CAB401 High Performance and Parallel Computing

Name: Guo Liang Ken Ho

Student ID: N11730706

Title: Bioinformatics – Genome similarity using Frequency vectors (C++)

Contents

Introduction 3
Call Graph 3
Analysis of Potential Parallelism 4
CompareAllBacteria() 4
CompareBacteria() 5
Parallel Language and Framework Used 5
OpenMP 5
Timing and Profiling Results 6
Results for 3-mers 6
Results for 4-mers 6
Results for 5-mers 6
Results for 6-mers 7
SpeedUp Curve 8
Sequential Results VS Parallel Results 8
Description of Compilers, Software, Tools, and Techniques Used 9
Overcoming Performance Barriers 9
Reflection 9

Introduction

This report focuses on the optimization and parallelization of crucial application within the realm of bioinformatics, specifically Bioinformatics – Genome Similarity using Frequency Vectors. In this report, we will take an existing sequential C++ code that primarily deals with DNA sequence analysis of various bacteria.

The current code iterates through a list of bacteria genomes, comparing each genome's DNA sequence to that of others. Then, it computes correlation values using a stochastic model. However, we have observed that the core computation involves iterating over a large number of vectors with repetitive calculations for each pair of bacteria. Additionally, the main body of the code invokes a function called CompareAllBacteria(), which can be performed independently since comparisons between each pair of bacteria are not affected by others. These characteristics of the code are favourable for us to leverage parallel computing to improve its efficiency and performance.

Call Graph

A simplified call graph of the application is shown below (Figure 1).

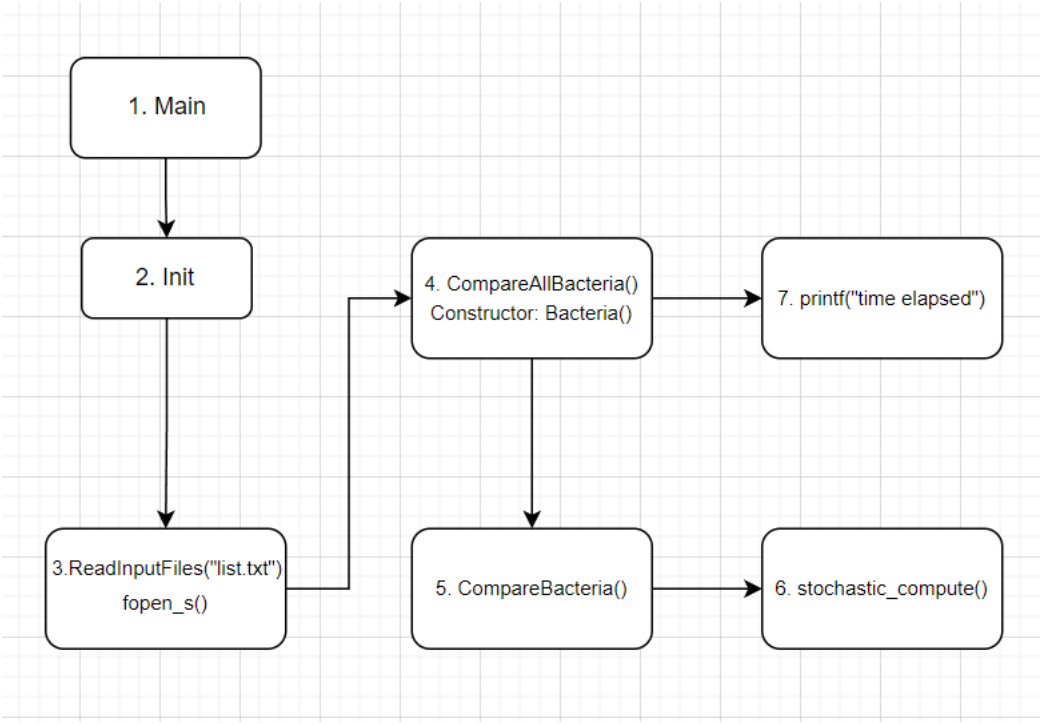


Figure 1: Call graph of Bioinformatics – Genome similarity using Frequency vectors (C++)

A brief explanation of the various components in the call graph.

1. Main

- Entry point of the program.
- Initializes a timer (t1) to measure execution time.
- Calls Init() to initialize some constants.

2. Init

- Initialize global constants M, M1, M2 using predefined values of LEN and AA_NUMBER.

3. ReadInputFiles()

- Reads a file named “list.txt” to obtain the names of the bacteria.
- Calls CompareAllBacteria().

4. CompareAllBacteria()

- Iterates over all pairs of bacteria files.
- For each pair of bacteria, it creates Bacteria objects.
- Calls CompareBacteria().
- Prints the correlation value of that pair of bacteria.

5. CompareBacteria()

- Calls stochastic_compute().
- Calculates the correlation between each pair of bacteria.

6. Stochastic_compute()

- Computes the stochastic value.

7. Printf()

- Prints the time elapsed to compute the correlation values of each pair in the file.

Analysis of Potential Parallelism

Upon closer analysis, I’ve identified functions in the code that can potentially be parallelize.

CompareAllBacteria()

CompareAllBacteria() performs a series of independent comparisons between each pair of bacteria. Each comparison is independent and does not depend on the results of other comparisons, making it ideal for parallel execution.

The function also contains nested loops, with the outer loop iterating over bacteria 'i' and the inner loop iterating over bacteria 'j'. These iterations can be distributed among multiple threads to perform the comparisons concurrently.

Parallelizing this function can potentially improve the program's performance and efficiency, especially on multi-core systems. As the number of pairs to compare is significant, the benefits of parallelization become more apparent. Leveraging parallelization allows pairwise comparisons to be completed much faster, making it advantageous when dealing with a large number of bacteria and computationally intensive comparisons (see Figure 2).

Sequential - CompareAllBacteria()

```
void CompareAllBacteria()
{
    for (int i = 0; i < number_bacteria - 1; i++)
    {
        Bacteria* b1 = new Bacteria(bacteria_name[i]);

        for (int j = i + 1; j < number_bacteria; j++)
        {
            Bacteria* b2 = new Bacteria(bacteria_name[j]);
            double correlation = CompareBacteria(b1, b2);
            printf("%03d %03d -> %.10lf\n", i, j, correlation);
            delete b2;
        }
        delete b1;
    }
}
```

Parallel – CompareAllBacteria()

```
void CompareAllBacteriaParallel()
{
#pragma omp parallel
    {
        int num_threads = omp_get_num_threads();
        int thread_id = omp_get_thread_num();
        int chunk_size = number_bacteria / num_threads;
        int start = thread_id * chunk_size;
        int end = (thread_id == num_threads - 1) ? number_bacteria : start + chunk_size;

        for (int i = start; i < end; i++)
        {
            for (int j = i + 1; j < number_bacteria; j++)
            {
                CompareBacteriaParallel(i, j);
            }
        }
    }
}
```

Figure 2: Comparison between sequential and parallel versions of CompareAllBacteria().

Several variables have been initialized to prevent race conditions, ensuring that each thread operates on a distinct portion of the bacteria array.

Variables	Purpose
int num_threads = omp.get_num_threads();	Returns total number of threads in the parallel section.
int thread_id = omp_get_thread_num();	Returns the ID of the current thread.
int chunk_size = number_bacteria / num_threads;	chunk_size represents an approximate equal distribution of work among threads.
int start = thread_id * chunk_size;	start refers to the index which a thread should begin its work.
int end = (thread_id == num_threads – 1) ? number_bacteria : start + chunk_size;	end refers to the index which a thread should stop its work.

Since each thread has its unique ‘start’ and ‘end’ indices, there are no overlaps or interference with each other, thus preventing race conditions from arising.

CompareBacteria()

Race conditions cannot be neglected, especially when multiple threads attempt to write simultaneously, such as when printing the correlation of different pairs of bacteria. To prevent race conditions, precautionary measures must be implemented.

Original

```
void CompareAllBacteria()
{
    for (int i = 0; i < number_bacteria - 1; i++)
    {
        Bacteria* b1 = new Bacteria(bacteria_name[i]);

        for (int j = i + 1; j < number_bacteria; j++)
        {
            Bacteria* b2 = new Bacteria(bacteria_name[j]);
            double correlation = CompareBacteria(b1, b2);
            printf("%03d %03d -> %.10lf\n", i, j, correlation);
            delete b2;
        }
        delete b1;
    }
}
```

Modified

```
void CompareBacteriaParallel(int i, int j)
{
    Bacteria* b1 = new Bacteria(bacteria_name[i]);
    Bacteria* b2 = new Bacteria(bacteria_name[j]);

    double correlation = CompareBacteria(b1, b2);

#pragma omp critical
    {
        printf("%03d %03d -> %.10lf\n", i, j, correlation);
    }

    delete b1;
    delete b2;
}
```

Figure 3: Handling race condition in CompareAllBacteria().

Parallel Language and Framework Used

OpenMP

A data-level parallelism approach was adopted to distribute the computational workload among multiple processors. The OpenMP parallel programming framework was utilized to efficiently divide the work among threads.

Inside the ‘CompareAllBacteriaParallel()’ function, OpenMP”s ‘#pragma omp parallel’ directive was used to parallelize the nested loops responsible for comparing the bacteria. This allowed computation to be mapped to multiple threads, with each thread assigned to process distinct pair of bacteria.

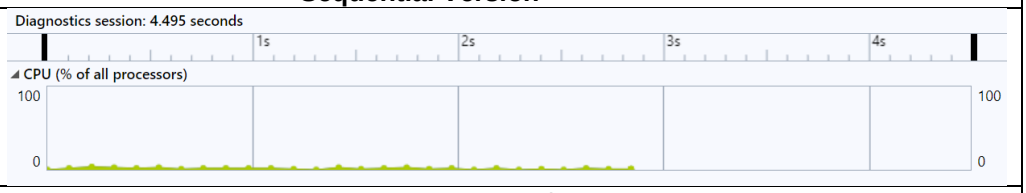
To prevent data inconsistency, OpenMP constructs such as critical section were employed. These critical sections ensured that only one thread at any point in time could print the correlation results. This prevented race conditions when multiple threads attempted to write the output simultaneously.

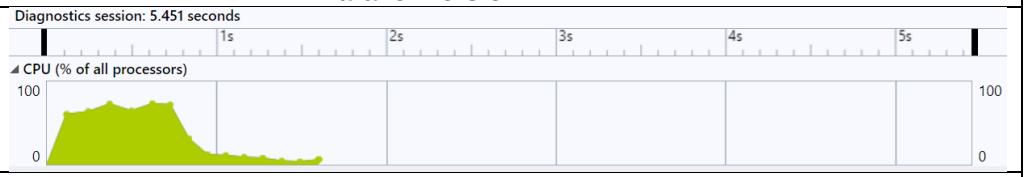
To handle synchronization, critical sections were used, guaranteeing exclusive access to the standard output. This approach ensured that the parallelized code executed correctly and efficiently.

Timing and Profiling Results

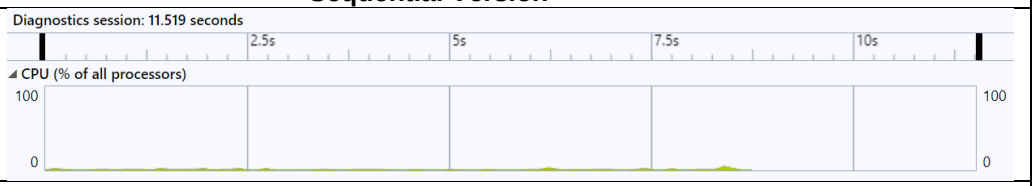
Below are the direct comparison of the CPU utilization and time taken for the code to execute while varying the value of k in k-mers (Figure 4).

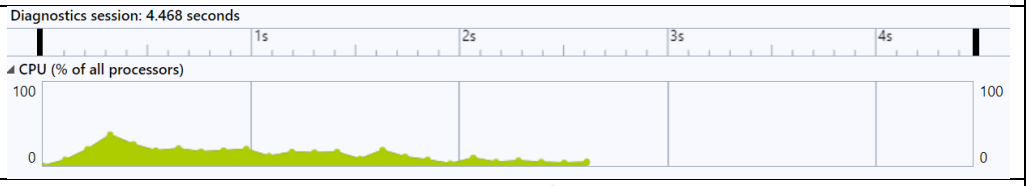
Results for 3-mers

Sequential Version	
CPU Utilization	
Time Taken	2 seconds

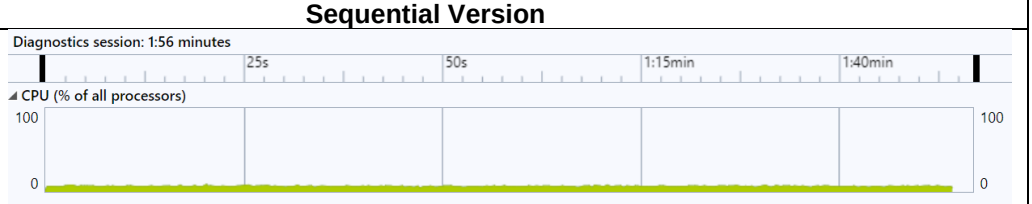
Parallel Version	
CPU Utilization	
Time Taken	2 seconds

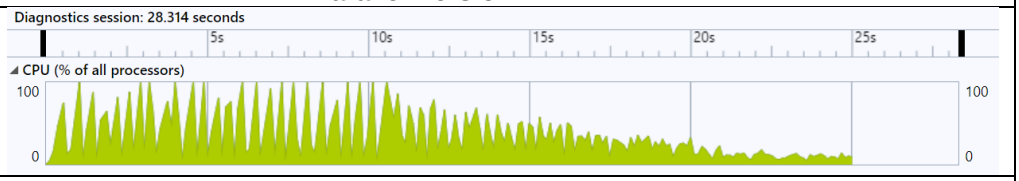
Results for 4-mers

Sequential Version	
CPU Utilization	
Time Taken	9 seconds

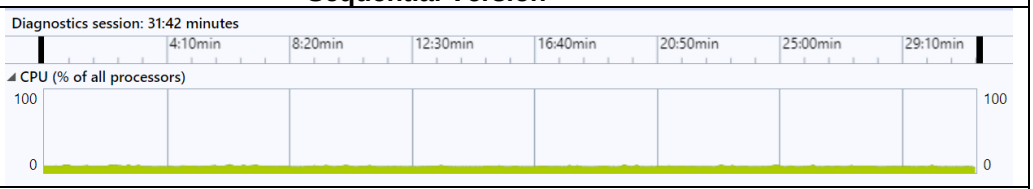
Parallel Version	
CPU Utilization	
Time Taken	3 seconds

Results for 5-mers

Sequential Version	
CPU Utilization	
Time Taken	114 seconds

Parallel Version	
CPU Utilization	
Time Taken	25 seconds

Results for 6-mers

Sequential Version	
CPU Utilization	
Time Taken	1900 seconds

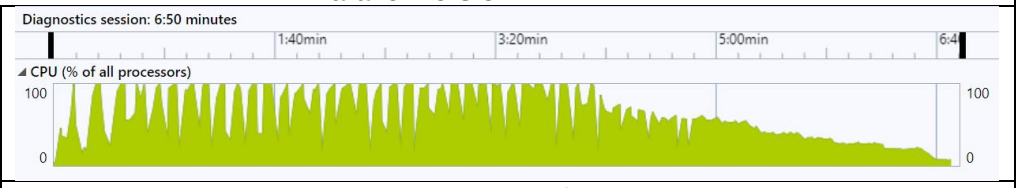
Parallel Version	
CPU Utilization	
Time Taken	407 seconds

Figure 4: Comparison of results between the sequential and parallel versions.

The impact of code parallelization on CPU utilization and execution time reduction during bacteria analysis from 3-mers to 6-mers is evident. Prior to parallelization, the code ran sequentially, resulting in the underutilization of the CPU's multiple cores. However, the implementation of OpenMP parallelization techniques efficiently distributed the computational workload across 16 threads, fully harnessing the available CPU cores.

Consequently, CPU utilization notable increased as each thread executed concurrently to analyse different pairs of bacteria. This simultaneous processing of data led to a significant reduction in execution time. Parallelization noy only optimized CPU utilization but also substantially decreased the time required to complete the analysis of k-mers.

SpeedUp Curve

A SpeedUp Curve was plotted by varying the number of threads from 2 to 16. Below is the graph obtained when SpeedUp is plotted against the number of threads (Figure 5).

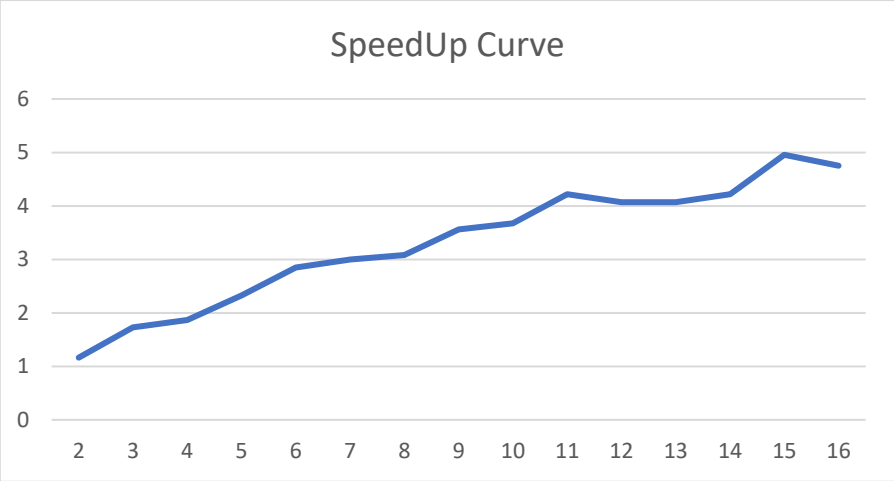


Figure 5: SpeedUp Curve

From the SpeedUp Curve, we observed scalable parallelism as the number of threads increased. However, towards the end, the graph appeared to reach a plateau. This suggests that we might be approaching a point of constant parallelism, where further increases in the number of threads will no longer significantly increase the SpeedUp value.

Sequential Results VS Parallel Results

To ensure the validity of the obtained results, I conducted a comparison between the results generated from 3-mers to 6-mers using the modified code. The outcomes of multiple runs were recorded in an Excel spreadsheet. However, due to the unordered nature of parallel computation results, it was necessary to sort the parallel run results in ascending order before applying Excel formulas to compare pairs and their corresponding values (Figure 6).

Sequential Version			Parallel Verison			Are the results the same?
0	1 ->	0.033633	0	1 ->	0.033633	TRUE
0	2 ->	0.012001	0	2 ->	0.012001	TRUE
0	3 ->	0.00167	0	3 ->	0.00167	TRUE
0	4 ->	0.005507	0	4 ->	0.005507	TRUE
0	5 ->	0.003612	0	5 ->	0.003612	TRUE
0	6 ->	0.001681	0	6 ->	0.001681	TRUE
0	7 ->	0.010804	0	7 ->	0.010804	TRUE
0	8 ->	0.001355	0	8 ->	0.001355	TRUE
0	9 ->	0.001188	0	9 ->	0.001188	TRUE
0	10 ->	0.000696	0	10 ->	0.000696	TRUE
0	11 ->	0.000634	0	11 ->	0.000634	TRUE
0	12 ->	0.000826	0	12 ->	0.000826	TRUE
0	13 ->	0.659455	0	13 ->	0.659455	TRUE
0	14 ->	0.001349	0	14 ->	0.001349	TRUE
0	15 ->	0.001644	0	15 ->	0.001644	TRUE
0	16 ->	0.001478	0	16 ->	0.001478	TRUE
0	17 ->	0.001124	0	17 ->	0.001124	TRUE
0	18 ->	0.000818	0	18 ->	0.000818	TRUE
0	19 ->	0.001514	0	19 ->	0.001514	TRUE
0	20 ->	0.001618	0	20 ->	0.001618	TRUE
0	21 ->	0.00144	0	21 ->	0.00144	TRUE
0	22 ->	0.001552	0	22 ->	0.001552	TRUE
0	23 ->	0.001218	0	23 ->	0.001218	TRUE
0	24 ->	0.001411	0	24 ->	0.001411	TRUE
0	25 ->	0.001693	0	25 ->	0.001693	TRUE
0	26 ->	0.119512	0	26 ->	0.119512	TRUE
0	27 ->	0.108529	0	27 ->	0.108529	TRUE
0	28 ->	0.001401	0	28 ->	0.001401	TRUE
0	29 ->	0.001655	0	29 ->	0.001655	TRUE
0	30 ->	0.002817	0	30 ->	0.002817	TRUE

Figure 6: Comparison of results between sequential and parallel versions of 6-mers.

Description of Compilers, Software, Tools, and Techniques Used

In parallelizing the application, I utilized several key components and techniques. The primarily programming language used for the project was C++, together with OpenMP (Open Multi-Processing) framework as an imperative tool for parallelization. OpenMP provided the necessary constructs and directives to parallelize the code effectively.

Below is a summary of tools and software used in this parallelization project:

Compiler: Microsoft C++ Compiler.

Software: Microsoft Visual Studio Community 2022 17.6.5.

Tool: Open Multi-Processing (OpenMP).

Overcoming Performance Barriers

To overcome performance barriers in the code, several steps were taken:

- Load Imbalance: Distributed the workload among multiple threads. Each thread handles a portion of the pairwise comparisons, leading to better load distribution.
- Granularity: Divided the work into chunks based on the number of threads. This provides a balanced distribution of work and avoids overly fine or coarse granularity.
- Data Dependencies: Used critical sections to ensure safe access to shared resources.

Reflection

The parallelization of this application was an intriguing experience that highlighted the advantages and complexities of parallel computing. Notably, there was a significant reduction in execution time when transitioning to the parallelized version, demonstrating the importance of harnessing the full computational power of multi-core processors through workload distribution among threads.

However, the process also revealed challenges. Load balancing was a concern, requiring adjustments to workload distribution and optimized parallel regions for improved CPU utilization. Additionally, managing concurrent access to shared resources demanded careful synchronization.

To close, this parallelization project emphasized the importance of code optimization and the role of human intervention in improving performance, which machines are currently limited in achieving.