CAB401 Parallelisation Project

Bismillah Sultani | N11247282

Introduction

Bioinformatics is a field that merges biology, computer science, and information technology to analyse and interpret biological data. One important focus within bioinformatics is comparative genomics, which involves examining the genetic material of various organisms to uncover evolutionary relationships, shared ancestry, and genetic markers. In this context, the CVTree app is a tool developed to analyse bacterial genomes by identifying genetic similarities across species. It achieves this by breaking down each genome into shorter segments, known as k-mers, to capture recurring genetic patterns. By comparing these k-mer frequencies across bacterial genomes, the given app determines genetic likeness, offering insights into evolutionary connections. Such analysis is crucial for understanding bacterial lineage, developing treatments for infections, and tracking genetic variations over time.

The current implementation of the application operates on a single core, resulting in considerable processing time. This limitation prompted the parallelisation of the application to enhance computational efficiency and reduce runtime. The application performs intensive computational tasks, particularly during the comparison phase, where each bacterium is compared pairwise with others. These comparisons involve extensive k-mer distribution calculations across potentially large datasets, making them computationally demanding. Due to nested loops, this phase incurs significant processing delays. By parallelising the application, the computational load can be distributed across multiple processors, substantially decreasing runtime and rendering the application more suitable for large-scale analysis.

Analysis of potential Parallelism

To understand functional dependencies and the application’s workflow, I have constructed a schema of the original application, showing a high-level overview of the main functions and loops responsible for the application's operation.

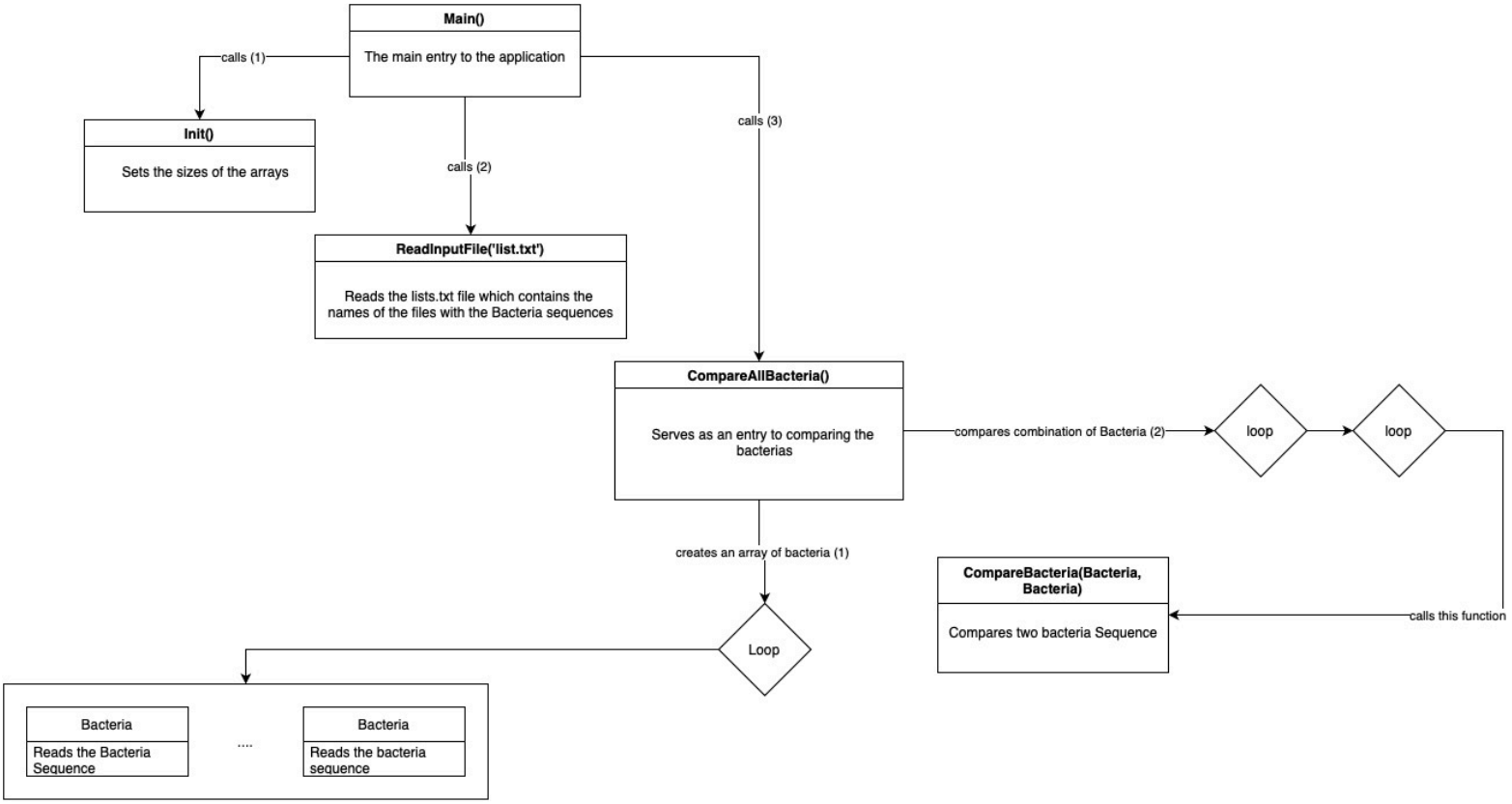


Figure 1: A schema showing the main functions and loops of the original application and their dependencies. Dependencies are numbered such that dependency 2 happens after 1.

The diagram (above) shows the workflow of the application and also where performance bottlenecks may arise. It starts with a Main() function, serving as the entry to the program. From there, other functions, such as Init(), are called to handle tasks like initialising arrays and reading files. The Main() function then calls CompareAllBacteria(), which is responsible for comparing bacteria data files. This function uses looping structures to create a bacteria array and then perform comparisons between the bacteria sequences.

Potential inefficiencies may arise in the creation and comparison of the bacteria array, both of which rely heavily on sequential looping structures. These loops could become performance bottlenecks, particularly as the dataset grows.

As such, I will explore in greater detail what is involved in creating a Bacteria instance and how the bacteria sequences are compared within the application with the CompareBacteria() function.

Bacteria Constructor

The main computation in the Bacteria Instance happens within its constructor.

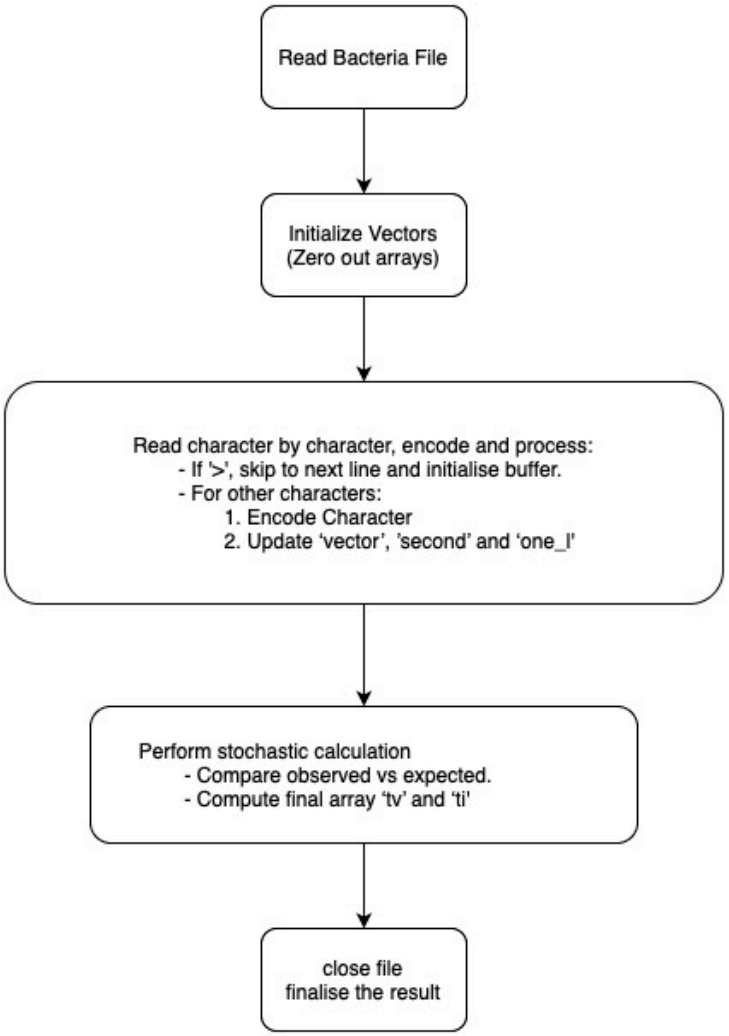


Figure 2: A schema of the Bacteria constructor.

As it can be seen in the figure above, the current implementation of the Bacteria class processes data from a file sequentially, performing several key operations that could introduce performance bottlenecks when handling large bacteria equences .

The process begins by reading the bacteria file and initialising arrays to track the frequency of encoded sequences. As each character is read and encoded, the frequency arrays are updated. This character-by-character processing can become a bottleneck, especially when dealing with large files, due to the need to constantly update these arrays.

Then stochastic calculation loops happen, where observed sequence patterns are compared to expected stochastic values. This involves iterating the created arrays and performing multiple computations per iteration, which can become resource-intensive as the dataset grows.

Additionally, memory management, such as frequent allocations and deallocations of large arrays, may further contribute to performance delays. These areas could lead to slowdowns in the overall performance of the class, particularly when dealing with extensive biological sequence data.

CompareBacteria Function

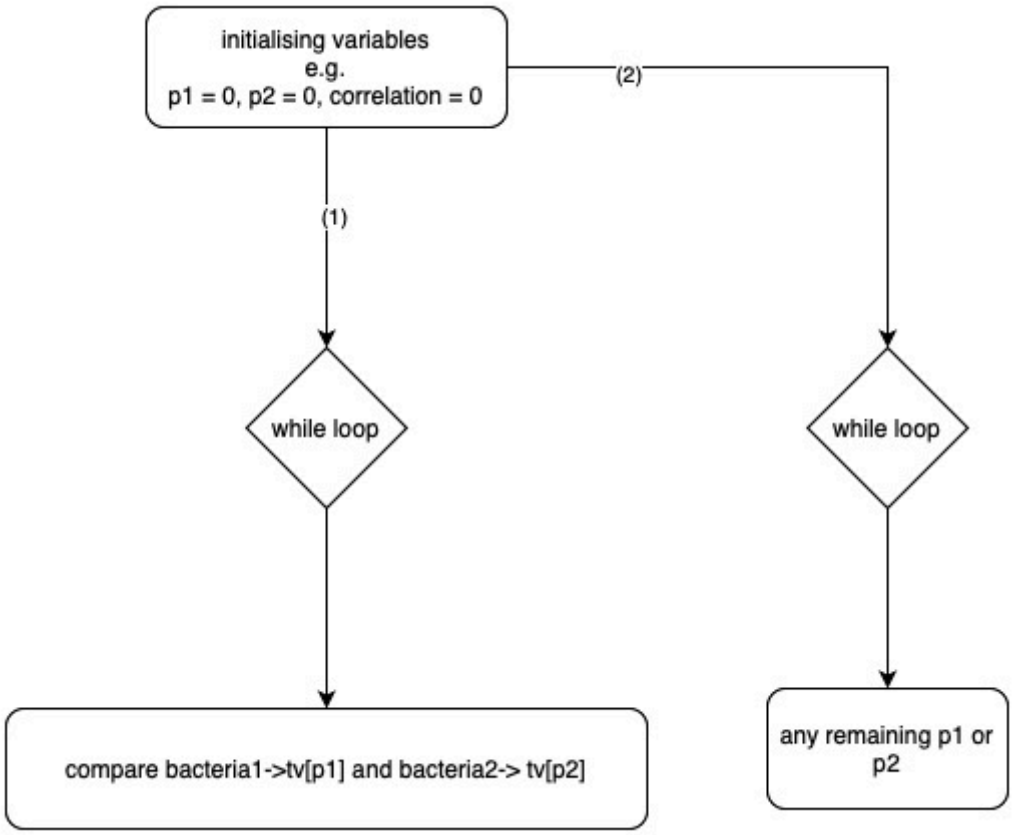


Figure 3: A schema showing the main loops of the CompareBacteria function.

The CompareBacteria function is designed to compute the correlation between two bacteria based on their precomputed stochastic values. The function begins by intialisng key variables. The key variables in this function are correlation, which accumulates the similarity score between the two bacteria. The function uses two pointers, p1 and p2, to traverse the tv arrays, which hold the precomputed values for each bacterium.

The function enters a while loop that iterates as long as both pointers are within the bounds of their respective bacteria's data. At each iteration, the function compares the indices from the two bacteria (b1->ti[p1]and b2->ti[p2]) and computes the correlation. After exiting the main loop, the function checks if any elements remain in either bacterium's tv array and continues updating vector_len1 or vector_len2 accordingly. Finally, the correlation value is returned.

Heavy bottlenecks could arise in the comparison loop, where each step requires accessing and comparing indices, followed by computations that involve squaring values and updating multiple variables. This character-by-character processing, combined with the need to traverse the `tv` arrays sequentially, could significantly slow down the function, especially for large arrays.

In the next section, I will examine the potential impact of these loops on the application. Depending on the severity, appropriate actions will be taken to address the original application.

Performance Bottlenecks and Profiling Results (sequential)

While analysing the original application gave significant insight into where potential inefficiencies may occur, a more concret approach was needed to the severity of these potential efficiencies. By utilising *'gprof'*, (Fenlason, n.d.) a widely-used profiling tool, I was able to effectively analyse the execution time of each function. The profiling results highlighted two functions, responsible for the bottleneck:

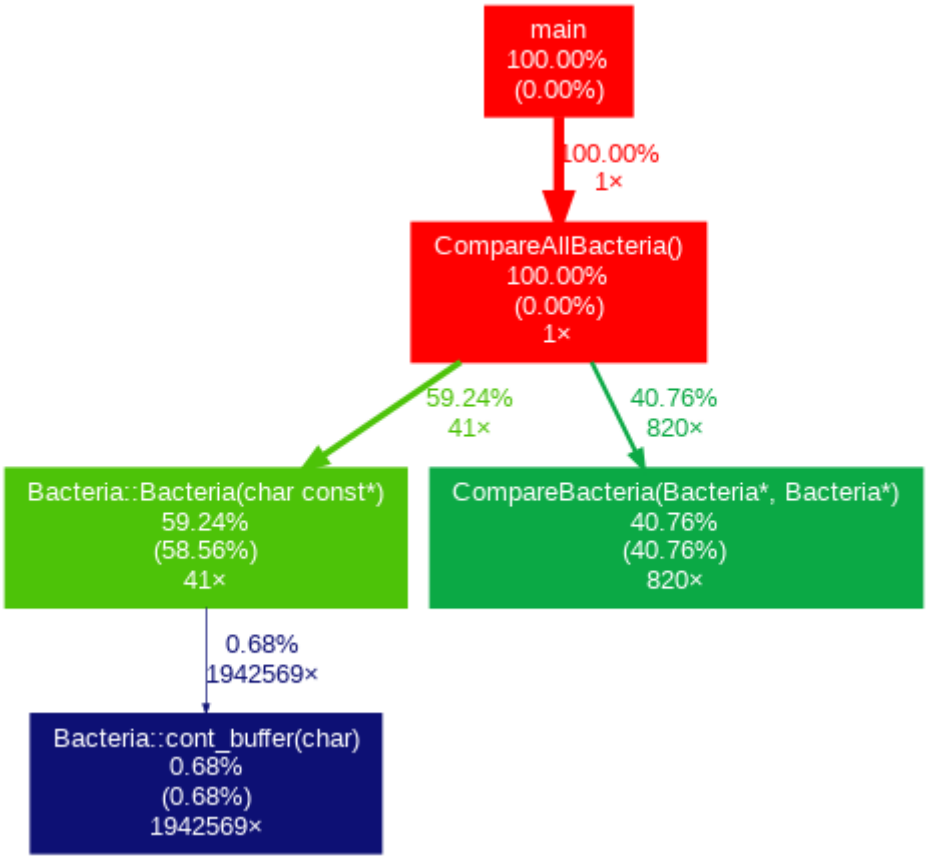


Figure 4: A graphic display of the results from gprof. Full result from gprof is shown in appendix 1.

As the graph in Figure 4 illustrates the call graph of key functions within the application and illustrates their overall effects. The call graph begins with the main function, which accounts for 0% of self-execution time, indicating that it primarily serves as a coordinator for subsequent function calls. This function then invokes CompareAllBacteria, which also registers 0% self-execution time. Within CompareAllBacteria, the Bacteria:Bacteria constructor is instantiated, followed by a call to the CompareBacter function on the instantiated bacteria

The Bacteria::Bacteria constructor accounts for 58.56% of the total CPU time, highlighting it as a significant performance bottleneck, despite being called only 41 times overall. This means that each call has significant performance issues. Unexpectedly, the file reading process accounts for only 0.68% of the constructor's execution time, as shown in the figure. This indicates that 99% of the bottleneck comes from sequentially processing stochastic

calculations. This is further supported by the screenshot below, where the timing of each loop is shown. The loops involving stochastic calculations take significantly longer than the file reading process.

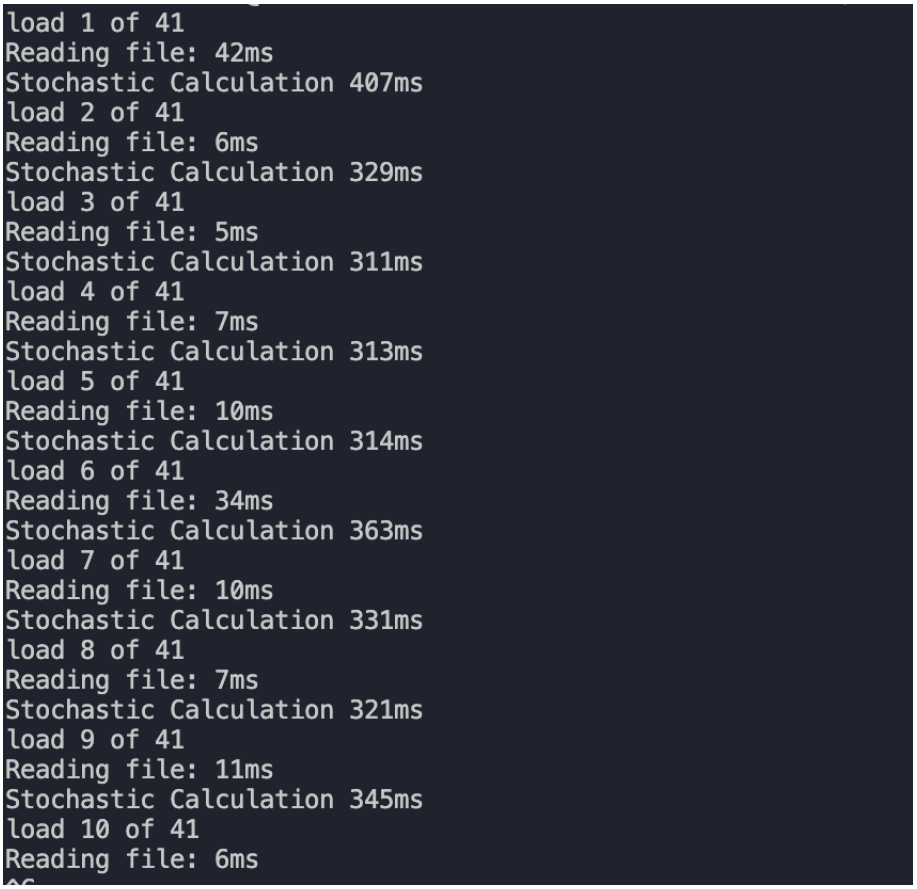


Figure 5: A snapshot of the timing for stochastic calculation for a bacteria file

And the CompareBacteria function accounts for 40.76% of the execution time, but it is also called substantially more, 820 times, indicating high frequency of calls contributing to high execution time. The main loop inside the function executes quickly and does not significantly contribute to the overall execution time:

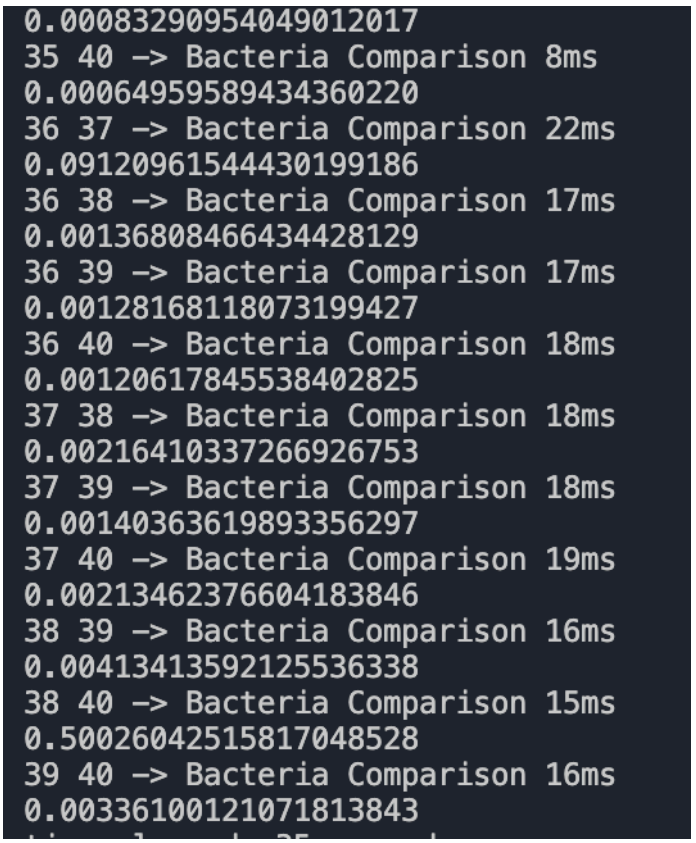


Figure 6: A snapshot of compare loop within the compare bacteria function.

Overall the application took 39 seconds to run.

Figure 7: A screenshot showing the amount of time the sequential program ran.

Overall, key areas that stand to benefit significantly from parallelisation include file reading, stochastic computations, and file comparisons.

Parallelisation Strategy

Having identified major regions of inefficiencies, in this section I will analyse feasibility of parallelisation, by discussing data and control dependencies, granularity and scalability of parallelisaiton, as well as the code restructure needed for parallelisation.

Bacteria File Reading

Each bacteria file is independent, with no data or control dependencies between them. This allows for parallel reading and processing without concerns about synchronisation or race conditions. Processing entire files represents a coarse-grained level of parallelism, which is highly beneficial for scalability.

```
278     #pragma omp parallel for
279     for(int i=0; i<number_bacteria; i++)
280     {
281         printf("load %d of %d\n", i+1, number_bacteria);
282         b[i] = new Bacteria(bacteria_name[i]);
283     }
```

Figure 7: showing the loop responsible for loading the bacteria along with an added OpenMP imperative to parallelise it.

The easiest way to achieve parallelisation is to use #pragma omp parallel for, as seen in the figure above, a directive from OpenMP which instructs the compiler to divide the loop iterations among multiple threads, enabling concurrent execution.

Stochastic Calculations

Another way where parallelisation could improve the performance of the application is the stochastic calculation loop, however there are several data dependencies due to the sequential updates of the index variables (i_mod_aa_number, i_div_aa_number, i_mod_M1, and i_div_M1). These variables are incremented within each iteration and carry over to the next, creating a true data dependency because each iteration depends on the state of these variables from the previous one. This makes it difficult to parallelise the loop safely, as race conditions or memory access issues could occur. Additionally, the shared count variable, which is updated in each iteration, poses a potential risk for race conditions in a parallel setting without proper handling.

```
139         for(long i=0; i<M; i++)
140         {
141             double p1 = second_div_total[i_div_aa_number];
142             double p2 = one_l_div_total[i_mod_aa_number];
143             double p3 = second_div_total[i_mod_M1];
144             double p4 = one_l_div_total[i_div_M1];
145             double stochastic =  (p1 * p2 + p3 * p4) * total_div_2;
146 
147             if (i_mod_aa_number == AA_NUMBER-1)
148             {
149                 i_mod_aa_number = 0;
150                 i_div_aa_number++;
151             }
152             else
153                 i_mod_aa_number++;
154 
155             if (i_mod_M1 == M1-1)
156             {
157                 i_mod_M1 = 0;
158                 i_div_M1++;
159             }
160             else
161                 i_mod_M1++;
162 
163             if (stochastic > EPSILON)
164             {
165                 t[i] = (vector[i] - stochastic) / stochastic;
166                 count++;
167             }
168             else
169                 t[i] = 0;
170         }
171 
```

Figure 8: Code for stochastic calculation in the original code.

To resolve these data dependencies and make the loop parallelisable, I removed the need for sequential updates by directly calculating the indices (i_mod_aa_number, i_div_aa_number, i_mod_M1, and i_div_M1) based on the loop variable i. By computing these indices as functions of i using modulo and division operations, each iteration becomes independent of the others. I also handled the shared count variable using an OpenMP reduction clause, ensuring thread-safe accumulation across iterations. These changes eliminated the data dependencies and enabled safe parallelisation, allowing each thread to process a different part of the data without interference.

```
138         #pragma omp parallel for reduction(+:count)
139         for (long i = 0; i < M; i++) {
140             int i_mod_aa_number = i % AA_NUMBER;
141             int i_div_aa_number = (i / AA_NUMBER) % M1;
142             int i_mod_M1 = i % M1;
143             int i_div_M1 = i / M1;
144 
145             double p1 = second_div_total[i_div_aa_number];
146             double p2 = one_l_div_total[i_mod_aa_number];
147             double p3 = second_div_total[i_mod_M1];
148             double p4 = one_l_div_total[i_div_M1];
149 
150             double stochastic = (p1 * p2 + p3 * p4) * total_div_2;
151 
152             if (stochastic > EPSILON) {
153                 t[i] = (vector[i] - stochastic) / stochastic;
154                 count++;
155             } else {
156                 t[i] = 0;
157             }
158         }
159 
```

Figure 9: proposed code restructuring for the stochastic calculation with parallelisaiton.

The code restructuring required for this was adding a parallelisaiton directive outside the loop.

Comparing Bacteria

Comparing files has nested loops, i and j. The i and j nested loops are good candidates for parallelisation because there are no data dependencies between iterations, as each (i, j) pair operates independently when comparing bacteria. The control dependency, where j starts at i + 1, is straightforward to handle, ensuring each pair is only processed once. Given that each comparison likely involves significant computation, the loop granularity is coarse, making it suitable for parallelisation with minimal overhead. With the number of iterations growing quadratically (O(n^2)), the scalability is high, making this loop ideal for parallel processing, though any console print may not be in order.

```
285     #pragma omp parallel for
286     for(int i=0; i<number_bacteria-1; i++)
287         #pragma omp parallel for
288         for(int j=i+1; j<number_bacteria; j++)
289         {
290             printf("%2d %2d -> ", i, j);
291             double correlation = CompareBacteria(b[i], b[j]);
292             printf("%.20lf\n", correlation);
293         }
```

Figure 10: proposed code restructure for comparing bacteria (loops).

Following the parallelisation of the above code, the execution achieved a substantial performance improvement, completing in only 8 seconds - a nearly fivefold increase in speed compared to the original implementation.

Figure 11: the amount of time the program took to ran after elementary loop parallelisaiton.

Besides this, I explored alternative strategies to further enhance the application's and overcome performance barriers, specifically exploring another method of parallelisation, a consumer and producer model for parallelisation.

To implement the producer-consumer model in the bacteria comparison application, I refactored the code to separate the tasks of loading bacteria data (producer role) and comparing bacteria data (consumer role). This approach optimises resource usage and allows data processing and comparison to proceed simultaneously, rather than waiting for all data to load before initiating comparisons. With this I halved the previous time to only 4 seconds. Here’s how I implemented this model, the code changes made, and the benefits achieved.

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

    std::vector<std::string> results;

    #pragma omp parallel for
    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]);

            std::ostringstream result;
            result << std::setw(2) << i << " " << std::setw(2) << j << " -> ";
            result << std::fixed << std::setprecision(20) << correlation;

            #pragma omp critical
            results.push_back(result.str());
        }
    }

    for (int i = 0; i < number_bacteria; i++)
    {
        delete b[i];
    }
    delete[] b;

    for (const auto& result : results)
    {
        std::cout << result << std::endl;
    }
}
```

Figure 12: Original code for CompareAllBacteria function.

In the original implementation, the CompareAllBacteria function operated in a straightforward two-phase manner. Initially, it employed OpenMP's #pragma omp parallel for directive to load all bacteria data concurrently into an array of Bacteria objects. Each thread was responsible for instantiating a Bacteria object from the provided filenames, effectively distributing the workload across multiple cores. Once all bacteria were loaded, the function proceeded to perform pairwise comparisons using nested OpenMP parallel loops. The outer loop iterated through each bacteria, while the inner loop compared it with every subsequent bacteria in the array. This approach worked well; however, bacteria comparisons had to wait until all the bacteria were fully loaded. This sequential dependency meant that the comparison phase could not commence until the loading phase was entirely complete, potentially leading to inefficiencies, especially when dealing with a large number of bacteria or substantial data files, possibly resulting in suboptimal CPU utilisation and longer overall execution times.

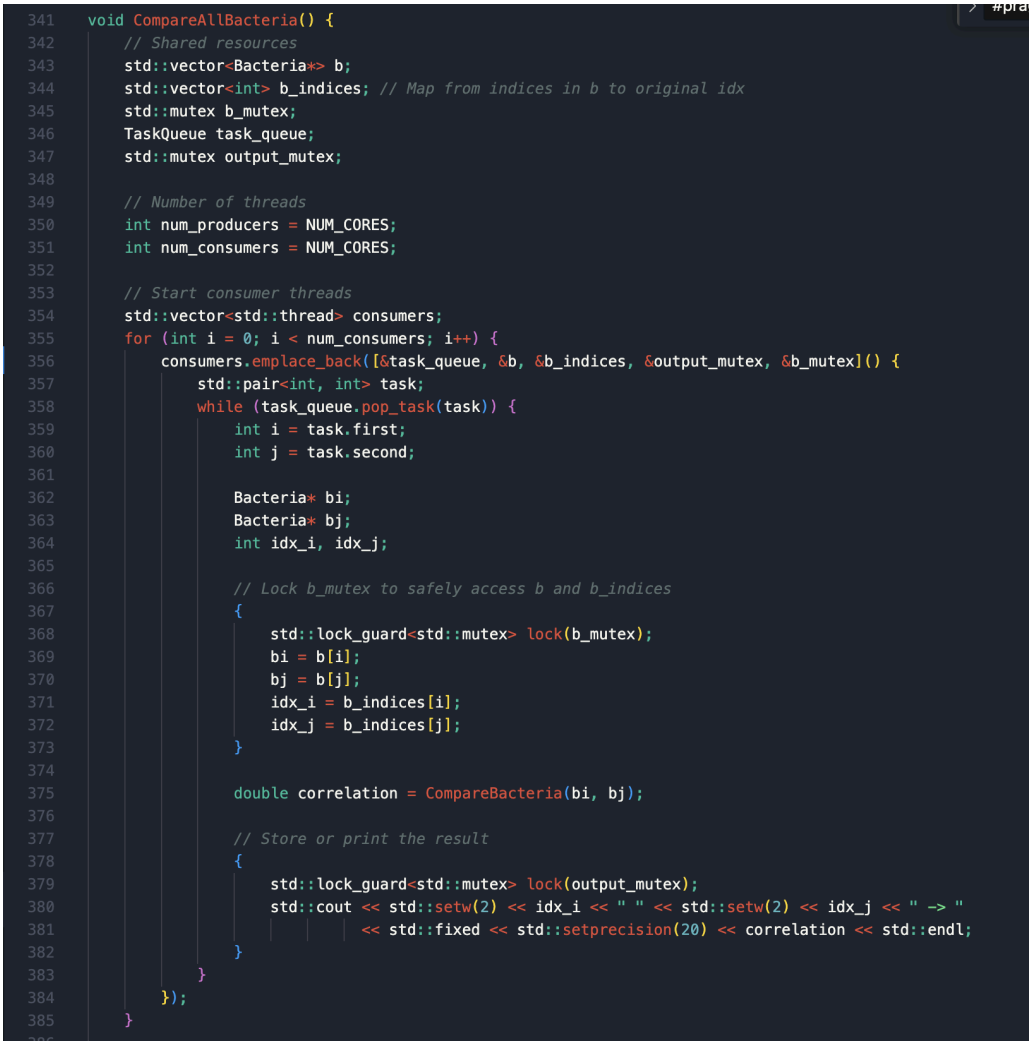
The primary motivation for this change was to enable concurrent loading of bacteria data and the initiation of pairwise comparisons, thereby overlapping these two intensive tasks to improve overall performance and resource utilisation.

```
class TaskQueue {
private:
    std::queue<std::pair<int, int>> tasks;
    std::mutex mtx;
    std::condition_variable cv;
    bool done = false;

public:
    void push_task(const std::pair<int, int>& task) {
        {
            std::lock_guard<std::mutex> lock(mtx);
            tasks.push(task);
        }
        cv.notify_one();
    }

    bool pop_task(std::pair<int, int>& task) {
        std::unique_lock<std::mutex> lock(mtx);
        while (tasks.empty() && !done) {
            cv.wait(lock);
        }
        if (!tasks.empty()) {
            task = tasks.front();
            tasks.pop();
            return true;
        } else {
            return false;
        }
    }

    void set_done() {
        {
            std::lock_guard<std::mutex> lock(mtx);
            done = true;
        }
        cv.notify_all();
    }
};
```

Figure 13: Added code for keeping track of tasks.

The code restructuring required included a producer-consumer model facilitated by a newly implemented TaskQueue class. This class managed a thread-safe queue of comparison tasks, allowing producer threads to load Bacteria objects and enqueue corresponding comparison tasks dynamically. Concurrently, consumer threads dequeue these tasks and perform the necessary comparisons. By doing so, comparisons can commence as soon as sufficient data is available, without waiting for the entire loading phase to finish. This overlapping of tasks not only reduces idle time but also ensures that CPU resources are more effectively utilised throughout the program's execution.

There were also huge restructuring in CompareAllBacteria functions which leveraged mutexes and condition variables to manage synchronisation between threads, ensuring thread-safe operations when accessing shared resources such as the bacteria vector and output streams. Instead of accumulating all comparison results in a shared vector for later output, the revised function opts to print results immediately within consumer threads. This real-time output reduces memory overhead and provides instant feedback on the comparison process.



```cpp
void CompareAllBacteria() {
    // Shared resources
    std::vector<Bacteria*> b;
    std::vector<int> b_indices; // Map from indices in b to original idx
    std::mutex b_mutex;
    TaskQueue task_queue;
    std::mutex output_mutex;

    // Number of threads
    int num_producers = NUM_CORES;
    int num_consumers = NUM_CORES;

    // Start consumer threads
    std::vector<std::thread> consumers;
    for (int i = 0; i < num_consumers; i++) {
        consumers.emplace_back([&task_queue, &b, &b_indices, &output_mutex, &b_mutex]() {
            std::pair<int, int> task;
            while (task_queue.pop_task(task)) {
                int i = task.first;
                int j = task.second;

                Bacteria* bi;
                Bacteria* bj;
                int idx_i, idx_j;

                // Lock b_mutex to safely access b and b_indices
                {
                    std::lock_guard<std::mutex> lock(b_mutex);
                    bi = b[i];
                    bj = b[j];
                    idx_i = b_indices[i];
                    idx_j = b_indices[j];
                }

                double correlation = CompareBacteria(bi, bj);

                // Store or print the result
                {
                    std::lock_guard<std::mutex> lock(output_mutex);
                    std::cout << std::setw(2) << idx_i << " " << std::setw(2) << idx_j << " -> "
                              << std::fixed << std::setprecision(20) << correlation << std::endl;
                }
            }
        });
    }
```

Figure 14: The top snippet of CompareAllBacteria function showing creation of consumer threads.

```
    // Start producer threads
    std::vector<std::thread> producers;
    std::atomic<int> next_bacteria_index(0);

    for (int i = 0; i < num_producers; i++) {
        producers.emplace_back([&]() {
            int idx;
            while ((idx = next_bacteria_index.fetch_add(1)) < number_bacteria) {
                Bacteria* bi = new Bacteria(bacteria_name[idx]);

                std::vector<std::pair<int, int>> new_tasks;

                int b_index;
                int existing_size;

                // Lock b to add bi and generate tasks
                {
                    std::lock_guard<std::mutex> lock(b_mutex);
                    b_index = b.size();
                    b.push_back(bi);
                    b_indices.push_back(idx); // Map b_index to idx
                    existing_size = b.size();
                }

                // Generate comparison tasks with all existing bacteria except itself
                for (int j = 0; j < existing_size; j++) {
                    if (j != b_index) {
                        // Push the task immediately
                        task_queue.push_task(std::make_pair(b_index, j));
                    }
                }
            }
        });
    }

    // Wait for producers to finish
    for (auto& t : producers) {
        t.join();
    }

    // Signal that no more tasks will be added
    task_queue.set_done();

    // Wait for consumers to finish
    for (auto& t : consumers) {
        t.join();
    }

    // Cleanup
    for (auto& bi : b) {
        delete bi;
    }
}
```

Figure 15: The bottom snippet of CompareAllBacteria function showing creation of producer threads.

I also tried using non-prallelisation methods to improve the performance:

I optimised the file reading process by modifying the program to load each bacteria data file entirely into memory before processing. Instead of reading the file character by character or line by line - which involves multiple disk I/O operations and can be inefficient for large files - I utilised the fseek and ftell functions to determine the total file size. With this information, I allocated a buffer large enough to hold the entire file content and used fread to read the file into memory in a single operation. This method reduces the overhead associated with disk access by minimising the number of I/O calls. Processing the data directly from memory not only accelerates the file reading but also improves data locality, leading to better cache utilisation.

```
94
95      Bacteria(const char* filename)
96      {
97          FILE* bacteria_file = fopen(filename, "r");
98
99          if (bacteria_file == NULL)
100         {
101             fprintf(stderr, "Error: failed to open file %s\n", filename);
102             exit(1);
103         }
104
105         InitVectors();
106
107         char ch;
108         while ((ch = fgetc(bacteria_file)) != EOF)
109         {
110             if (ch == '>')
111             {
112                 while (fgetc(bacteria_file) != '\n' && !feof(bacteria_file)); // skip rest of line
113
114                 char buffer[LEN-1];
115                 if (fread(buffer, sizeof(char), LEN-1, bacteria_file) == LEN-1) {
116                     init_buffer(buffer);
117                 }
118             }
119             else if (ch != '\n')
120                 cont_buffer(ch);
121         }
```

Figure 16: Bacteria file reading original code.

```
95      Bacteria(const char* filename)
96      {
97          FILE* bacteria_file = fopen(filename, "rb");
98          if (bacteria_file == NULL)
99          {
100             fprintf(stderr, "Error: failed to open file %s\n", filename);
101             exit(1);
102         }
103         InitVectors();
104         fseek(bacteria_file, 0, SEEK_END);
105         long file_size = ftell(bacteria_file);
106         fseek(bacteria_file, 0, SEEK_SET);
107
108         char* buffer = new char[file_size + 1];
109
110         // Read file into buffer
111         size_t read_size = fread(buffer, 1, file_size, bacteria_file);
112         buffer[read_size] = '\0';
113
114         fclose(bacteria_file);
115
116         char* p = buffer;
117         char* buffer_end = buffer + read_size;
118
119         while (p < buffer_end)
120         {
121             char ch = *p++;
122             if (ch == '>')
123             {
124                 while (p < buffer_end && *p != '\n') p++;
125                 if (p < buffer_end) p++; // Skip '\n'
126                 // Read next LEN-1 characters into seq_buffer, skipping '\n'
127                 char seq_buffer[LEN-1];
128                 int i = 0;
129                 while (p < buffer_end && i < LEN-1)
130                 {
131                     if (*p != '\n')
132                     {
133                         seq_buffer[i++] = *p;
134                     }
135                     p++;
136                 }
137                 if (i == LEN-1)
138                 {
139                     init_buffer(seq_buffer);
140                 }
141             }
142             else if (ch != '\n')
143             {
144                 cont_buffer(ch);
```

Figure 17: Bacteria file reading improved.

Besides this, I implemented large arrays as std::vector for dynamic allocation and memory management. However, this introduced delays and didn't improve the project's performance and as such were not included in the final implementation.

To deal with load balancing, I also implemented dynamic scheduling. This approach allows threads to dynamically receive smaller chunks of iterations, enabling idle threads to pick up remaining tasks as others finish, thereby balancing the workload more effectively and minimising idle time across threads. This was added to all the OpenMP imperative, an instance shown below, schedule(dynamic).

```
172
173         omp_set_num_threads(NUM_CORES);
174         #pragma omp parallel for schedule(dynamic) reduction(+:count)
175         for (long i = 0; i < M; i++) {
176             int i_mod_aa_number = i % AA_NUMBER;
177             int i_div_aa_number = (i / AA_NUMBER) % M1;
178             int i_mod_M1 = i % M1;
179             int i_div_M1 = i / M1;
180
181             double p1 = second_div_total[i_div_aa_number];
182             double p2 = one_l_div_total[i_mod_aa_number];
183             double p3 = second_div_total[i_mod_M1];
184             double p4 = one_l_div_total[i_div_M1];
185
186             double stochastic = (p1 * p2 + p3 * p4) * total_div_2;
187
```

Figure 18: Screenshot showing added dynamic scheduling in the OpenMP imperative to load balance.

Testing and Verification

Validation of Correctness

To verify the correctness of the parallelisation, I saved the output from both sequential and parallelised version into a text file and compared them using an online ‘diff’ tool:

```
0  1 -> 0.03363286346937661986
0  2 -> 0.01200145443471613976
0  3 -> 0.00166977696066470532
0  4 -> 0.00550707828505980160
0  5 -> 0.00361159010210996027
0  6 -> 0.00168100589667431331
0  7 -> 0.01080361257130349126
0  8 -> 0.00135476701885223877
0  9 -> 0.00118778529715499138
0 10 -> 0.00069638146701031685
0 11 -> 0.00063449474679786518
0 12 -> 0.00082632894340963295
0 13 -> 0.65945546093667217757
0 14 -> 0.00134945410161818906
0 15 -> 0.00164362600846727487
0 16 -> 0.00147772992461791012
0 17 -> 0.00112369179219814542
0 18 -> 0.00081769280600686430
0 19 -> 0.00151350014139477618
0 20 -> 0.00161847379846194613
0 21 -> 0.00143961958578208739
0 22 -> 0.00155226834070001495
0 23 -> 0.00121784560182287339
0 24 -> 0.00141092164410177873
0 25 -> 0.00169338104649012109
0 26 -> 0.11951243589558020741
0 27 -> 0.10852865558671274948
0 28 -> 0.00140086414608775814
0 29 -> 0.00165524383451264284
0 30 -> 0.00281718863310481682
0 31 -> 0.01545918186175229692
0 32 -> 0.00191891061780653360
0 33 -> 0.01244230021133243234
```

```
0  1 -> 0.03363286346937661986
0  2 -> 0.01200145443471613976
0  3 -> 0.00166977696066470532
0  4 -> 0.00550707828505980160
0  5 -> 0.00361159010210996027
0  6 -> 0.00168100589667431331
0  7 -> 0.01080361257130349126
0  8 -> 0.00135476701885223877
0  9 -> 0.00118778529715499138
0 10 -> 0.00069638146701031685
0 11 -> 0.00063449474679786518
0 12 -> 0.00082632894340963295
0 13 -> 0.65945546093667217757
0 14 -> 0.00134945410161818906
0 15 -> 0.00164362600846727487
```

Figure 19: A snippet of the diff tool comparison of the two outputs.

Upon analysis, most of the values matched, however, some discrepancies were observed in certain values. Deeper inspection found that these inconsistencies were not attributable to

erroneous calculations but rather to differences in the execution order compared to the sequential version. For instance, the sequential implementation reported the correlation between 0 and 33 as (0, 33), whereas the parallelised version presented it as (30, 0). Despite the variation in ordering, both outputs conveyed identical information and were consistent with one another.

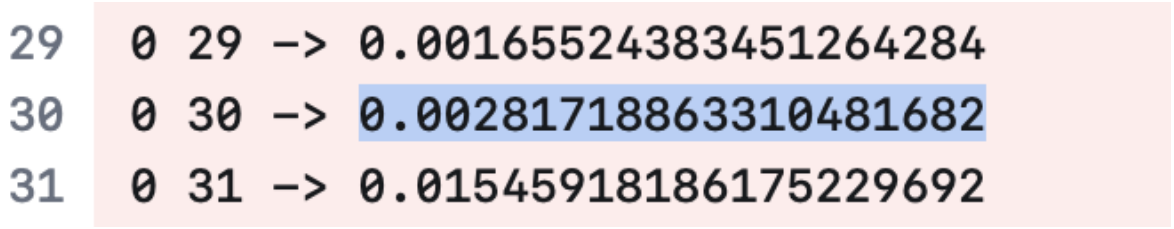


Figure 20: The correlation coefficient of 0 and 30 in the sequential version.

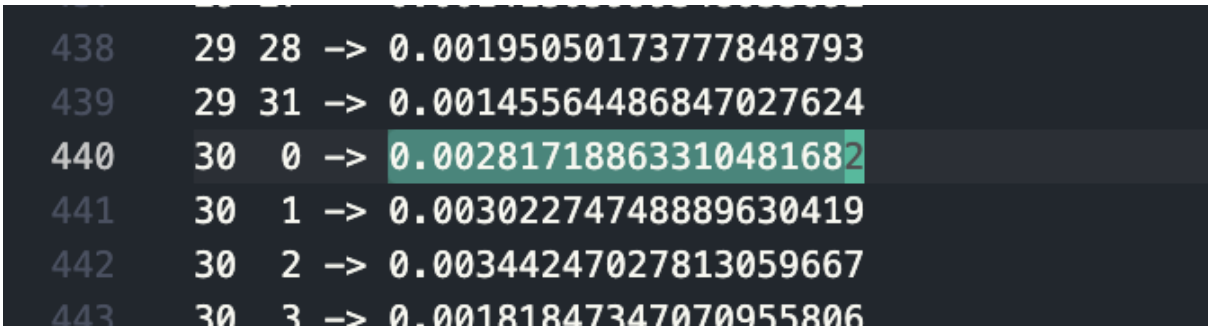


Figure 21: The correlation coefficient of 0 and 30 in the sequential version in the parallelised version shown in a different format.

Overall, the calculations were correct and matched.

Performance Evaluation

In this section I will present detailed timing results of the parallel application that is essential for quantifying performance improvements. This data enables a clear assessment of scalability, efficiency, and resource utilisation, providing a strong foundation for future development and comparative evaluation.

To evaluate the performance improvements of the implemented methods, I measured the speedup achieved by two parallelized approaches: the non-producer-consumer approach, which loads files in parallel before conducting comparisons, and the producer-consumer approach. As illustrated in the table below, the producer-consumer method demonstrated significantly faster speedups. This can be attributed to the producer-consumer strategy, where the CPU remains active during file loading for bacteria comparisons, effectively minimise idle time. Consequently, for further analysis, the producer-consumer method was selected as the preferred approach.

Number of Cores	Virtual Cores	Time (without producer and consumer)	Time (producer and consumer)
1	2	46	31
2	4	27	11
3	6	19	7
4	8	16	7
5	10	14	6
6	12	12	5
7	14	11	5
8	16	8	5

Table 1: showing the speedup results from the two parallelisation strategies, with and without consumer-producer models.

The producer-consumer approach not only achieved rapid speedups but also attained an overall performance improvement of nearly sevenfold, as illustrated in figure below. In the initial phase, with 1 to 3 cores, the observed speedup surpasses the theoretical speedup. This can be attributed to effective cache usage and minimal synchronisation overhead at lower core counts. With fewer cores, data often fits well within the cache, reducing memory access times and enhancing performance. Additionally, operations might overlap or pipeline efficiently, allowing the program to gain speedup beyond theoretical expectations.

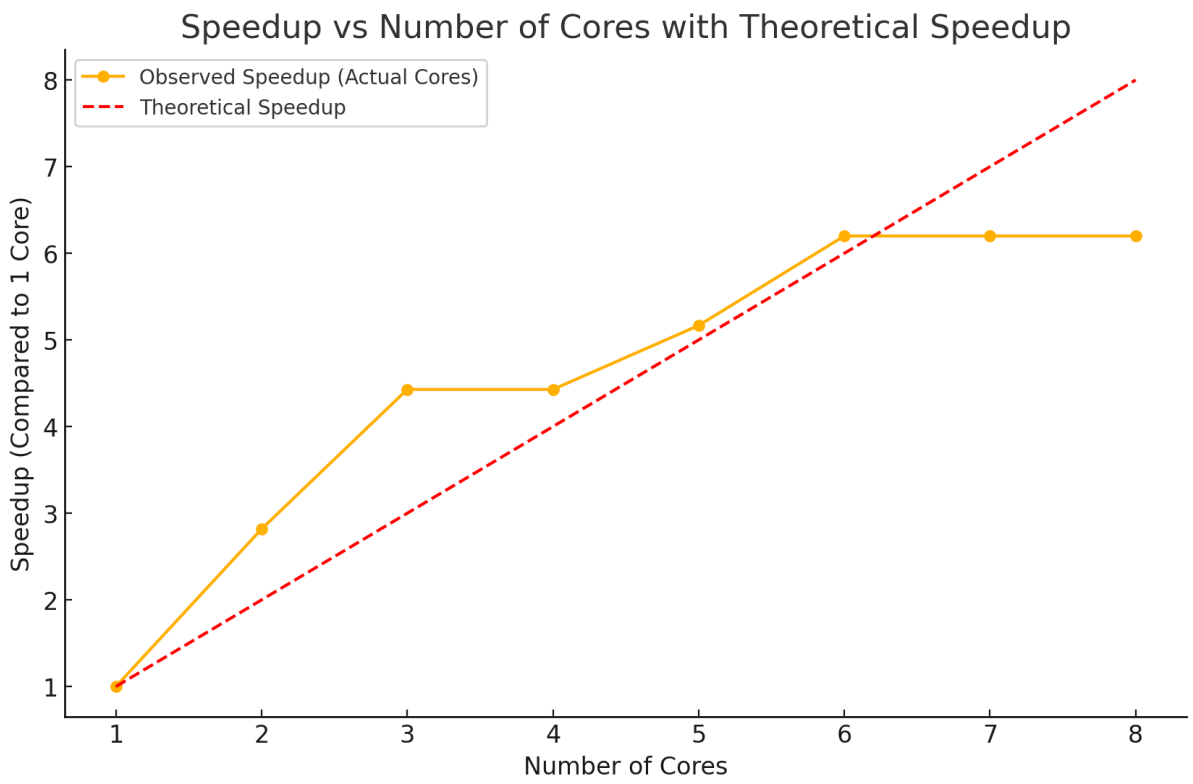


Figure 21: showing the speedup graph of the producer-consumer parallelisation strategy and theoretical grah.

However, as the number of cores increases from 4 to 8, the observed speedup plateaued and fell below the theoretical line. This decline is likely due to the increased synchronisation and communication overhead that comes with more cores, as managing and coordinating tasks becomes more complex. Memory bandwidth limitations also contribute, with more cores contending for access to shared memory, which slows down performance. According to Amdahl's Law, any remaining sequential portions of the code begin to dominate as the parallel part reaches its optimisation limits, capping further speedup.

Throughout the execution of this parallelised method, CPU utilisation remained nearly 100% across all 16 cores, indicating optimal performance with minimal idle time.

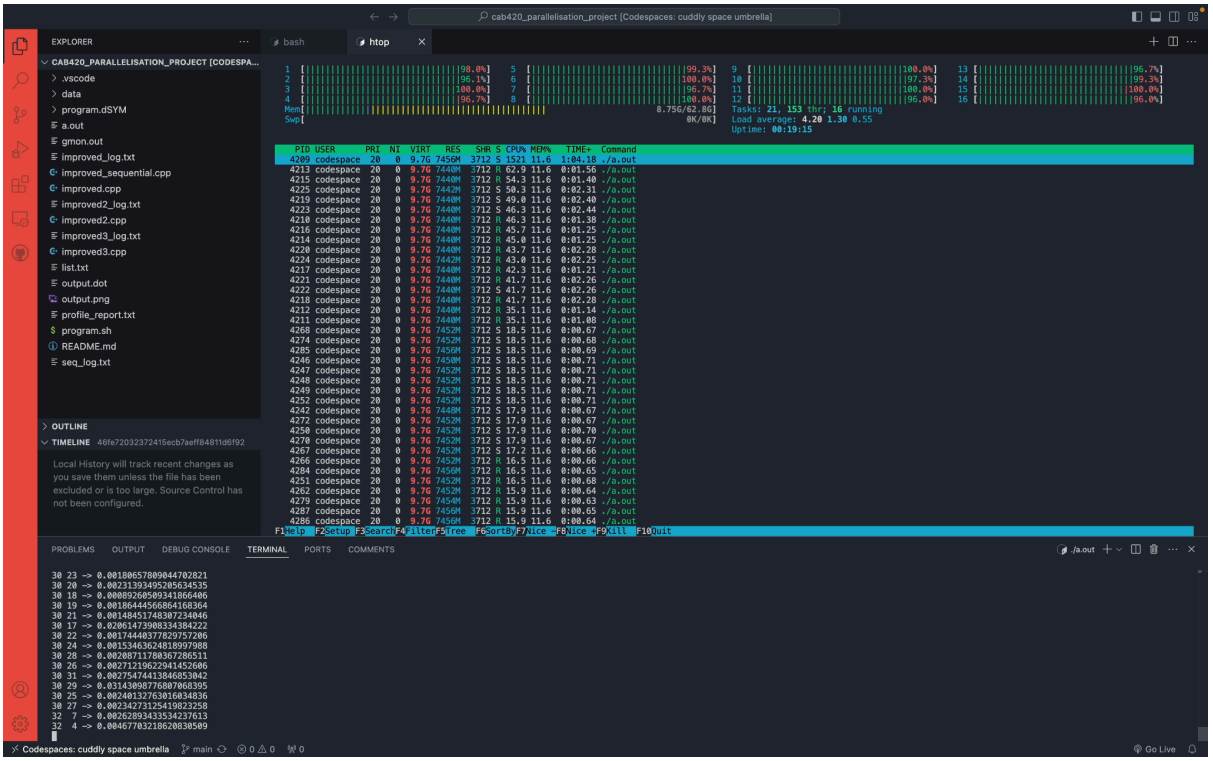


Figure 22: showing a snapshot of the CPU utlisation for the parallelised model.

For a detailed view of CPU utilisation, please refer to the video link below. Due to limited access to Visual Studio (as I was using Ubuntu) and cross compatibility issues with code, I could not transfer my code to a windows format to take advantage of Visual Studio's Performance Analyser tools. The best method in Linux that I found was recording the CPU utlisation.

https://drive.google.com/file/d/1SWi4-hKCPq_Nq7NfJDSso7YbnPchS85C/view?usp=sharing

I conducted profiling as well, but the results were indecipherable, and I was unable to identify an alternative tool or method for clearer analysis.

Figure 23: showing performance analysis of the parallelised version with gprof.

Tools and Technologies Used

The following technology stack was used:

Technology Type	Used
Programming Language	C++
Profiling Tools	Gprof, htop
Main libraries	Thread, mutex, iomanip
OS	Ubuntu 20.04.6 LTS
Compiler	GCC and G++
Compiler flags	g++ -std=c++17 -fopenmp -pthread

Table 2: showing a table of the main tools and technologies used.

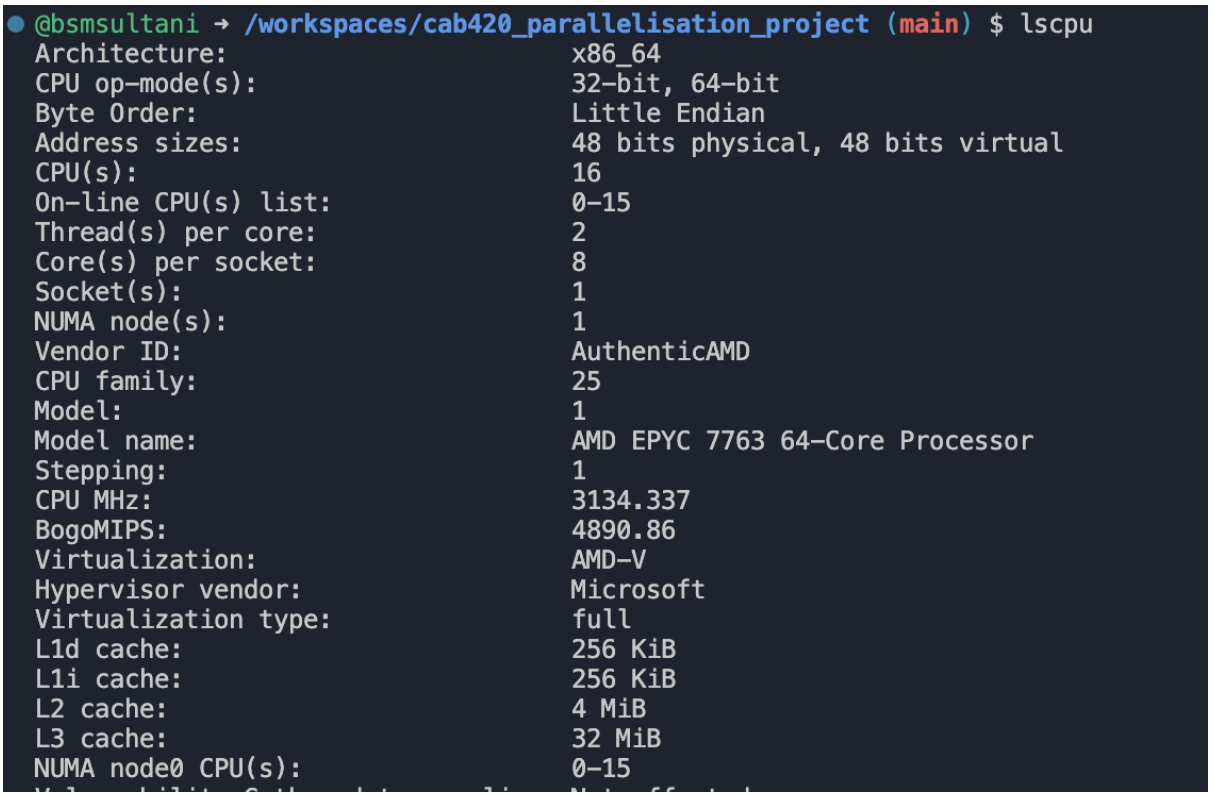


Figure 24: showing hardware including information about the CPU cores and caches.

Reflection

Although this report is presented sequentially, the project itself was highly iterative, requiring me to overcome numerous barriers and experiment with various implementations.

For instance, an initial challenge involved restructuring the code's primary loops to make them compatible with parallel execution by eliminating loop dependencies. This required an in-depth analysis of the program, followed by extensive testing and debugging to ensure that the restructured code produced accurate and consistent output.

After this initial parallelisation, I implemented the more efficient version of the program with producer-consumer concept. In this method I faced a lot of challenges related to synchronisation and locks. As this concept was initially unfamiliar, I consulted with my tutor to assess its feasibility. After these discussions, I proceeded with the implementation, engaging in continuous debugging and testing over several weeks. Ultimately, this solution proved to be effective in improving the program's efficiency. During this period, I gained valuable insights into mutexes, synchronisation, and locks, which have proven invaluable to my understanding of concurrent programming.

I also have learned a number of other tools in this project. I learned how to use ‘gprof’ for profiling and became familiar with Visual Studio Profiling as well. Overall the experience was insightful and joyful and I’ve gained important skills to tackle future problems.

Conclusions

The analysis of the CVTree application highlighted significant performance bottlenecks in its single-core implementation. Critical functions such as the Bacteria constructor and the CompareBacteria function relied heavily on sequential processing and extensive looping structures, which became increasingly inefficient as the dataset grew. The character-by-character file reading and the intensive computational loops contributed to substantial delays, hindering the application's ability to handle large-scale genomic analyses effectively.

Addressing these inefficiencies was essential for enhancing computational efficiency and reducing runtime. By identifying and understanding the specific areas where bottlenecks occurred, strategies such as parallelisation and optimising memory management were implemented. These improvements not only decreased processing time but also made the application more suitable for handling vast biological datasets, thereby facilitating more timely and accurate insights into bacterial evolution and genetic relationships.

Appendices

Appendix 1: Screenshot showing the output of *gprof* results.

- **% Time:** Total percentage of execution time spent in the function.
- **Cumulative Seconds:** Total seconds spent in the function and all called functions.
- **Self Seconds:** Seconds spent exclusively in this function.
- **Calls:** Number of times the function was invoked.
- **s/call:** Average seconds per call to the function.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 58.68     21.61    21.61       41     0.53     0.53  Bacteria::Bacteria(char const*)
 40.85     36.65    15.04      820     0.02     0.02  CompareBacteria(Bacteria*, Bacteria*)
  0.68     36.90     0.25  1942569     0.00     0.00  Bacteria::cont_buffer(char)
  0.00     36.90     0.00     5642     0.00     0.00  Bacteria::init_buffer(char*)
  0.00     36.90     0.00       41     0.00     0.00  Bacteria::InitVectors()
  0.00     36.90     0.00       41     0.00     0.00  Bacteria::~Bacteria()
  0.00     36.90     0.00        1     0.00     0.00  ReadInputFile(char const*)
  0.00     36.90     0.00        1     0.00    36.90  CompareAllBacteria()
  0.00     36.90     0.00        1     0.00     0.00  Init()
```