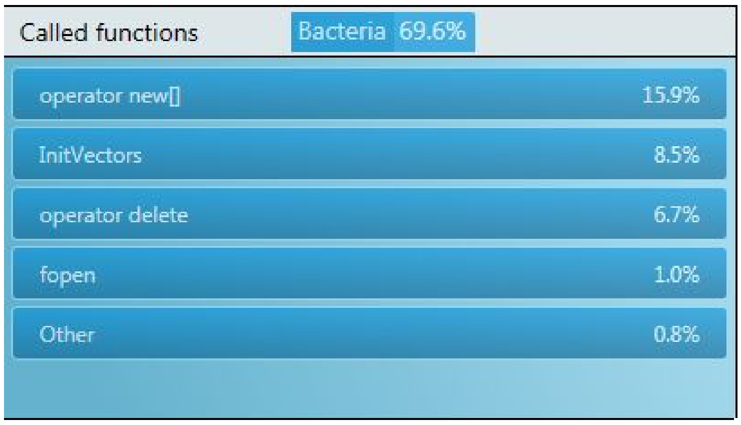


Called functions CompareAllBacteria 99.8%

Bacteria	69.6%
CompareBacteria	29.5%
printf	0.8%
operator new	< 0.1%
operator new[]	< 0.1%

The picture above tells us that 99.8% of the runtime happens in CompareAllBacteria. This is because the function is used as an initiator for all of the CPU intensive functions. The picture also presents the amount of runtime that is used by the called functions. Due to the long runtime in Bacteria and CompareBacteria, it is really clear that the parallelisation should happen at these functions.

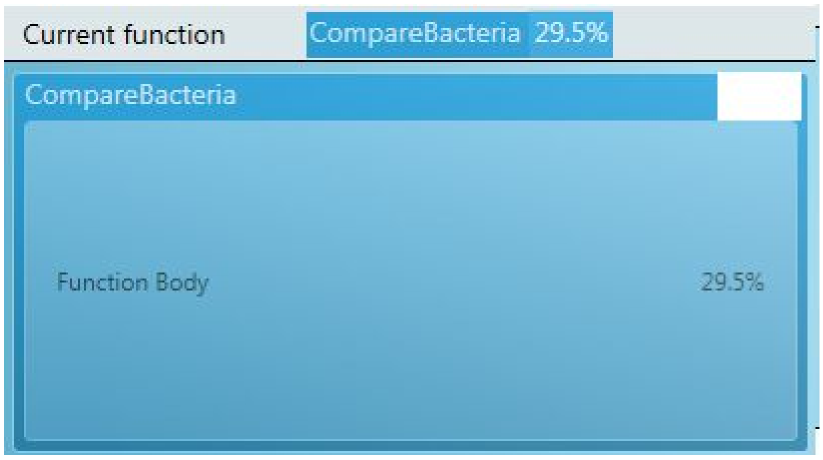
Picture - 3: Bacteria()



Called functions	Bacteria 69.6%
operator new[]	15.9%
InitVectors	8.5%
operator delete	6.7%
fopen	1.0%
Other	0.8%

The Bacteria function uses 69.6% of the runtime. If you subtract this value with all of the called functions you get the percentage of only the Bacteria constructor. Bacteria uses 36.7%.

Picture - 4: CompareAllBacteria()

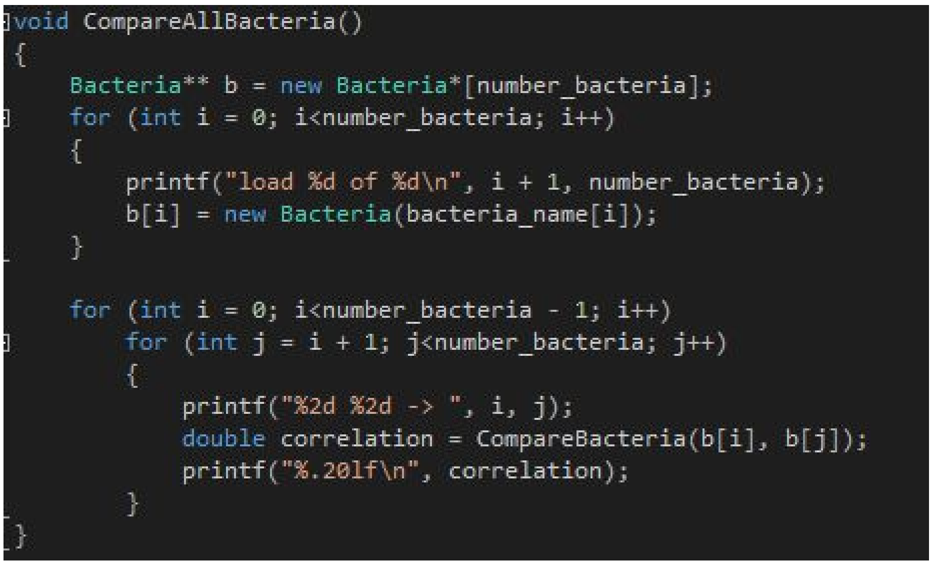


Current function	CompareBacteria 29.5%
CompareBacteria	
Function Body	29.5%

CompareBacteria does not call on any other functions, but uses the total of 29.5%.

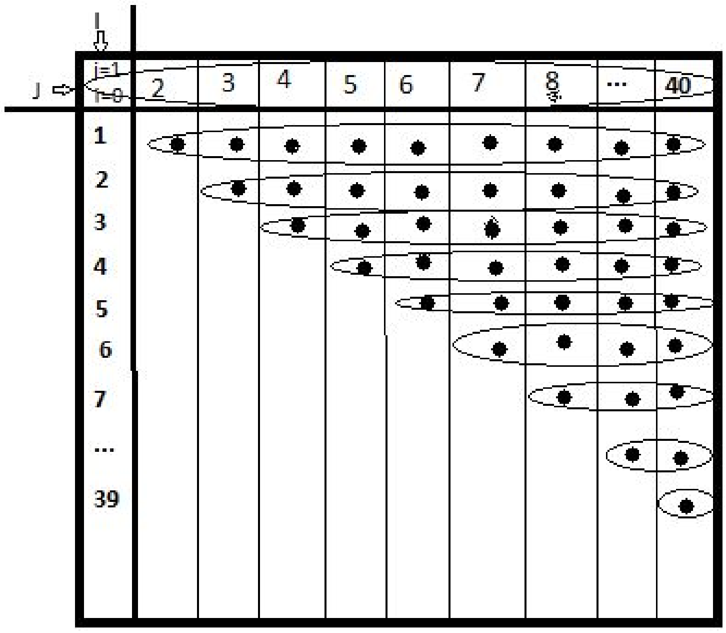
Analyzing and parallelizing the sequential program:

Profiler picture 1 shows us that almost all (99.8%) of the computation time happens in the function CompareAllBacteria.



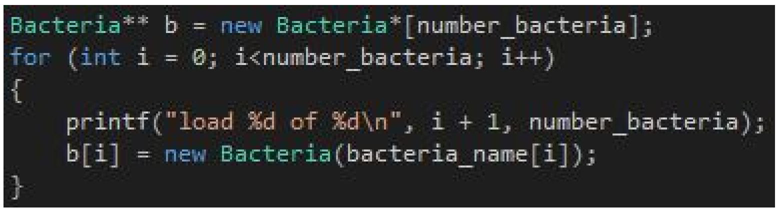
```
void CompareAllBacteria()
{
    Bacteria** b = new Bacteria*[number_bacteria];
    for (int i = 0; i<number_bacteria; i++)
    {
        printf("load %d of %d\n", i + 1, number_bacteria);
        b[i] = new Bacteria(bacteria_name[i]);
    }

    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);
        }
}
```

The CompareAllBacteria class contains 2 ‘for’ loops and these two loops have a flow dependance. To be able to run the second ‘for’ loop, all of the bacteria objects have to be created. This is due to the order the objects are supposed to be compared to each other in. The picture below illustrates the flow of the second for loop. The row is representing the “i” values, and and the columns are representing the “j” values.



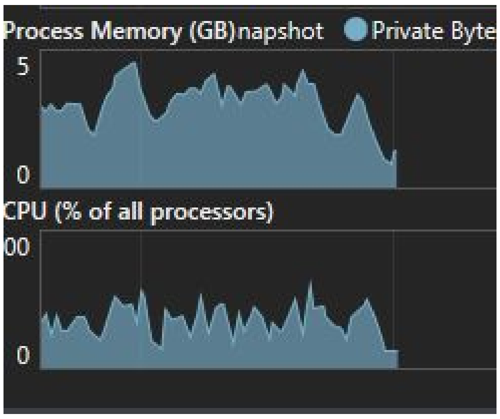
The picture above shows us that that if we're going to compare Bacteria i, we also need all of the bacterias produced after (i+1, i+2+, ...). If the goal was to combine the 2 'for' loops and parallelise after, this would be possible by running a loop in reverse. Then the bacterias could be produced **while** also comparing them, instead of **before**. This would result in a more clunky code, and it wouldn't be significantly faster than parallelising loop by loop. Although, it could potentially save a small amount of memory, as the bacteria objects not in use could be deleted from the memory. However, the saved memory is not significant enough to play a big difference due to the each comparable object b[] is type double (which uses 8 bytes).

Parallelization of the first 'for' loop:

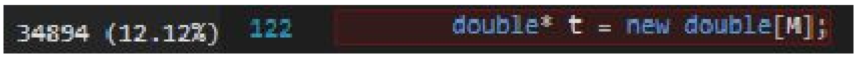


```
Bacteria** b = new Bacteria*[number_bacteria];
for (int i = 0; i<number_bacteria; i++)
{
    printf("load %d of %d\n", i + 1, number_bacteria);
    b[i] = new Bacteria(bacteria_name[i]);
}
```

Since the first 'for' loop already places the results in a 1D array, parallelisation is straightforward. Adding the line "#pragma omp parallel for" before the 'for' loop, instructs computer to divide the 'for' loop into as many threads the computer has available. There is a limit of 41 threads due to the amount of bacterias that are being created (1 thread maximum for each bacteria).

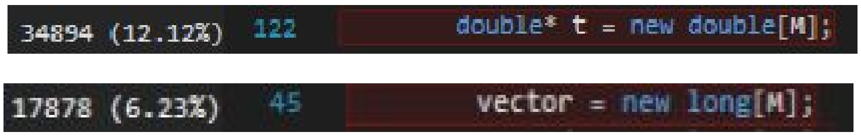


At this point, the bacterias were created faster, but did not utilise all of the core processing power (as shown on the picture to the above). This was due to a memory allocation flaw in the original sequential application. Instead of using the old memory allocation, the code deleted them and assigned them again. This caused the whole program to slow down due to the speed of the memory bus. This will be explained further in “How I overcame performance problems”.



```
34894 (12.12%)   122          double* t = new double[M];
```

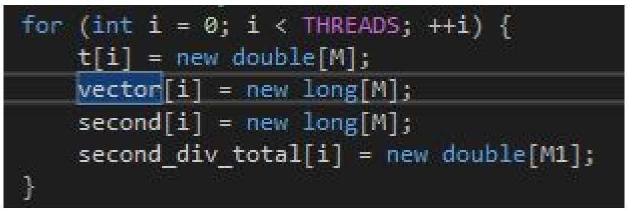
As proven through profiling, the Bacteria class uses ~70% of the overall runtime. By running the visual studio CPU profiler, I was able to see exactly which parts of the code used the most processing time. I noticed that for every time a new bacteria object was constructed, it created big arrays to store temporary information. Instead of reusing these arrays in every object creation, the program was set to delete and initiate a new array every time.



```
34894 (12.12%)   122          double* t = new double[M];
17878 (6.23%)    45             vector = new long[M];
```

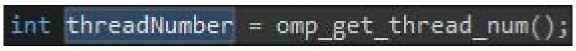
The pictures above shows that over 18% of the time in the class Bacteria was spent on initiating the arrays. The M equals 20^6 = <u>64 million.</u>

To overcome this barrier, I created a public 2D array, where every thread has its own row (see picture below).



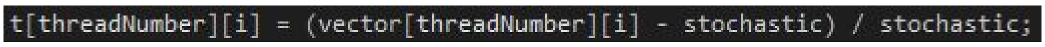
```
for (int i = 0; i < THREADS; ++i) {
    t[i] = new double[M];
    vector[i] = new long[M];
    second[i] = new long[M];
    second_div_total[i] = new double[M1];
}
```

To make these changes work in the Bacteria class i created an int that kept track of which thread is working in the class.



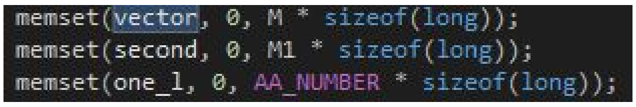
```
int threadNumber = omp_get_thread_num();
```

I can now change the 1D arrays into 2D by adding "[threadNumber][]" as shown below.



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

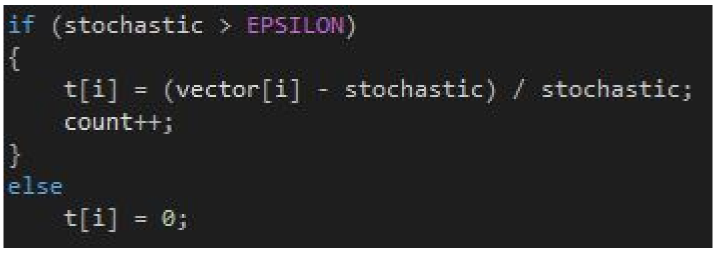
The reason it was possible to reuse these arrays without deleting them is due to the structure in the bacteria class (explained below):



```
memset(vector, 0, M * sizeof(long));
memset(second, 0, M1 * sizeof(long));
memset(one_l, 0, AA_NUMBER * sizeof(long));
```

Memset sets all of the values back to zero during the initiation of the bacteria. Therefore, it will never reuse an old value when the program run.

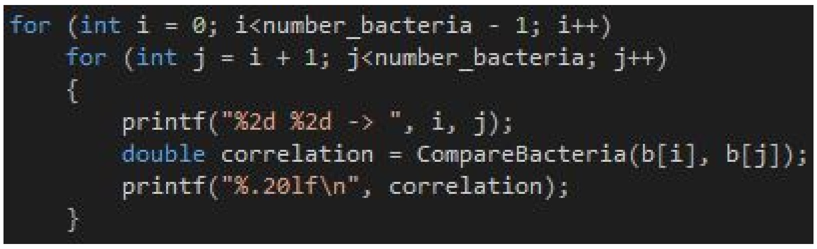
And for the array "t", a 'for' loop runs through all of the data and changes the value of t[i] in every loop. (as shown on the picture below).



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

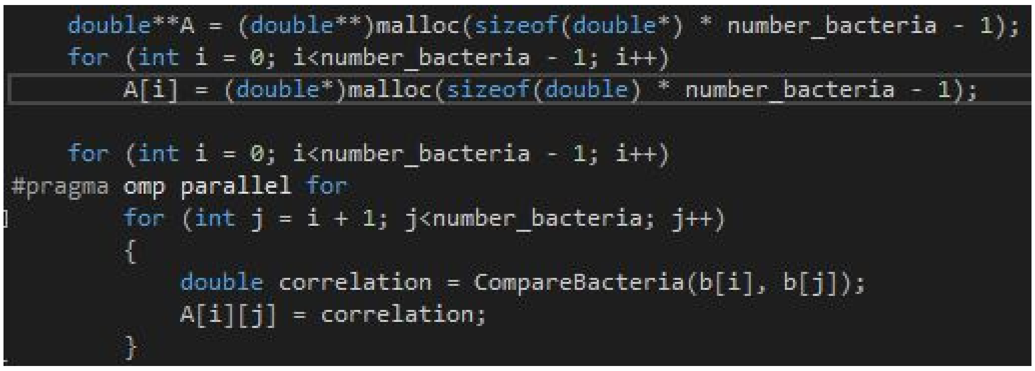
Parallelization of the second 'for' loop:

Second loop:



```
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);
    }
```

This loop handles the comparison and output of the results. In this case, I made inner loop run in parallel with "#pragma omp parallel for". However, since the order had to be the same as the sequential, I created a temporary array to keep track of the original order. The parallel code is shown below:

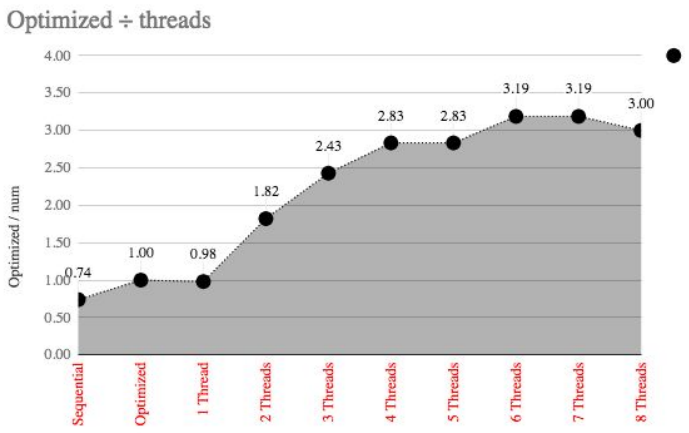


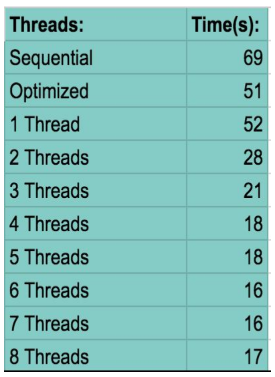
```
    double**A = (double**)malloc(sizeof(double*) * number_bacteria - 1);
    for (int i = 0; i<number_bacteria - 1; i++)
        A[i] = (double*)malloc(sizeof(double) * number_bacteria - 1);

    for (int i = 0; i<number_bacteria - 1; i++)
#pragma omp parallel for
        for (int j = i + 1; j<number_bacteria; j++)
        {
            double correlation = CompareBacteria(b[i], b[j]);
            A[i][j] = correlation;
        }
```

After all of the comparisons have been processed I ran an identical sequential 'for' loop that writes out the information in the array. The printf command is in this case switched out with "fprintf(fp, "%2d %2d -> %.20lf\n", i, j, A[i][j]);" so that it writes it in a txt file, instead of the console.

The parallel version (overall):

Speedup graph:





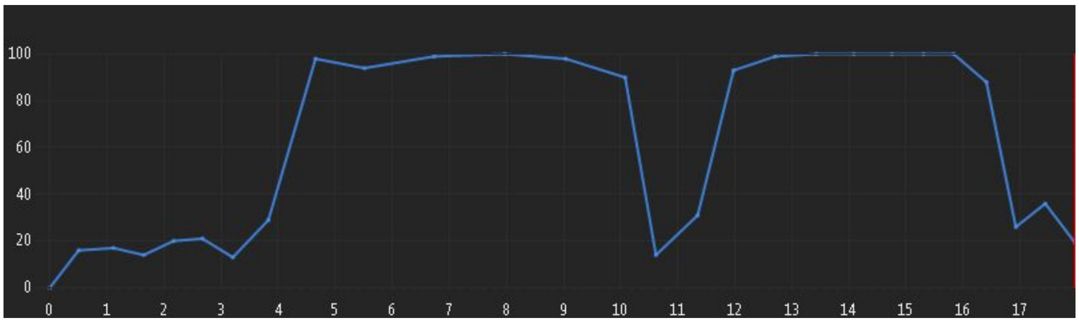
Threads:	Time(s):
Sequential	69
Optimized	51
1 Thread	52
2 Threads	28
3 Threads	21
4 Threads	18
5 Threads	18
6 Threads	16
7 Threads	16
8 Threads	17

The picture above shows the performance gain according to how many threads I assign to the program. The original sequential version had a memory bottleneck that was slowing down the program. I optimised this version making it 26% faster. The number on the y-axis is how much faster the different versions are compared to the optimised sequential version.

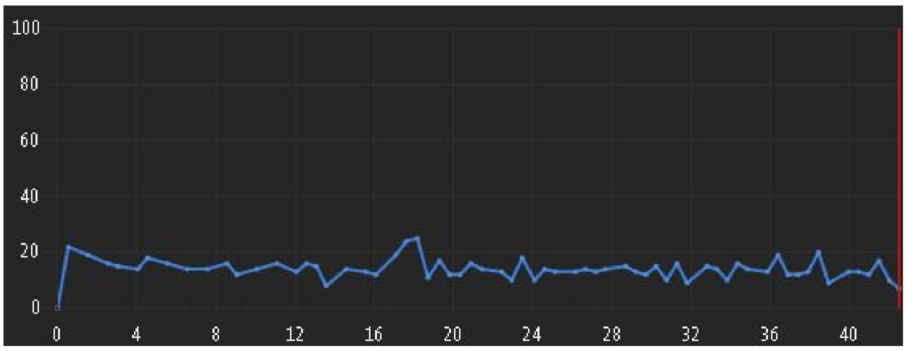
When running the application at different amounts of cores, the best speedup is from 1 -> 4 threads, which is because the computer only has 4 cores.

It also has 4 more virtual cores that works with the help of Hyperthreading. Hyperthreading works by simulating cores by assigning the the unused parts from the other cores. In this way the computer tricks the computer into using more of the core processing power. This resulted in a slight speedup from thread 4->8.

Profiling results:



The first ~3.5 seconds are used for allocating the memory and can therefore not perform faster. The timespan 4->10 seconds are when the Bacterias are created. The remaining time (11s->) of the application are used for comparing and printing out the comparisons.



The optimized original version uses the same amount of CPU processing power throughout the whole runtime which results in a significantly longer processing time.

Scalability:

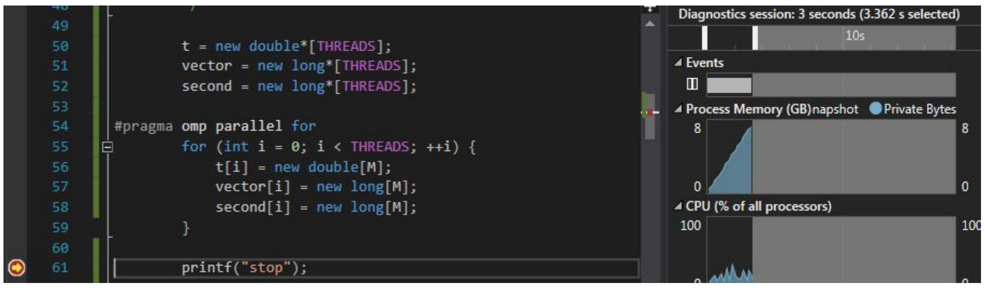
The parallel version of the application is scalable to an extent. Since the Bacteria class is parallel it will only be able to use as many computer cores as there are bacteria to process. The input data for my testing had 41 different bacteria, so in that case it would be possible and effective using 41 different cores. However, if there was more cores than bacterias then it would bottleneck the application.

How I overcame performance problems and barriers:

When working with the parallelisation of the program, all the threads had to store data in the memory. This caused Visual Studio to crash when debugging. This was because of the way Windows handles the allocation of memory. In 32 bit programs there is a limit of 2 GB memory used for the user space. After making the program run in 64 bit, the program can, in theory, use 8 TB of memory for the user space. However, this is limited by the amount of RAM the computer has.

After moving the array of initiators out of the bacteria class and making them global, I discovered a new problem. Since all of code in the loops (in the picture below) were allocating memory, it wasn't possible to speed up with parallelism. This is due to a bottleneck in the memory bus in which the memory bus cannot transfer above a certain speed. Running this in parallel will only cause overhead. The 2 pictures below show us that both the parallel and the sequential loop runs at 3.36sec. However, in the parallel version, the cpu performance is still a bit higher (overhead).

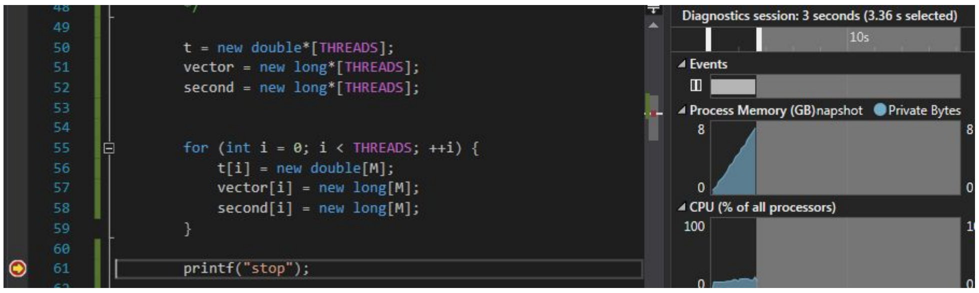
Parallel loop:



```
        t = new double*[THREADS];
        vector = new long*[THREADS];
        second = new long*[THREADS];

#pragma omp parallel for
        for (int i = 0; i < THREADS; ++i) {
            t[i] = new double[M];
            vector[i] = new long[M];
            second[i] = new long[M];
        }

        printf("stop");
```

Diagnostics session: 3 seconds (3.362 s selected)
10s
Events
Process Memory (GB)napshot
Private Bytes
8
0
CPU (% of all processors)
100
0

Sequential loop:



```
        t = new double*[THREADS];
        vector = new long*[THREADS];
        second = new long*[THREADS];


        for (int i = 0; i < THREADS; ++i) {
            t[i] = new double[M];
            vector[i] = new long[M];
            second[i] = new long[M];
        }

        printf("stop");
```

Diagnostics session: 3 seconds (3.36 s selected)
10s
Events
Process Memory (GB)napshot
Private Bytes
8
0
CPU (% of all processors)
100
0

Since all of the memory allocated needs to be used in the first CPU intensive function, it is not possible to divide the allocation over time.

Reflection of the outcome:

After parallelising the program I learned a lot about how much small sections of code can slow down the overall runtime of a program. I have also learned that memory can, in many cases, be a big bottleneck. Memory allocation cannot run in parallel due to the speed of the memory bus connecting the CPU and RAM.

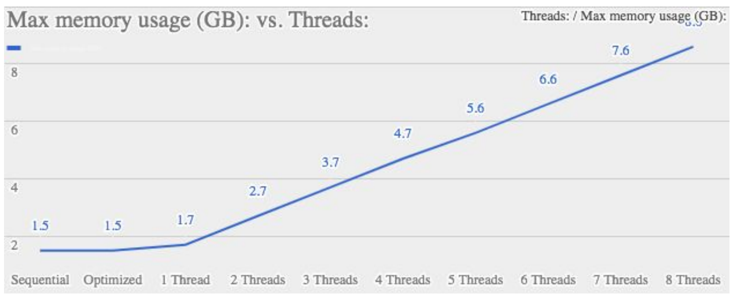
Another thing that I noticed when parallelising a sequential program is that if the coder is not creating a program with the thought of parallelizing it, the code can be really hard to parallelise. This is because the sequential version might have been built in a way that requires all of the code to be run in order. When this is the case, it will require a lot more work to make it parallel.

How successful was my attempt:

My attempt was really successful. I managed to make the application run 3.2 times faster than the optimised sequential version. The theoretical speedup that could be achieved was, at maximum, 4 times as fast (due to using 4 cores). If I also found a way to make the application use less memory, or be able to divide the memory allocation throughout the program instead of in the beginning, the speed up could be significantly higher.

The way I parallelised it could be done differently if I wanted to reduce the amount of RAM necessary for the program to run. I could, for instance, parallelise inside the bacteria function instead. In this case, it would be more difficult to utilise all of the CPU speed. I might've not been able to make the function run at ~90% CPU power, as I did with my Bacteria creation.

My parallelisation demands a lot of available memory from the computer, which can be a big issue if it's run on a lower end computer. For every thread that is initiated an additional 1gb of RAM is needed. The picture below illustrates the connection between threads and memory usage. The x-axis is the amount of threads, while the y-axis is the amount of memory required.



The reason for why it need +1GB for each Thread is because every thread instance of a bacteria creation needs this for allocating data.