Parallelization of CV Sequence Analysis

Jacob Marks

April 15, 2018

Contents

Abstract

The following report discusses the parallelization of an alignment-free composition vector sequence analysis program. As will be covered, it was found that 89.4% of the program was able to be parallelized, and after exposing shared memory loop parallelism, speedups of up to 2.5 times were observed when compared to the sequential program. The project was considered a success, but not without it's flaws and potential further optimizations.

Chapter 1

The Program

1.1 Alignment-Free Sequence Analysis

In bioinformatics, sequence analysis is the process of subjecting genomatic data (DNA, RNA, etc.) to a range of analytic methods to better understand their composition or properties. Sequence alignment is one such method that compares the 'genetic letters' of two or more genome sequences and groups them into protein/gene families where matches are found. A portrayal of pair-wise sequence alignment can be seen below in Figure 1.1, where two genome sequences are matched through their similar composition.

AJWIDJFRKLENNSIELKD
WJEEOQPZMSNLNSIQDXZ

Figure 1.1: Portrayal of pair-wise sequence alignment.

However, these sequence alignment algorithms can become computationally expensive when subjected to larger and larger datasets, rendering them inefficient and limiting their use [5]. Alternative 'alignment-free' algorithms have been developed to not only improve the speed at which genome sequences can be analysed, but to also extend their methods of comparison beyond the examination of homologous segments [6].

The program to be parallelized implements one such alignment-free sequence analysis method, based on k-mer/word frequency, known as composition vector (CV). In this method, the frequency of appearance of all possible fixed-length k-mers are calculated and stored in a vector. Given the frequency vectors of two sequences, a distance function is applied to calculate a normalised measure of dissimilarity between them. The program compares all unique combinations of sequences within a given set and forms a dissimilarity matrix, which can be particularly useful in the construction of phylogenetic trees using clustering algorithms. [1]

1.2 Program Call Hierarchy

Shown below in Figure 1.2 is a diagram representing the structure and execution of the program.

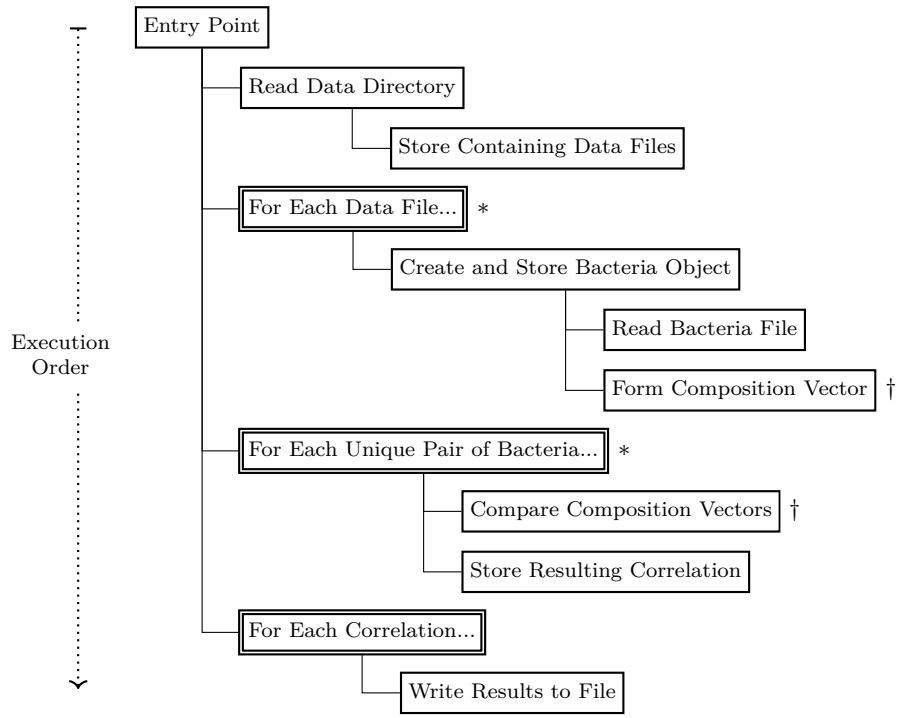


Figure 1.2: Program Call Hierarchy Overview
∗ Potential Parallelism
† Bottleneck

As can be seen in the diagram, most primary top-level looping operations can be targeted as points of potential parallelism, this idea is further covered in Chapter 3.

1.3 Bottlenecks

There are two rather large bottlenecks within the program, where the bulk of computation takes place. As noted in Figure 1.2 above, these two points include the forming of composition vectors, and their comparison.

Through the use of profiling software, explained in Section 4.3, it was found that the program spent 68.5 and 31.4 percent of its total execution time both forming and comparing vectors, respectively; Undoubtedly exposing these points as the programs biggest bottlenecks.

1.4 Data Dependencies

When it comes to the potential parallelism of the sequential program, only a few data dependencies exist that need be examined. Referring to the above diagram in Figure 1.2, there are basic flow dependencies between each top-level task, including the reading of Bacteria data files, formation of composition vectors, comparison of those vectors, and outputting of results. Each task relies only on the output of the previous task, with all sub-tasks utilising independent data with no need to communicate outside their bounds.

A visualization can be seen in the figure below, in which each task of the program is flow dependant on the previous.

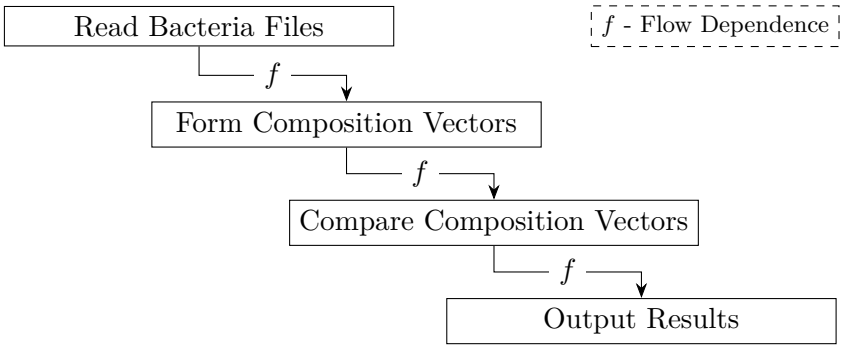


Figure 1.3: Program Flow Dependence

Chapter 2

Fixes and Optimizations

Before considering the parallelization of the original program, a few edits were made to improve the sequential program; As discussed below.

2.1 Data File Referencing

The program was intended to be run with a single argument giving reference to a text file, hereby referred to as a 'data list', containing the number of sequences on the first line, and then listing the sequence names on subsequent lines. These names would then be appended with a hardcoded filetype (in this case `.faa`), to then be used to open and read the files.

This method, which can be seen in Appendix I, required the data files be in the root directory of the executing program (unless prepended onto the filenames in the data list), as well as the construction of a data list itself, which would be unreasonable given any decent amount of sequence files.

This method was deemed harmful to the usability of the program as changing the data or its location would require tedious edits be made also to the data list. An improved method of data referencing was implemented that bypasses the use of a data list and reads a given directory directly. All files within this directory, given to the application as an argument, have their filenames (including extension) stored in the program, and can be easily read after prepending the given directory. This process also counts the total number of files encountered, to be used for various memory allocations.

Some small adjustments had to be made throughout the program to achieve this, including the usage of a `string` vector over a `char` array for storing the filenames. One small side effect also arose, that being there is no guarantee in the order that the filenames are read in the given directory, and while not completely necessary, is negated by sorting the string vector after the directory has been read using the inbuilt `C++ sort` function.

This refined method, which can be seen in Appendix J, not only eliminates the need for a data list, but also extends the freedom of data locale.

2.2 Carriage Returns

The data that was provided with the original program would produce 'segmentation faults' at runtime, with no real information why. The cause was eventually tracked down to the `encode` function, where unexpected non-alphabetical characters were being passed. It was then found that these characters were carriage returns (`\r`) contained within the text, most likely due to the formation of the data on a Windows operating system, which handles newlines differently compared to Linux-based operating systems.

This differentiation proved troublesome, and the problem was solved by simply ignoring the characters when reading the data, as can be seen in the snippet below.

```
...
else if (ch == '\r') {
    // Skip carriage return.
}
...
```

The complete section of code for reading sequence data characters can be found in Appendix K.

2.3 Result Outputting

A small section of code was added to write the results of the program to a text file in a comma-separated format. The contents of the file are formatted as can be seen in the table below.

Index A	Bacteria A	Index B	Bacteria B	Correlation
0	data/AIHV_1.faa	1	data/AMEV.faa	0.00120256
0	data/AIHV_1.faa	2	data/APMiV.faa	0.00290071
0	data/AIHV_1.faa	3	data/ASFV.faa	0.00149284
...	...	...	...	...

Table 2.1: Output CSV format.

This writing of results is not only important for the examination and analysis of Bacteria correlations, but also useful in ensuring results are equivalent between the original, improved and parallel versions of the application. The complete section of code for the task can be found in Appendix L.

2.4 Vectors

C++ vectors have been implemented to replace some critical arrays within the program. These arrays, used for the construction of an individual composition vector for a given Bacteria data file, were found (through profiling) to be using considerable amounts of memory. They were replaced with vectors in an attempt to alleviate this memory strain and favour their dynamic allocation. This difference in allocation can be seen below in Figure 2.1.

(a) Original Array Usage

```
long* vector;
long* second;
...
vector = new long [M];
second = new long [M1];
memset(vector, 0, M * sizeof(long));
memset(second, 0, M1 * sizeof(long));
```

(b) Improved Vector Usage

```
vector<long> vectorA;
vector<long> vectorB;
...
vectorA = vector<long>(M, 0.0);
vectorB = vector<long>(M1, 0.0);
```

Figure 2.1: Array vs Vector Allocation

Vectors support index-based access, similar to that of arrays, meaning no other code had to be altered to utilize them (other than some variable renaming). To de-allocate the vector memory, each vector was "swapped" with an empty one, as unfortunately they do not yet gracefully support the task. This difference in de-allocation can be seen below in Figure 2.2.

(a) Original Array Usage

```
delete vector;
delete second;
```

(b) Improved Vector Usage

```
vector<long>().swap(vectorA);
vector<long>().swap(vectorB);
```

Figure 2.2: Array vs Vector De-Allocation

This usage of vectors over arrays resulted in a 36% execution time speedup, as well as a 37% increase in average DRAM bandwidth for the sequential program. These results can be seen graphed in Appendices F and G respectively.

Chapter 3

Potential Parallelism

Given the sequential program's biggest bottlenecks, the forming and comparison of composition vectors (previously discussed in Section 1.3) consisting a total of 99.9% of the execution time, these tasks are the biggest targets when it comes to the implementation of parallelism.

In evaluating the possibility of such parallelism, the data dependencies affecting these tasks and the structural nature of their code within the program must be considered. As previously covered in Section 1.4, all top-level tasks, including the two targeted for parallelism, are simply flow dependant on their previous. This, in addition to the loop-based nature of both tasks, means that they would be more than, if not 'embarrassingly', parallelizable; with the only requirement being that the task to compare all of the composition vectors occurs after the task in which they are formed, as to adhere to their dependencies.

Using Amdahl's Law, a theoretical speedup can be calculated given the percentage of the sequential program that could be parallelized. This percentage, however, is not 99.9%, as is the time consumed by both targeted tasks. This is due to the involvement of on-disc data file reading, which is present within the task for forming composition vectors, as can be seen in Figure 1.2.

This task has been kept sequential as to prevent any I/O bottlenecking that may occur as multiple threads attempt to read from the same location. Through profiling, this inherently sequential section within the task to be parallelized was found to consume 10.5% of the execution time, meaning the total portion of the sequential program that could be parallelized is 89.4%.

With this metric, the following speedup formula can be constructed. Where s is the speedup of the parallel section.

$$S_{latency}(s) = \frac{1}{(1 - 0.894) + 0.894/s}$$

The speedup value s is equivalent to the number of processors given perfect parallel efficiency. Using this formula, the following theoretical maximum speedup graph can be formed representing the parallel program's performance given increasing processor counts, and represents strong potential speedup given the successful parallelization of the targeted tasks.

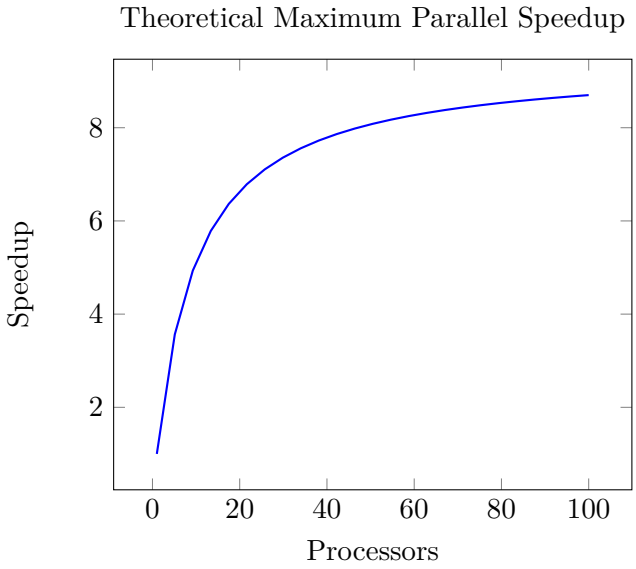


Figure 3.1: Theoretical Maximum Parallel Speedup

Given this proposed parallelism, a new program flow diagram can be constructed, as can be seen in Appendix A, representing the execution of the parallel program.

Chapter 4

Programs and Tools

4.1 OpenMP

The OpenMP API has been used to implement and achieve the parallelism of the sequential program. Managed by the *OpenMP Architecture Review Board* (OpenMP ARB), it provides a range of easy-to-use compiler directives and runtime library routines to direct multi-threaded, shared memory parallelism using threads [2].

Two directives have been used for the parallelization of this program, as discussed further in Chapter 8, primarily exposing loop parallelism.

4.2 Compiler

To compile the program, the commonly used GNU C++ compiler (`g++`) was used, given its widespread and ongoing support. Compile optimizations have been used as a convenient way to initially reduce the execution time of the program. Given the various optimization levels the GNU compiler provides, optimization level 2 (O2) has been found to produce a 60% speedup when compared to the execution of the sequential program with no optimization. These results, as well as the results for all optimization levels, can be seen graphed in Appendix H.

4.3 Profiler

Intel's VTune Amplifier was used to profile the application. Integrated within Visual Studio and also available as a standalone application (used in this case), it provides a range of high-performance profiling tools for both sequential and parallel applications, supporting OpenMP, for a wide range of programming languages [3].

Utilized aspects include high resolution execution timing, hotspot determinism, memory access benchmarking and CPU utilization monitoring.

Chapter 5

Abstractions

5.1 Fork-Join Model

OpenMP uses a fork-join model to achieve parallelism, in which the main or "master" thread delegates, at specific locations, an amount of work to be completed by another thread.

A visualization of this process can be seen in the figure below, in which the master thread executes code throughout the application, creating extra threads for parallel sections where needed.

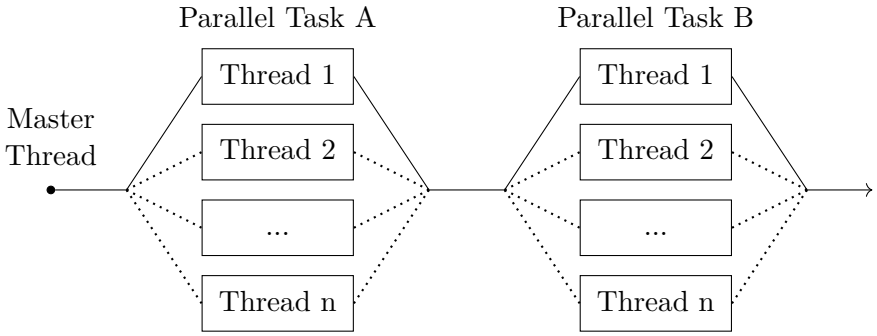


Figure 5.1: Fork-Join Model Visualization

This model can be easily seen represented in the parallel program flow diagram in Appendix A, whereby the program forks to complete the task of creating composition vectors, and again when comparing the vectors.

5.2 Loop Parallelism

A common type of parallelism when transforming sequential programs, loop parallelism is the process of taking a total amount of work to be done within a loop, and splitting the work across multiple threads [4]. The process

is largely supported by OpenMP and can be implemented using compiler directives.

Being that the sequential program is primarily concerned with loop-based work, loop parallelism is the primary method of parallelism that has been utilized, splitting the recursive work (via the fork-join model) of forming and comparing composition vectors.

5.3 Shared Memory

A shared memory model has been utilised, as supported by OpenMP, in which each separate thread is allowed unrestricted access to data outside of its local space. Given the nature and lightly-bound data dependencies of the sequential program, as covered in Section 1.4, this model can be effectively utilised with little to no need for explicit synchronization.

This problematic-free shared memory model is supported through the pre-allocation of vector space where each thread will store a unique result, which also alleviates the need for each thread to perform its computation in any particular order. A visualization can be seen in the figure below, and applies to both the storing of created composition vectors, and comparison results.

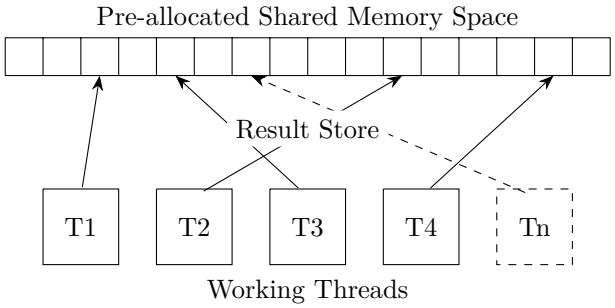


Figure 5.2: Shared Memory Result Storing Within the Program

Chapter 6

Timing and Profiling Results

The implementation of parallelism into the sequential program initially resulted in a 235% speedup. This was utilising OpenMP's automatic maximum thread determination in which the number of threads created will match the number of detected CPU cores. In this case, this number was 8, comprising of 4 physical cores and 4 virtual cores through Hyperthreading.

To explore this speedup in greater depth however, varying thread counts were profiled (using OpenMP's thread limiting) and their speedup documented and contrasted with the previously theorized maximum, as graphed in the figure below.

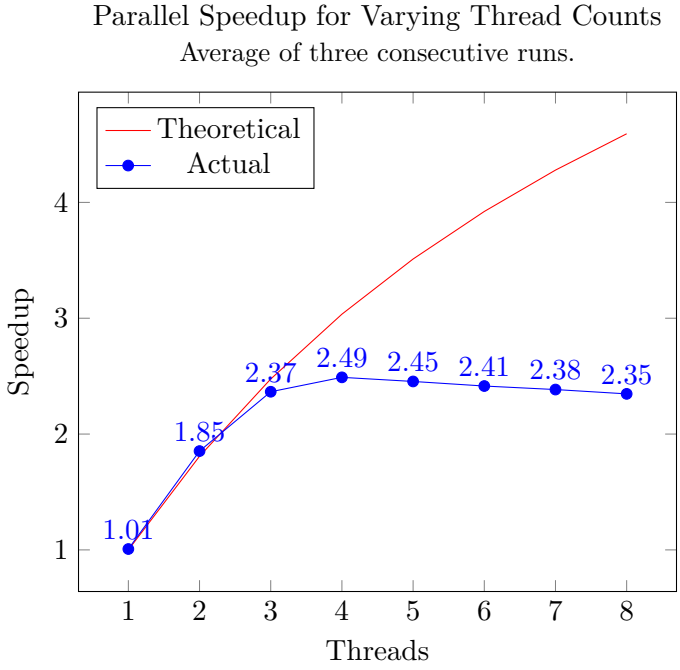


Figure 6.1: Parallel Speedup for Varying Thread Counts

Results were somewhat as expected, with the parallel program showing great performance improvements from the utilization of multiple threads, up to 4. However, as can be seen in the results, the speedup steadily declined as 5 to 8 threads were used.

It was initially thought that this may be due to the relatively low input data size leading to thread starvation as – per the results graphed in Appendix C and further covered in Section 7.3 – it was found that increasing the number of input data files resulted in higher speedup multipliers.

However, after further profiling, it was found that increasing the input data size had no effect on the thread count speedup, and the same declination beyond 4 threads was examined.

With this threshold, coincidentally matching the number of physical cores on the test machine, it was concluded that the issue lie within the utilization of virtual cores, as further discussed in Section 7.2, and for this reason a limit of 4 threads will be placed on the final parallel program.

As can also be seen in the results, the examined speedup (excluding the skewed results of thread counts beyond 4) undercut the theorized speedup determined by Amdahl's Law. Again, this is somewhat as expected, as the parallelization is not perfectly efficient given the overheads the implementation brings with it. However, as to the degree of these differences, and to why the speedup of thread count 2 seemingly exceeded the theorized maximum, it is unknown.

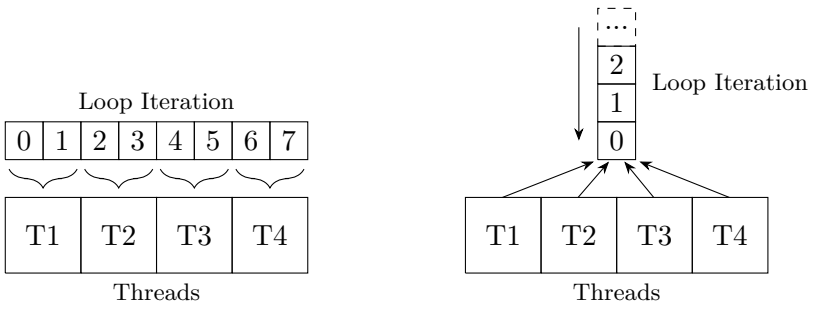
It is also unknown why the single-thread limited parallel program slightly outperformed (by 1%) the untouched sequential program, as their should be no difference but slight overheads with the inclusion of OpenMP compiler directives. It is possible, however, that there are differences in how OpenMP programs are compiled and threads deal with certain tasks, leading to improved execution.

Chapter 7

Limitations and Challenges

7.1 Scheduling

When implementing loop parallelism with OpenMP, each iteration of the loop, by default, is evenly divided between all threads. This is known as static scheduling, a visualization of which can be seen in Figure 7.1a below in which a loop containing 8 iterations has its work delegated to 4 threads.



(a) Static Loop Task Scheduling (b) Dynamic Loop Task Scheduling

Figure 7.1: Visualizations of Loop Task Scheduling

This method of scheduling, however, can be detrimental if the amount of work to be completed within each loop iteration is not fixed. Given the nature of the program, whereby each data file can be of different lengths, static scheduling can result in thread starvation as work loads are not evenly distributed.

Dynamic scheduling is an alternate method of delegating loop iteration work, in which each task to be completed is delegated to threads in a queue-like fashion, as can be seen in Figure 7.1b above. Rather than static scheduling, where a thread cannot complete more work than it is initially assigned, dynamic scheduling allows threads that have completed a task to request and handle the next task in the queue.

With this dynamic scheduling however comes increased overheads, and the possibility of thread starvation if the work to be completed can be completed fast enough by less than the created number of threads. In this case however, these issues are no problem and dynamic scheduling improves the parallel speedup of the program by an extra 30%, as can be seen in the results graphed in Appendix B.

7.2 Virtual Cores

As previously mentioned in Chapter 6, the speedup of the parallel application declines beyond the utilization of 4 threads. This has been concluded to be the fault and limitation of the utilization of virtual cores created via Hyperthreading.

While no expert on the matter, it is thought to perhaps be an issue regarding memory contention and bottlenecking. There exists only 4 physical cores, thus the point where 5 to 8 threads are created, any one core will be dealing with 2 threads.

For this reason, the maximum number of threads that the program can utilize will be limited, using the OpenMP function `omp_set_num_threads`, to 4, ensuring the maximum speedup as seen graphed in Figure 6.1.

7.3 Input Data Size

As can be seen in the profiling results in Appendix C, it was found that increasing the size of the input data, that being the number of sequence data files to compare, increased the overall speedup of the parallel application compared to smaller input sizes utilizing the same number of threads.

This improved performance is thought to relate to the decreasing overhead of dynamic scheduling relative to the work to be completed. With increasing workloads, overheads become less apparent as less time is spent creating/delegating threads and more time doing work.

As further discussed in Chapter 9 however, this discovery was rather encumbersome, as it exposed characteristics of the parallel program for which complete understanding and reason was not known nor readily obtainable.

Chapter 8

Additional Code

As previously mentioned, in-code compiler directives have be used to achieve parallelism with OpenMP. The following `parallel for` directive, as can be seen below, has been used to implement loop parallelism (as discussed in Section 5.2) within the sequential program.

```
#pragma omp parallel for schedule(dynamic)
```

This directive has been used in two locations, once at the loop for forming composition vectors, and again for their comparison, the code for which can be seen in Appendix D. As can also be seen, it is within this directive that a scheduling method is specified, in this case dynamic, as discussed in Section 7.1 to further improve parallel performance.

The second directive that was used within the program was the `critical` directive, as can be seen below. It is used to define a section of code within a parallel task that will run sequentially, limited to one thread at a time.

```
#pragma omp critical
```

This directive has been used to sequentialize the data file reads within the task of forming composition vectors, preventing I/O bottlenecks, as can be seen in Appendix E.

With these directives, the successful implementation of the proposed parallelism, as previously discussed in Chapter 3 and visualised in Appendix A, has been fully achieved.

The total lines of codes added to the sequential program to achieve this parallel transformation was 5, including 2 brace lines, and of course excluding other modifications discussed in this report made to the sequential program alone.

Chapter 9

Reflection

While considered a success, the parallelization of the sequential program is not without its flaws. As discussed, there were many abnormalities, for the most part unexpected, that arose during development and profiling. Certain aspects affecting the speedup, such as the input data sizing and virtual core limitations, were discovered relatively late in the parallelization of the program and should have been considered and tested sooner. At times like these, the direction and flow of the report, and even understanding of the underlying processes within the application, would somewhat be disrupted as these relatively major characteristics of the program were observed and taken into account.

There were also limitations that came with the lack of complete understanding of the sequential program. While there was certainly enough knowledge to comprehend the purpose and tasks of the program, and to implement parallelism, further optimizations could have undoubtedly been made if a deeper and lower-level understanding was developed. However, with the original sequential program's lack of documentation and for the most part obscure variable names, such a grasp was difficult.

Bibliography

[1] Alberto Apostolico and Olgert Denas. Fast algorithms for computing sequence distances by exhaustive substring composition. *Algorithms for Molecular Biology*, 3(1):13, Oct 2008.

[2] Blaise Barney. OpenMP Tutorial, 2017. https://computing.llnl.gov/tutorials/openMP/.

[3] Intel Corporation. Intel VTune Amplifier, 2017. https://software.intel.com/en-us/intel-vtune-amplifier-xe.

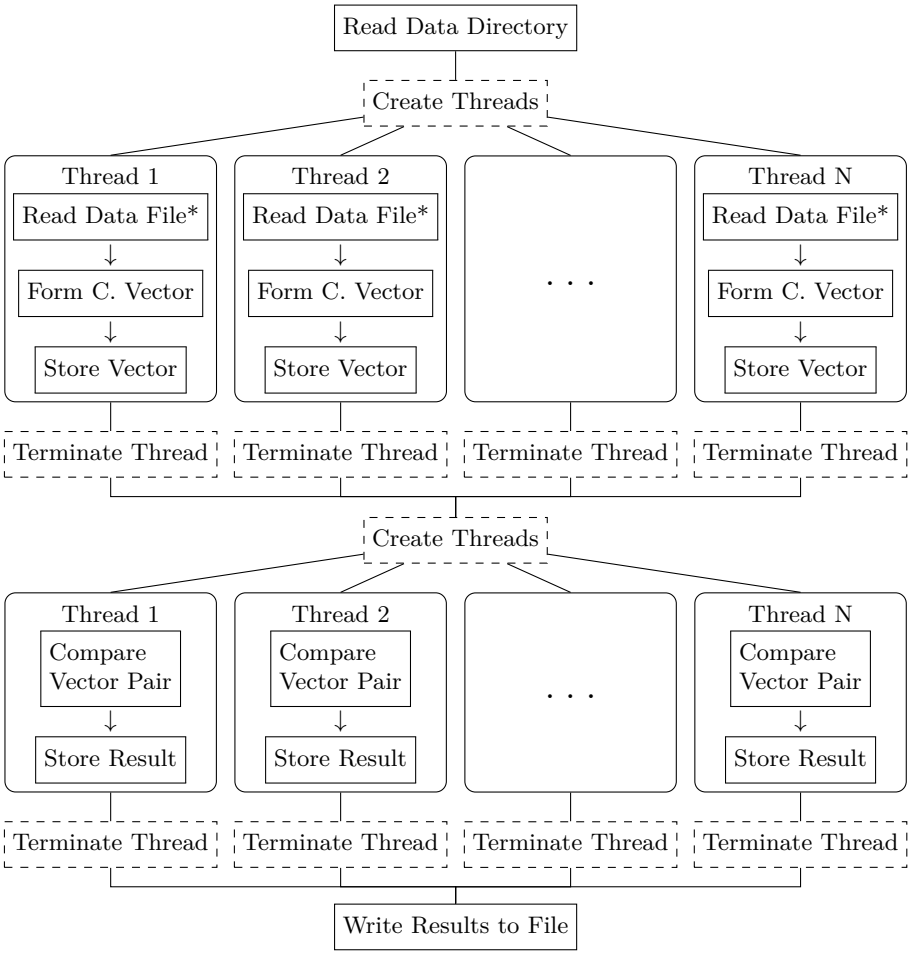
[4] Victor Eijkhout. OpenMP Topic: Loop Parallelism, 2016. http://pages.tacc.utexas.edu/ eijkhout/pcse/html/omp-loop.html.

[5] Carsten Kemena and Cedric Notredame. Upcoming challenges for multiple sequence alignment methods in the high-throughput era. *Bioinformatics*, 25(19):2455–2465, 2009.

[6] Susana Vinga and Jonas Almeida. Alignment-free sequence comparison—a review. *Bioinformatics*, 19(4):513–523, 2003.

Appendix A

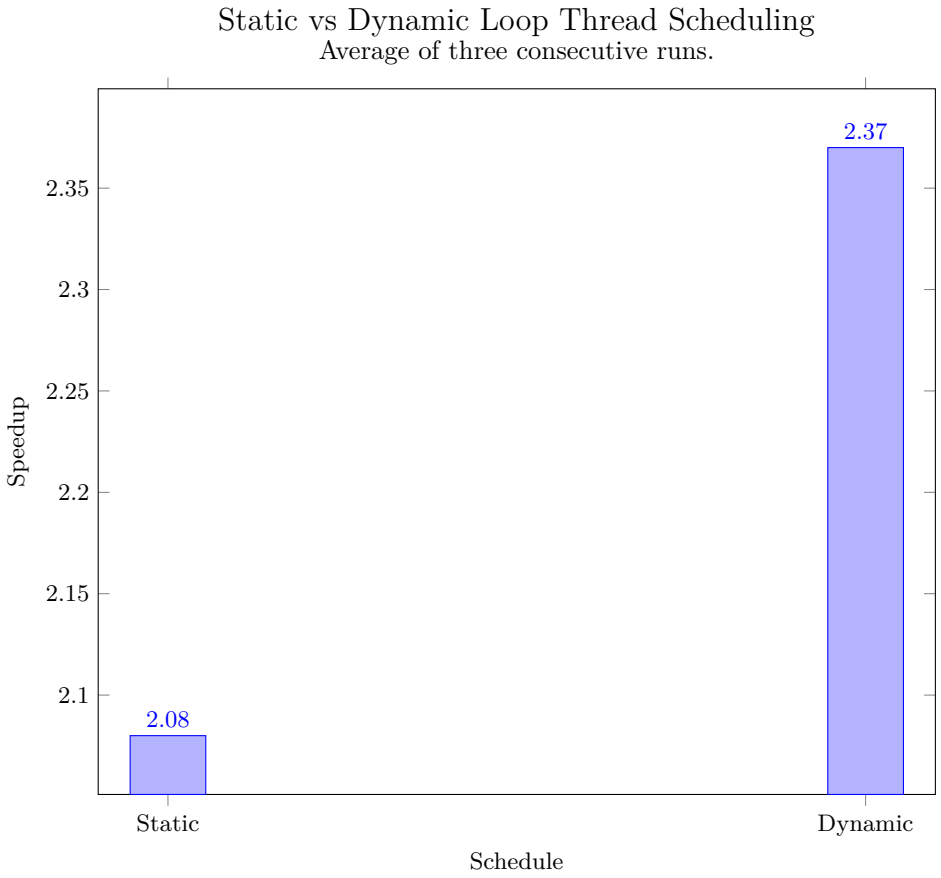
Parallel Program Flow



*Task can only be worked on by 1 thread at a time.

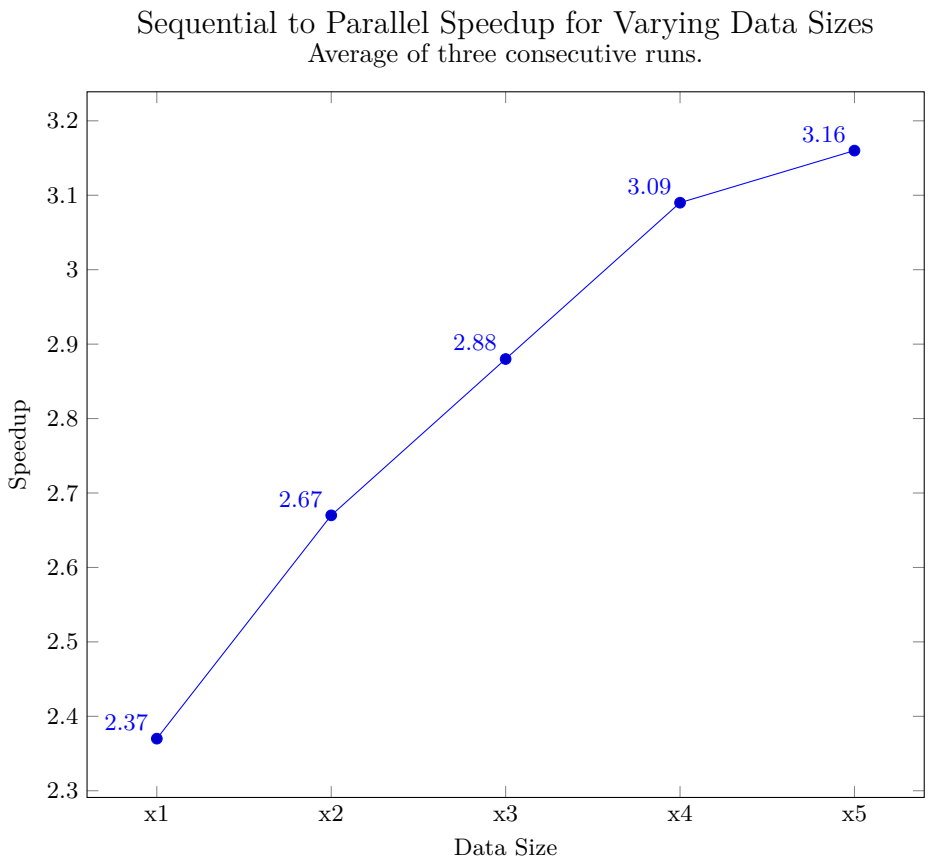
Appendix B

Loop Thread Scheduling



Appendix C

Increasing Input Data Sizes



Appendix D

OMP For-Loop Directive

```
1  #pragma omp parallel for schedule(dynamic)
2  for(int i = 0; i < number_bacteria; i++) {
3      b[i] = new Bacteria(bacteria_name[i].c_str());
4  }
5
6  ...
7
8  #pragma omp parallel for schedule(dynamic)
9  for(int i = 0; i < number_bacteria-1; i++) {
10     results[i] = vector<double>(number_bacteria, 0.0);
11     for(int j = i+1; j < number_bacteria; j++) {
12         double correlation = CompareBacteria(b[i], b[j]);
13         results[i][j] = correlation;
14     }
15 }
```

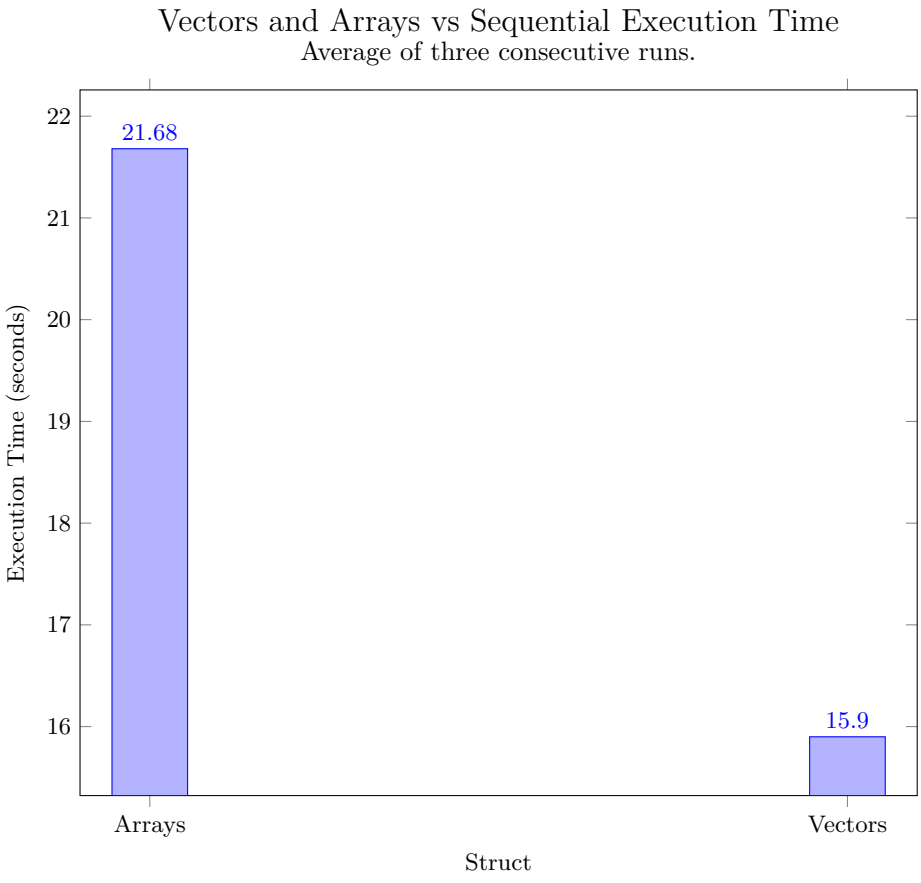
Appendix E

OMP Critical Directive

```
1  #pragma omp critical
2  {
3      FILE * bacteria_file = fopen(filename,"r");
4      if (bacteria_file == NULL) {
5          throw invalid_argument("BAD␣DATA␣DIRECTORY.␣
               Please␣specify␣full␣relative␣path.");
6      }
7      InitVectors();
8
9      char ch;
10     while ((ch = fgetc(bacteria_file)) != EOF) {
11         if (ch == '>') {
12             while (fgetc(bacteria_file) != '\n'); // skip
                   rest of line
13
14             char buffer[LEN-1];
15             fread(buffer, sizeof(char), LEN-1,
                   bacteria_file);
16             init_buffer(buffer);
17         } else if (ch == '\r') {
18             // Skip carriage return.
19         } else if (ch != '\n') {
20             cont_buffer(ch);
21         }
22     }
23     fclose (bacteria_file);
24 }
```

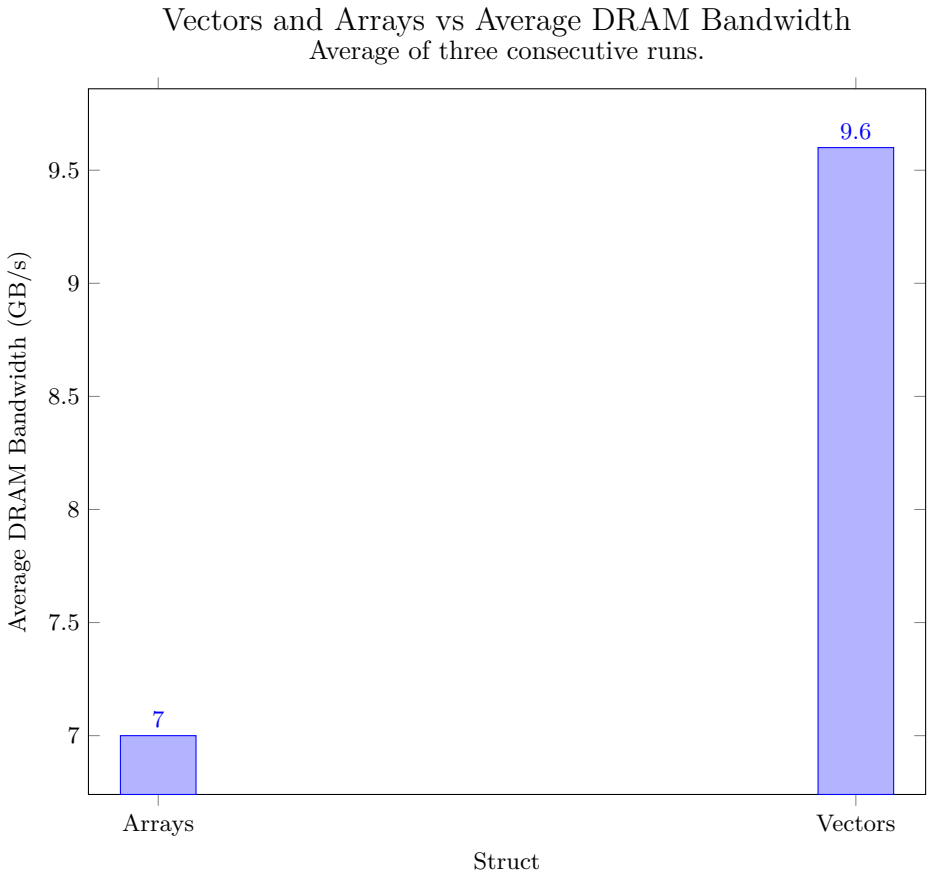
Appendix F

Vector Execution Time



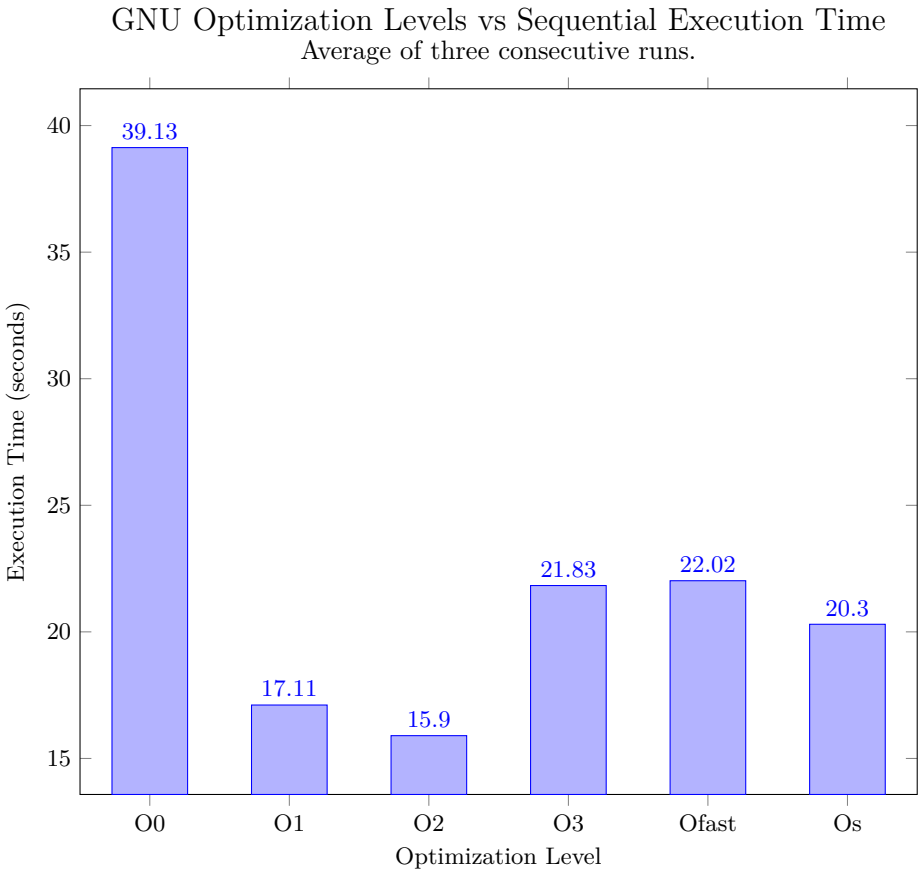
Appendix G

Vector Memory Bandwidth



Appendix H

Compiler Optimizations



Appendix I

Original Data Referencing

```
1  void ReadInputFile(char* input_name)
2  {
3      FILE* input_file = fopen(input_name,"r");
4      fscanf(input_file,"%d",&number_bacteria);
5      bacteria_name = new char*[number_bacteria];
6
7      for(long i=0;i<number_bacteria;i++)
8      {
9          bacteria_name[i] = new char[20];
10         fscanf(input_file, "%s", bacteria_name[i]);
11         strcat(bacteria_name[i],".faa");
12     }
13
14     fclose(input_file);
15 }
```

Appendix J

Dynamic Data Referencing

```
1  void ReadDataDir(char* data_dir) {
2      // Retrieve filenames in data directory.
3      // Based on https://stackoverflow.com/a/612176
4      DIR *dir;
5      struct dirent *ent;
6      if ((dir = opendir (data_dir)) != NULL) {
7          while ((ent = readdir(dir)) != NULL) {
8              string filename = ent->d_name;
9              if (filename != "." && filename != "..") {
10                 bacteria_name.push_back(data_dir +
                       filename);
11                 number_bacteria++;
12             }
13         }
14         closedir (dir);
15         sort(bacteria_name.begin(), bacteria_name.end());
16     } else {
17         perror ("");
18     }
19 }
```

Appendix K

Sequence Data Reading

```
1  char ch;
2  while ((ch = fgetc(bacteria_file)) != EOF) {
3      if (ch == '>') {
4          while (fgetc(bacteria_file) != '\n'); // skip
               rest of line
5
6          char buffer[LEN-1];
7          fread(buffer, sizeof(char), LEN-1, bacteria_file)
               ;
8          init_buffer(buffer);
9      } else if (ch == '\r') {
10         // Skip carriage return.
11     } else if (ch != '\n') {
12         cont_buffer(ch);
13     }
14 }
```

Appendix L

Result Outputting

```
1 ofstream output;
2 output.open("results-sequential.csv");
3 output << "indexA" << ',' << "bacteriaA" << ',' << "
     indexB" << ',' << "bacteriaB" << ',' << "correlation"
      << endl;
4 for (int i = 0; i < number_bacteria-1; i++) {
5     for (int j = i+1; j < number_bacteria; j++) {
6         output << i << ',' << bacteria_name[i] << ',' <<
              j << ',' << bacteria_name[j] << ',' <<
              results[i][j] << endl;
7     }
8 }
9 output.close();
```