CVTree – Genome Similarity

Rubin Mathew

N10513400

Sem 2

2022

Understanding the Program

Composition Vector Tree, or CVTree, is a useful tool in determining the phylogenetic relationships between genome sequences i.e., determining how similar genome sequences are to each other [1]. An implementation of the CVTree software will be investigated in this report in an attempt to safely parallelise and optimise the code for significant performance improvements. The first step in achieving this is to understand the program and discover any bottlenecks and data dependencies

A genome is an organism's genetic information which can be represented as sequence of twenty characters from the alphabet. These sequences are often thousands of characters long, but can be grouped into consecutively occurring characters called k-mers, where k corresponds to the group size. This is fundamental to how the CVTree application is able to analyse genome information as will be discussed below.

A high-level function call graph for the CVTree application can be seen in Figure 1. Firstly, the program reads in the list of bacteria genome files using `ReadInputFile()`, which specifies all genome files that is to be processed. This information is then available to `CompareAllBacteria`(), which first begins by instantiating a `Bacteria` object for each bacteria file. The constructor of `Bacteria` reads in the corresponding genome file and computes the frequency vectors for 6-mers, 5-mers and 1-mers in the genome sequence using a sliding window technique. After this is performed for all the files, `CompareBacteria()` is called on each bacteria pair combination, which performs comparative analysis using the previously computed frequency vectors (Composition Vectors), finally outputting the correlation between the two bacteria.

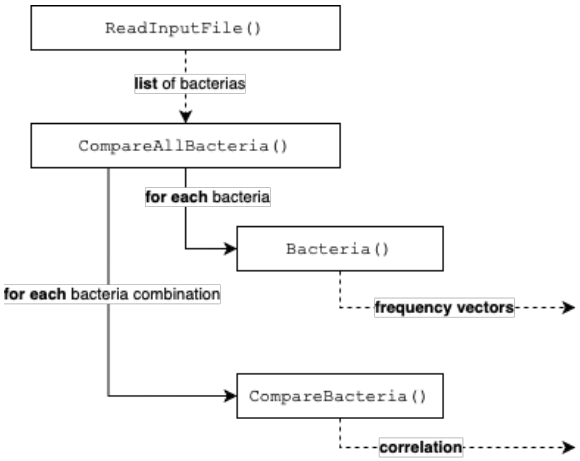


Figure 1 High-level program call graph, where functions are called from top to bottom

Data Dependencies

Uncovering data dependencies is critical to exploiting any potential parallelism. There exist flow dependencies in the sequential CVTree application, which are not immediately apparent when looking at Figure 1. However, after careful analysis, one can realise that for each function to be successfully executed, the previous function above it must be executed first, and so on [Refer to Figure 1]. The following figure outlines the flow dependence in terms of data specifically, with reference to the variable name/s affected.

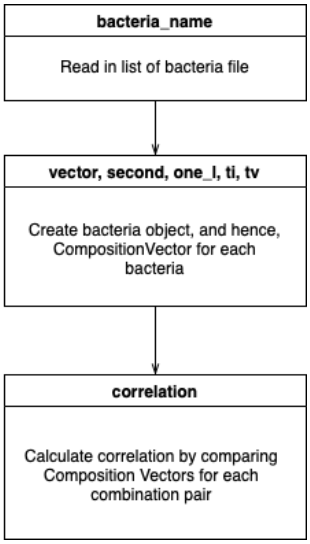


Figure 2 Flow dependency visualisation with reference to actual in-code variables

Potential Parallelism

Based on the data dependency analysis, it is evident that all stages in Figure 2 can be safely parallelised one after the other, such that the flow dependence is satisfied. However, this does not necessarily equate to guaranteed performance since the overhead of creating threads can potentially degrade speed. Consequently, to determine what portions of the program is worth parallelising, based on CPU usage, Visual Studio Profiler was used.

Table 1 Profiling results

Function Name	Total CPU %	Self CPU %
`main`	99.91%	0%
• `CompareAllBacteria`	99.91%	0%
o `Bacteria`	72.82%	36.16%
o `CompareBacteria`	26.97%	26.36%

As seen in Table 1, the `Bacteria` constructor and `CompareBacteria` have the highest CPU utilisation with 72.82% and 26.97% respectively. The major loops within each function are `while` loops and have no alternative loop-transformations that can uncover parallelism. Regardless, this would be tackling fine-grain parallelism which often does not achieve optimal speed-up, due to overheads in thread creation, synchronisation and communication.

Instead, it can be seen in Figure 3 that the parent function `CompareAllBacteria` contains easily parallelisable `for` loops which execute larger computation sizes, with no inter-dependencies between iterations. The large computations can be visualised from the red highlights on line 260 (`Bacteria`) and 271 (`CompareBacteria`), each of which is nested inside a separate `for` loop [Refer to Figure 3]. Consequently, it is apparent that the `for` loops on line 257 and 267, corresponding to `Bacteria` and `CompareBacteria` respectively, can be parallelised to exploit course-grain parallelism. Though there are two nested `for` loops for `CompareBacteria`, only the outer `for` loop will be parallelised to further exploit course-grain parallelism.

This program should achieve scalable parallelism, because as the input size increases (i.e., the number of bacteria) there will be more iterations of the two aforementioned `for` loops, meaning more cores can be used to parallelise those iterations and hence speed up the program.

```
                    253  void CompareAllBacteria()
                    254  {
                    255      Bacteria** b = new Bacteria * [number_bacteria];
                    256      auto start_time = std::chrono::high_resolution_clock::now();
                    257      for (int i = 0; i < number_bacteria; i++)
                    258      {
                    259          printf("load %d of %d\n", i + 1, number_bacteria);
242357 (72.82%)     260          b[i] = new Bacteria(bacteria_name[i]);
                    261      }
                    262
                    263      auto end_time = std::chrono::high_resolution_clock::now();
                    264      std::chrono::duration<double> duration = end_time - start_time;
                    265      printf("loading finished: %f seconds\n", duration.count());
                    266
      9 (0.00%)     267      for (int i = 0; i < number_bacteria - 1; i++)
      1 (0.00%)     268          for (int j = i + 1; j < number_bacteria; j++)
                    269          {
                    270              printf("%2d %2d -> ", i, j);
 89867 (27.00%)     271              double correlation = CompareBacteria(b[i], b[j]);
                    272              printf("%.20lf\n", correlation);
    268 (0.08%)     273          }
                    274  }
```

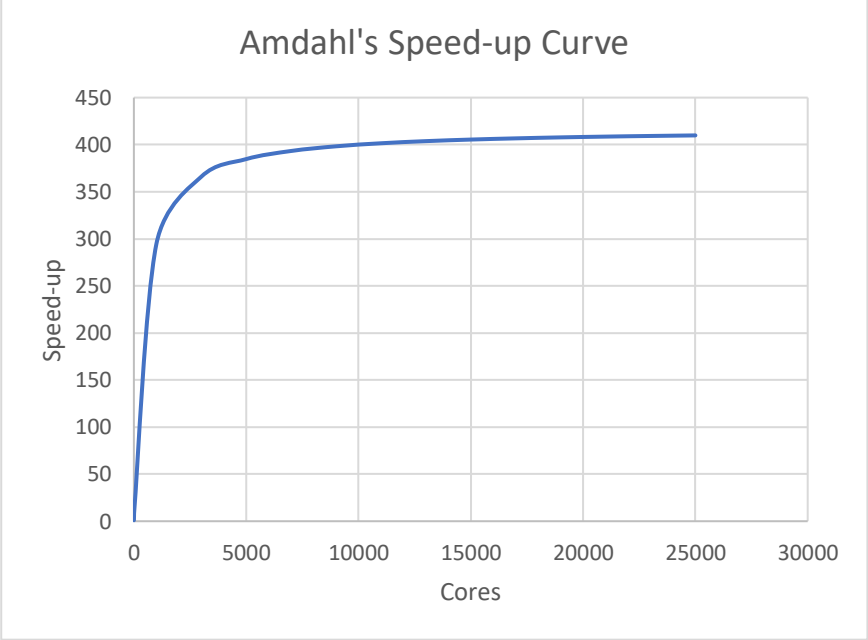
Figure 3 Visual Studio Profiler CPU intensive code highlighting

Amdahl's Law can be used to approximate the expected speed-up curve as the number of cores increase with the following equation:

$$S = \frac{1}{(1-P) + (\frac{P}{s})}$$

where S is the speed-up, P is the fraction of the program that can be parallelised, and s is the number of cores.

It is known from the Table 1 that the parallelisable portion of the code takes up a total of 99.76% of execution time. According to the profiler, the other parallelisable code, `ReadInputFile` function, accounted for 0% of the execution time, as it was dominated by the `CompareAllBacteria`, and hence is not worth parallelising. Therefore, knowing this information the following theoretical speed-up graph can be created.



Parallelisation - OpenMP

As aforementioned, the `for` loops to be parallelised have no inter-dependencies between iterations. Consequently, it's inherent parallelism can be easily exploited using an implicit parallelism model such as OpenMP.

Initial Program Changes

OpenMP parallelisation can be achieved by using compiler directives. The main directive that is used is:

```
#pragma omp parallel for
```

as seen in line 467 and 476 [Refer to Figure 4]. This tells the compiler to delegate the iterations of the `for` loop to slave threads. The default scheduling for threads is static, in which each slave thread is allocated approximately equal number of iterations. However, this can be detrimental when iteration computation sizes are not similar, like in the case of different sized bacteria genome files [Refer to Appendix A], resulting in thread starvation. To solve this, an additional keyword:

```
schedule(dynamic)
```

is defined which results in the iteration being dynamically assigned. This means when a slave thread is finished with its allocated work, it will be assigned a new work load if available, instead of it sitting idle. The size of the new work load is dynamically assigned by the compiler. Testing proves this performs around **11%** faster than static scheduling.

On line 476, you will also notice the keyword:

```
private(j)
```

this ensures that each slave thread has its own local reference of the variable `j`, and that it does not share it with other threads [Refer to Figure 4]. Without this keyword, slave threads can collide causing some `j` iterations to be overwritten and skipped since they are pointing to the same memory address. Alternatively, you can declare `j` inside the for loop and the compiler will implicitly know that it should be private. Similarly, the following directive:

```
#pragma omp critical
```

seen on line 481 defines a critical section in which no more than one thread can be inside at a time [Refer to Figure 4]. This is important as the `printf()` function or file writes are not atomic, hence, without the critical section, when these functions are called at the same time by two threads, race conditions, overwrites and odd behaviour can occur. An attempt was made to write the results to a 2D array first, and then print it outside the parallel code to reduce synchronisation overheads, but this had no perceivable improvement.

Speaking of file writes, the parallel program was tested for correctness by outputting the results of the sequential and parallel program into a csv format and comparing them using a python script seen in Appendix B. The commented-out code for writing the results to a csv file is seen in line 475, 485 and 489 [Refer to Figure 4]. The `printf()` on line 471 was deemed redundant and hence was commented out for a **4%** faster execution time.

Finally, to initialise the maximum number of threads to be used in parallel regions, in the main function the following directive was declared:

```
omp_set_num_threads(NUM_THREADS)
```

where `NUM_THREADS` is a macro indicating the number of threads wanted.

```
252  void CompareAllBacteria()
253  {
254      Bacteria** b = new Bacteria * [number_bacteria];
255      #pragma omp parallel for schedule(dynamic)
256      for (int i = 0; i < number_bacteria; i++)
257      {
258          b[i] = new Bacteria(bacteria_name[i]);
259      }
260
261      #pragma omp parallel for schedule(dynamic)
262      for (int i = 0; i < number_bacteria - 1; i++)
263          for (int j = i + 1; j < number_bacteria; j++)
264          {
265              double correlation = CompareBacteria(b[i], b[j]);
266              #pragma omp critical
267              {
268                  printf("%2d %2d -> %.20lf\n", i, j, correlation);
269              }
270
271          }
272  }
```

Figure 4 The parallelised `CompareAllBacteria` function

During a debug step-through of the program, it was noticed that an array `t` of the same size as `vector` was being initialised, however, this was unnecessary and computationally expensive. The profiler showed within `Bacteria`, the creation of array with the `new` keyword accounted for 7.2% of CPU time, whilst delete was 6.47%. A similar problem was noticed for other arrays to hence, code was rewritten to reuse arrays as much as possible. This can be seen in Figure 5, in which the `t` array was replaced with the `vector` array instead. This decreased the execution time by **16.54%.**

```
149              if (stochastic > EPSILON)
150              {
151                  vector[i] = (vector[i] - stochastic) / stochastic;
152                  count++;
153              }
154              else
155                  vector[i] = 0;
156          }
```

Figure 5 Reusing of arrays. 't' array was replaced with 'vector' array

Timing tests were performed on this optimised program on machine with 6 physical cores, 12 virtual cores, and 16GB memory. Unfortunately, when testing the program with increasing thread count, it was found that the speedup plateaued at a low 2.5 around six cores.

Overcoming Barriers

Countless hours were spent trying to diagnose the issue causing the speedup to plateau at 6 threads. Fortunately, two major amendments were implemented that drastically improved the performance. Both of these improvements exploit data locality by implementing an `unordered_map<>` and `vector<pairs<>>`.

First Barrier

The first amendment was discovered whilst implementing instrumentation, via a `std::vector` datatype to look inside the variable `vector`. Here, it was noticed that although `vector` was assigned memory to store 64,000,000 double types, it only ends up storing sometimes thousands of orders of magnitude smaller number of frequencies. Moreover, frequency values more than 0 were separated by a sea of zeros. The remaining redundant zero frequencies take up considerable amount of memory and hence isn't cache friendly. Additionally, since this large array is allocated in heap memory, countless cache line reads have to be made in heap to fully process it, which can be extremely slow and inefficient; and not mention flushes the cache of data that could be reused.

The purpose of the frequency vectors, especially that of `vector`, is perfectly suited for a data-structure such as a HashMap. The key can be the index and the value can be the frequency. This way only 6-mers that occur at least once exists inside the HashMap, significantly reducing its memory footprint. Consequently, spatial locality can be exploited i.e., it becomes more likely that an item that will be used very soon is read into the cache simply by existing in the same cache line as a previously used key-value pair. If a key does not exist in HashMap, it is then fair to assume the frequency is zero. The HashMap was implemented with the `unordered_map<>` type in C++ as follows:

```
std::unordered_map<long, double> vector;
```

Second Barrier

Till now, most optimisation efforts have been focused in the `Bacteria` class, however since it is optimised completely (if not mostly) now, attention can now be redirected at the 2nd most CPU intensive operation, the `CompareBacteria` function. After careful analysis and teaching team input, efforts were made to thoroughly analyse the data access pattern within the `while` loop inside `CompareBacteria`. There are two large arrays being accessed for each `Bacteria` pointer, `tv` and `ti`. This can be seen in Figure 6 where `tv` and `ti` are accessed one after the other for each bacterium. In the worst case four different arrays are being consecutively read in one iteration of the while loop i.e., operations on line 213, 214, 229 and 230 [Refer to Figure 6]. This will result in extremely inefficient cache access, as the consecutive reads of unique arrays will read in cache lines from each arrays memory, resulting in the cache being flushed, and limiting any potential of spatial locality for future loops of the `while`. Similarly, further down in the function, values from the `ti` array are read but never used, as seen in line 238 and 244 [Refer to Figure *7*]. This is detrimental to the cache as unnecessary cache line data will be read in, flushing any cached `tv` data which is clearly being contiguously read from memory in the while loops in line 236 and 242 [Refer to Figure *7*].

```
204  double CompareBacteria(Bacteria* b1, Bacteria* b2)
205  {
206      double correlation = 0;
207      double vector_len1 = 0;
208      double vector_len2 = 0;
209      long p1 = 0;
210      long p2 = 0;
211      while (p1 < b1->count && p2 < b2->count)
212      {
213          long n1 = b1->ti[p1];
214          long n2 = b2->ti[p2];
215          if (n1 < n2)
216          {
217              double t1 = b1->tv[p1];
218              vector_len1 += (t1 * t1);
219              p1++;
220          }
221          else if (n2 < n1)
222          {
223              double t2 = b2->tv[p2];
224              p2++;
225              vector_len2 += (t2 * t2);
226          }
227          else
228          {
229              double t1 = b1->tv[p1++];
230              double t2 = b2->tv[p2++];
231              vector_len1 += (t1 * t1);
232              vector_len2 += (t2 * t2);
233              correlation += t1 * t2;
234          }
235      }
```

Figure 6 A snippet of the original `CompareBacteria()` function

```
236      while (p1 < b1->count)
237      {
238          long n1 = b1->ti[p1];
239          double t1 = b1->tv[p1++];
240          vector_len1 += (t1 * t1);
241      }
242      while (p2 < b2->count)
243      {
244          long n2 = b2->ti[p2];
245          double t2 = b2->tv[p2++];
246          vector_len2 += (t2 * t2);
247      }
248
249      return correlation / (sqrt(vector_len1) * sqrt(vector_len2));
250  }
251
```

Figure 7 Unnecessary array reads in `CompareBacteria`

To overcome this, firstly redundant lines 238 and 244 can be removed entirely to exploit spatial locality better. Secondly, the `tv` and `ti` arrays can be combined and represented by a `vector` of `pairs`, where the `pair`'s first value is a `tv` value and the second value is a `ti` value. This means one read of a `pair` from the `vector` will almost guarantee the pair's second value remains in the cache, for faster reads, and possibly consecutive pairs, hence further exploiting spatial locality. The refactoring of code within `CompareBacteria` to utilise this new data structure called `tvVector` can be seen in Figure 8. The initialisation of `tvVector` in `Bacteria` can be seen in Figure 9 and the signature for it below:

```
std::vector<std::pair<long, double>> tvVector;
```

```
while (p1 < b1->count && p2 < b2->count)
{
    long n1 = b1->tvVector[p1].first;
    long n2 = b2->tvVector[p2].first;
    if (n1 < n2)
    {
        double t1 = b1->tvVector[p1].second;
        vector_len1 += (t1 * t1);
        p1++;
    }
    else if (n2 < n1)
    {
        double t2 = b2->tvVector[p2].second;
        p2++;
        vector_len2 += (t2 * t2);
    }
    else
    {
        double t1 = b1->tvVector[p1++].second;
        double t2 = b2->tvVector[p2++].second;
        vector_len1 += (t1 * t1);
        vector_len2 += (t2 * t2);
        correlation += t1 * t2;
    }
}
while (p1 < b1->count)
{
    double t1 = b1->tvVector[p1++].second;
    vector_len1 += (t1 * t1);
}
while (p2 < b2->count)
{
    double t2 = b2->tvVector[p2++].second;
    vector_len2 += (t2 * t2);
}
```

Figure 8 `CompareBacteria` refactored for vector of pairs

```
            if (stochastic > EPSILON)
            {
                double freq_i = 0.0;
                auto it = vector.find(i);
                if (it != vector.end()) freq_i = it->second;
                count++;
                tvVector.push_back({ i, (freq_i - stochastic) / stochastic });
            }
        }

        delete second;
        fclose(bacteria_file);
    }
};
```

Figure 9 Intialisation of `tvVector` in `Bacteria`

Results

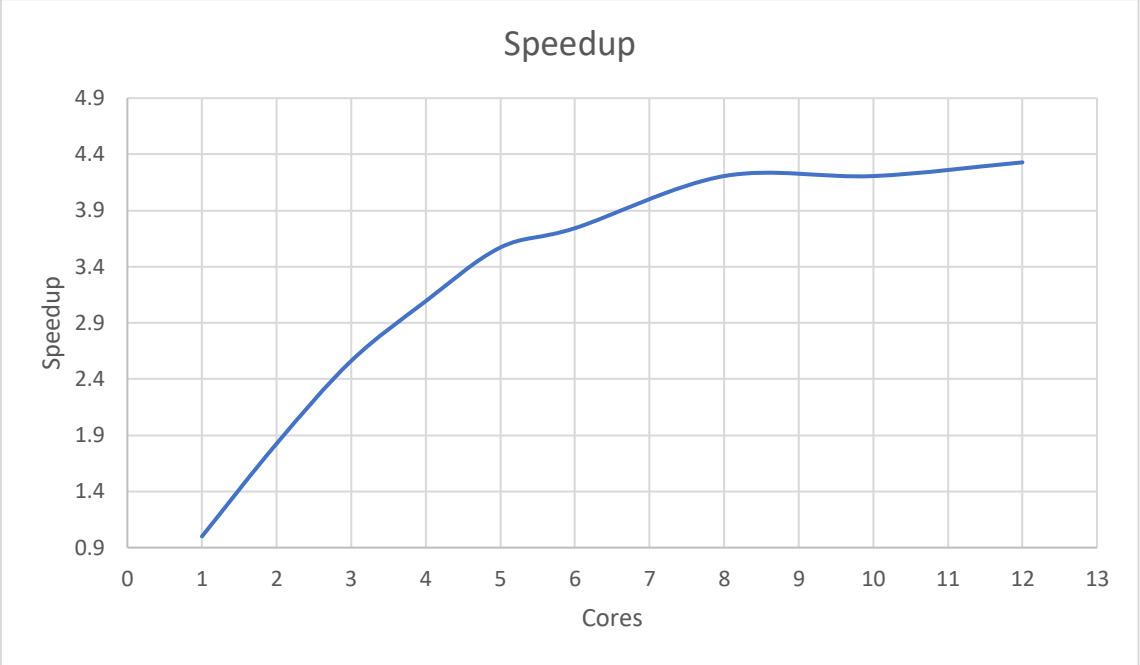


Figure 10 The final speedup curve

Tools

OpenMP

To achieve parallelism, OpenMP employs a fork-join model in which a master thread commissions slave threads and allocates them some portion of work. The master thread must wait for all slave threads to finish their work for it to continue execution [2]. Fortunately, OpenMP maintains a thread pool, and its size can be manually defined, hence the overhead in creating and destroying threads is reduced [3]. OpenMP was chosen because of its simplicity in implementation and ability to mostly preserve the original code, improving readability.

Compilers

The built in C++ compiler in Visual Studio called MicroSoft Visual C++ (MSVC) was used. Within the Visual Studio IDE, there are options to change the compiler behaviour. The most important one that was changed in this project was the compiler optimisers. Four options were provided by the IDE which include /Ox (enables most speed optimisations), /O1 (minimise executable size), and /O2 (maximises speed). Given the nature of this assignment is to explore speed maximations, the /O2 option was chosen which has a noticeable improvement to the other options.

Profiler

Visual Studio Profiler is an inbuilt profiling tool of the Visual Studio IDE, which can profile metrics such as CPU and Memory usage. Moreover, it provides information such as Hot Paths which is extremely useful when it comes to determining code optimisation and code worth parallelising. All profiling was performed in Debug mode, as it provides more symbolic information than Release, allowing Hot Paths to be more human-readable.

Chrono

A high-precision in-code timer provided in the `chrono` library was used for all program execution timing. This ensures any improvements, regardless of how small were accounted for. The timing code was implemented in `main()`, and can be seen in Appendix C.

Python

Python is a general-purpose programming language that was used in this project for two main tasks. The first task and more important functionality that it provided was to validate the output of modified code by

comparing it to the output of the sequential. Results from each program were outputted to a csv for comparison, code showcasing this is available in Appendix D. This csv from both the sequential and modified code where then compared with the code available in Appendix B. Essentially what it does is confirms each row in the sequential program csv file exists in the modified-code csv file. The second task Python was used for was to diagnose a load-imbalance issue with processing the bacteria files. This was achieved by checking all the bacteria genome files and outputting the size to the terminal to be plotted as can be seen in Appendix A and Appendix E.

Reflection

As can be seen in Figure 10, the optimisations have increased the speedup ratio by almost 200% compared to the first attempt. However, the speedup still plateaus now at around 4.4, significantly deviating from the expected speedup of 11.7 at 12 cores according to Amdahl's law. The reason for this is expected to be associated with the Von Neuman Bottleneck, as Amdahl's law doesn't account for it. Overall, optimisation efforts were incredibly successful, and this investigation has certainly provided a deeper understanding of cache access. Despite this, there is one future improvement that can be implemented that was not achieved in this report. This involves putting the HashMap `vector` into a contiguous memory data structure such as a `std::vector` for better cache access.

References

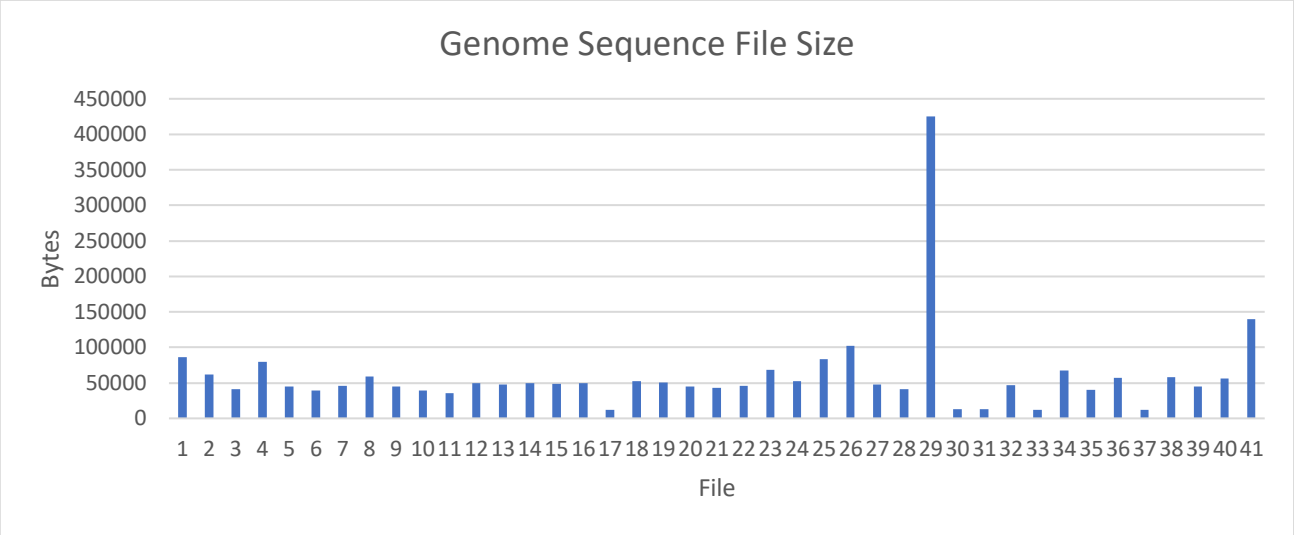
[1] Zuo, G. (2021). CVTree: A parallel alignment-free phylogeny and taxonomy tool based on composition vectors of genomes. *Genomics, proteomics & bioinformatics*, *19*(4), 662-667

[2] Jin, C., & Baskaran, M. (2018, November). Analysis of explicit vs. implicit tasking in OpenMP using kripke. In 2018 IEEE/ACM 4th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2) (pp. 62-70). IEEE.

[3] Oracle. (2010). OpenMP API User Guide: Chapter 4 Nested Parallelism. https://docs.oracle.com/cd/E19205-01/819-5270/aewbc/index.html [Accessed: 20/10/2022]

Appendix

Appendix A



Appendix B

Code to Compare Sequential and Parallel Output csv Files

```
import csv
from distutils.log import error

with open('sequentialResults.csv', 'r') as t1, open('test.csv', 'r') as t2:
    fileone = t1.readlines()
    filetwo = t2.readlines()

error_found = false
for line in filetwo:
    if line not in fileone:
        print("ERROR NO MATCH")
        error_found = True
        break

if error_found:
    print('MATCH ALL GOOD')
```

Appendix C

Chrono Timing Code

```
int main(int argc, char* argv[])
{
    // start timer
    auto start_time = std::chrono::high_resolution_clock::now();

    omp_set_dynamic(0);
```

```
    omp_set_num_threads(12);
    Init();
    ReadInputFile("list.txt");
    CompareAllBacteriaPar();

    // end timer
    auto end_time = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> duration = end_time - start_time;
    printf("time elapsed: %f seconds\n", duration.count());
    return 0;
}
```

Appendix D

Output Results to a CSV

```
void CompareAllBacteria()
{
    Bacteria** b = new Bacteria * [number_bacteria];

    int i;
    #pragma omp parallel for schedule(dynamic)
    for (i = 0; i < number_bacteria; i++)
    {
        b[i] = new Bacteria(bacteria_name[i]);
    }

    int j;
    std::ofstream results("test.csv");
    #pragma omp parallel for private(j) schedule(dynamic)
    for (i = 0; i < number_bacteria - 1; i++)
    {
        for (j = i + 1; j < number_bacteria; j++)
        {
            double ans = CompareBacteria(b[i], b[j]);
            #pragma omp critical
            {
                printf("%2d %2d -> %.20lf\n", i, j, ans);
                results << i << "," << j << "," << ans << "\n";
            }

        }
    }
    results.close();
}
```

Appendix E

Load Imbalance Analysis Python Script

```
import os


os.chdir("data/")
files = os.listdir()
for file in files:
    print(os.path.getsize(file))
```