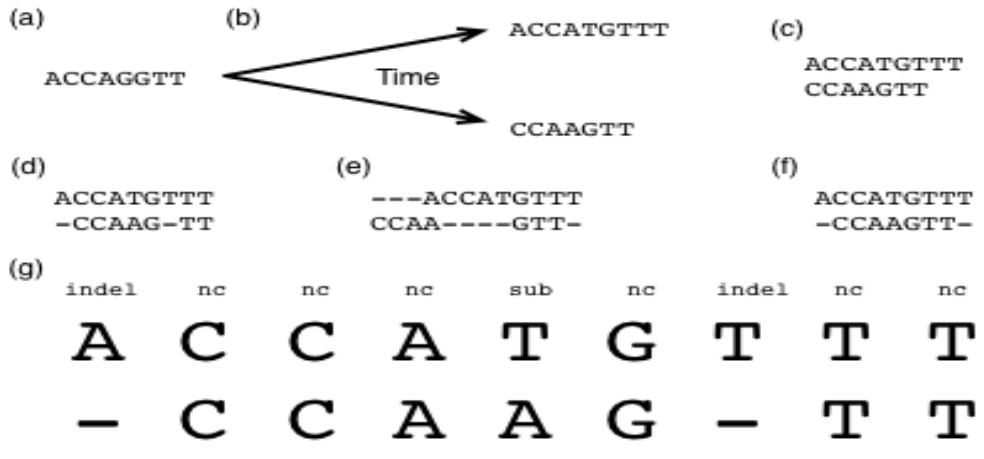
Ebrahim Alhaddad
Juan Pablo Mejia
EE451-Final Project Report

Parallelizing Needleman-Wunsch Algorithm

Introduction:

Within the field of bioinformatics, processing and analyzing protein sequences is of great importance. One way of doing so is to perform string alignment on DNA sequences, also known as pairwise-sequence alignment. Its goal is to rearrange both sequences in search for their most compatible arrangement - An arrangement with the most matching codon positions in both sequences. By doing so, we can study where mutations in the DNA occurred. The sequences analyzed are made up of nucleotides (Adenine, Thymine,Cytosine, and Guanine), where groups of 3 correspond to specific codons. These sequences come in large strings of consecutive uppercase letters where each letter corresponds to a nucleotide as such: A - Adenine, T - Thymine, C - Cytosine, G - Guanine. The length of these sequences are usually very long so aligning them becomes a complex problem to solve. A naive alignment approach is extremely inefficient as it must identify all possible permutations in order to find the optimal arrangement. An example of this approach is shown in Figure 1.1. This example starts with a DNA sequence (ACCAGGTT) and mutates into two different sequences, ACCATGTTT and CCAGTT. The first mutation was given a substitution and an insertion while the second was given a substitution and a deletion. The following steps show different permutations of aligning the two sequences, with step (g) as the optimal alignment. Deletions/insertions of nucleotides are seen represented as dashes and non-matching nucleotides are seen substitutions. Not only does cycling through each permutation of the sequences take a long substantial time, but the insertion/deletion gaps magnify the time complexity costs. A naive solution may require adding excess gaps throughout permutations in search for optimal alignment. This can increase the length of the sequences and consequently increase the data size. Given that M!*N! describes how many iterations are necessary to permute two inputs of size M and N, it is possible to presume that a naive solution is characterized by a time complexity with a lower bound Ω(M!*N!).

Figure 1.1



The Needleman-Wunsch algorithm, created by Saul B. Needleman and Christian D. Wunsch, was developed to alleviate the processing power needed to align large DNAsequences. The algorithm breaks down the complex problem into smaller, simpler problems to find the optimal pairwise-sequence alignment. Therefore instead of requiring a time complexity of O(n!m!) proposed by a naive solution, the Needlemen-Wunsch algorithm can compute an optimal solution in O(mn), where m and n are the length of DNA sequences under study.

Optimizing DNA alignment is crucial to the field of bioinformatics. One of the largest issues in this field is the computation requirements when working with the human genome. Currently the reference genome for a human is approximately 3.2 billion letters long. Where each letter represents the nucleobases that makeup the individual units in a DNA strand. To just identify and compute the complete set of nucleobases,scientists took over 13 year. This was even with a reduced number genome size. The actual size of a human genome is ~6.4 billion letters long, while the number mentioned before was just a reference genome. Algorithms like Needleman-Wunsch help decrease the complexity of performing genome calculations such as DNA alignment. However, the Needleman-Wunsch is still expensive with respect to time and space. Therefore our hope is to decrease these costs by parallelizing the certain aspects of the algorithm.

Context:

The Needleman-Wunsch Algorithm generally takes two inputs representing the DNA sequences under study. These sequences comprise of character arrays where each character is a letter representation of codons in sequential order. After execution, the algorithm outputs the optimal alignment of these sequences.

Our intention before parallelizing the algorithm is to produce a flexible program, which can read files and produce results in an organized and easy-to-read format. Auxiliary classes were designed to support the function of the algorithm, specifically in steps involving reading and writing data to and from files. FASTA files are the universal standard for storing nucleotide sequences and they are commonly used in the field of bioinformatics. Their layout comprises of fields of comments providing information about the sequence, designated by either “;” or “>” characters. The remainder of the files contain a long string of characters representing the sequence itself.

Figure(2.1)

```
>Sequence_2
CCCTAAACCCTAAACCCTAAACCCTAAACCTCTGAATCCTTAATCCCTAAATCCCTAAAT
CTTTAAATCCTACATCCATGAATCCCTAAATACCTAATTCCCTAAACCCGAAACCGGTTT
CTCTGGTTGAAAATCATTGTGTATATAATGATAATTTTATCGTTTTTATGTAATTGCTTA
```

In our program, the FASTAParse class reads in these files, figure(2.1), and isolate the nucleotide sequence such that it is ready for processing as a character array. Command line arguments provide directory path to the appropriate input, and data will be parsed and stored prior to running the algorithm.

The Needleman-Wunsch algorithm requires three main steps to fully compute the optimal solution: Initialization; Processing Nucleotide Pairs; and Backtracking. In addition, the algorithm requires a pair of two dimensional matrices. Given sequence A and sequence B, these sequences

can be represented at the edges of the matrices such that each character in sequence A is associated with each column in the matrix and each character in sequence B is associated with each row in the matrix. Figure(3.1) showcases the aforementioned layout. One of the two matrices is responsible for storing scores as integer values and the other matrix stores directional pointers necessary for finding the optimal solution. The score matrix will store negative and positive integer values while the pointer matrix will store one of three possible directions: *Above, Left, AboveLeft*.

Figure(3.1)

		G	C	A	T	G	C	U
G								
A								
T								
T								
A								
C								
A								

In the initialization step, the first row and first column of the score matrix must be filled with decrementing values while the origin point stores zero shown in figure(3.2). Additionally the pointer matrix must be filled with directional pointers *Left* in each cell of the first row and *above* in each cell of the first column as shown in figure(4.1).

figure(3.2)

		G	C	A	T	G	C	U
	0	-1	-2	-3	-4	-5	-6	-7
G	-1							
A	-2							
T	-3							
T	-4							
A	-5							
C	-6							
A	-7							

figure(4.1)

done	left	left	left	left
up	**diag**			
up				
up				

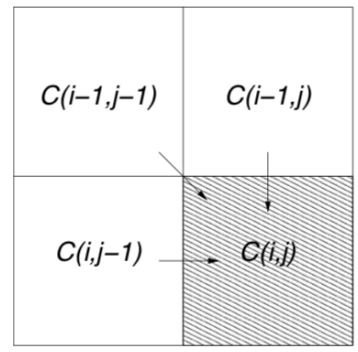
The initialization step requires iterating through a single row and column twice. Therefore, it can be characterized by a time complexity of O(M) where M is the length of the larger sequence.

In the processing step, each cell inside the matrices must be visited once to compute its score and pointer following the relationship described in figure(4.2).

Figure(4.2)

$$\text{score} = \max \begin{cases} F(i-1, j-1) + s(x_i, y_i) \\ F(i, j-1) - \text{gap penalty} \\ F(i-1, j) - \text{gap penalty} \end{cases}$$

Figure (4/3)



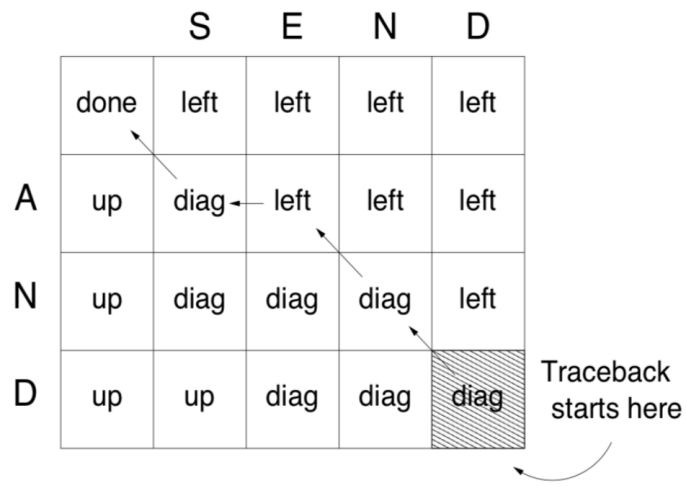
Figure(4.3) shows the dependencies required by the computation. The *diagonal* path is computed by fetching the score of the diagonal neighbor and adding 1 if the nucleotides at index i and j in both sequences match, or subtracting 1 if they mismatch. The *left* path is computed by fetching the score of the left neighbor and subtracting 1 to represent a gap in sequence A. Similarly, the *above* path is computed by fetching the score of the top neighbor and subtracting 1 to represent a gap in sequence B. Once all three paths are examined, the maximum score computed is stored in the score matrix cell. Additionally, a pointer to the path associated with the maximum score is stored in the pointer matrix cell. This process is repeated sequentially following traditional matrix traversal until all cells in both matrices are visited.

In the final step, the algorithm must start at the bottom-right corner of the matrices and find a path back to origin -cell(0,0)- by visiting each cell and following the direction of its pointer. Figure(5.1) shows an example of backtracking. When a *diagonal* pointer is visited, the nucleotides of both sequence at the given index are both printed. When a *left* pointer is visited, only the nucleotide found at index j of sequence A is

printed while sequence B prints a gap. Visiting an *above* pointer will print out the nucleotide found at index i of sequence B while sequence A prints out a gap.

Figure(5.1)

SEND
A-ND



The final alignment of both sequences is stored in two character arrays and printed out in the following format where vertical lines represent matching nucleotides to ease readability.

Objective:

Our goal is to improve the efficiency of Needleman-Wunsch Algorithm using parallelization techniques. We chose two specific parts of the Needlman-Wunsch algorithm to parallelize, the initialization of the two matrices and the computation required to fill each matrix. The former was done in two different approaches as described in the proceeding section

Project Parallelization:

Parallel Initialization of Matrices - Approach #1

The first approach only utilized two different threads to initialize the two matrices. As shown in figures 6.1 and 6.2, Thread 1 was responsible for filling the first row of the scoring grid and the pointer matrix with decrementing values and left pointers, respectively. Thread 2 was responsible for filling the first column of the scoring matrix with decrementing values and above pointers, respectively.

Figure (6.1)

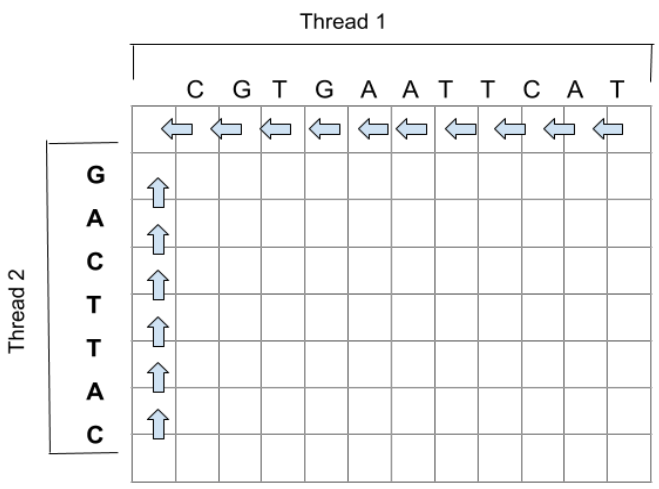
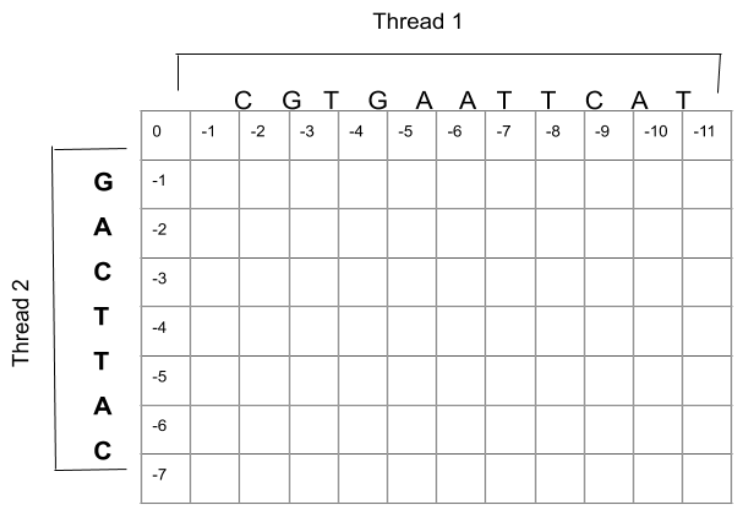


Figure (6.2)



Parallel Initialization of Matrices - Approach #2:

The second approach instead partitioned sections of the first row and the first column to each thread. The size of each partition depended on the number of threads specified. As figure 6.3 shows, each thread was given a section of the first row and the first column, and then was given the task of filling their respective cells with the appropriate values. The end result of both approaches is shown as the superimposed matrix shown in figure 6.4.

Figure (6.3)

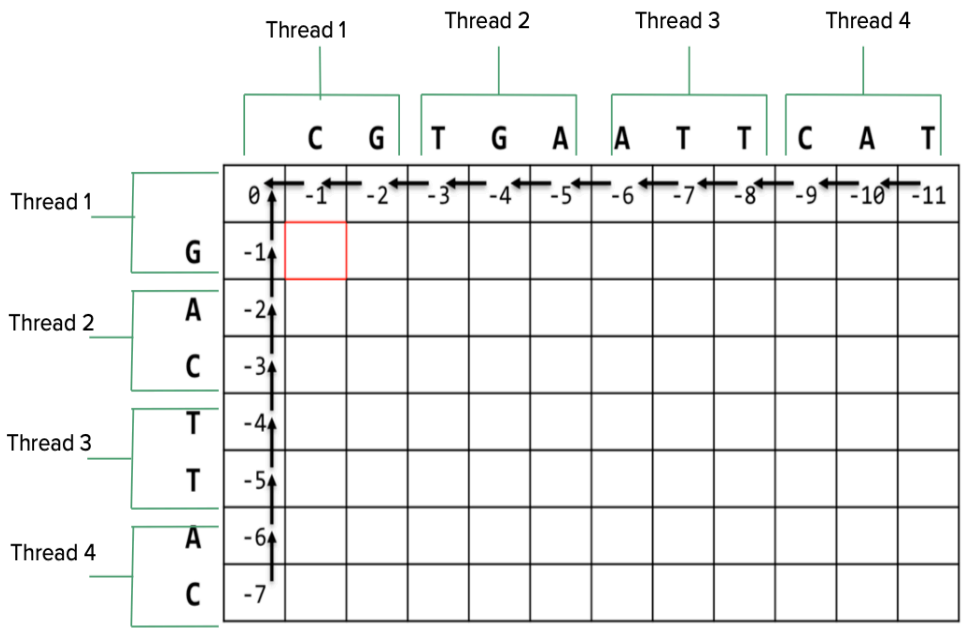
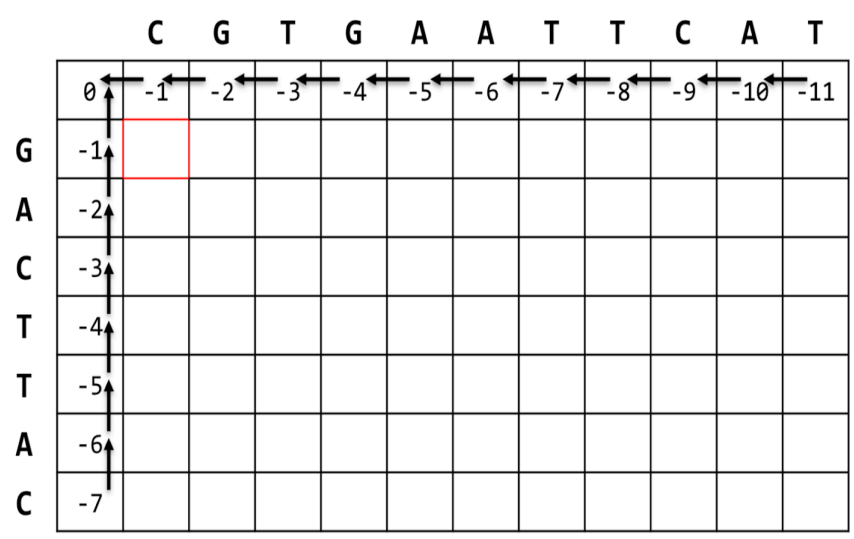


Figure (6.4)



Parallel Computation of Matrices

The serial implementation of the Needleman-wunsch algorithm traverses the matrix in the traditional fashion by visiting each in the first starting at index 0 then moving to the next row and so forth. The program utilizes a nested loop producing a time complexity of O(M*N). Figure(7.1) shows pseudo code of the serial implementation.

figure(7.1)

```
Do for (i = 0; I < Seq_A.length; i++){
      Do for (j = 0; j < Seq_B.length; j++){
            aboveLeft = C[i-1][j-1] + match(Seq_A[i],Seq_B[j])
            left = C[i][j-1] – 1
            above = C[i-1][j]
            C[i][j] = max(aboveLeft,left,above)
            If(max == aboveLeft) Pointer[i][j] = AL_pointer
            Else if(max == left) Pointer[i][j] = L_pointer
            Else Pointer[i][j] = A_pointer
      }
}
```

Computation at each cell has dependencies on three previously calculated neighboring cells. Also, every cell on the same row requires the cell preceding it to have its score ready. These dependencies make it difficult to parallelize the algorithm using traditional matrix traversal.

A possible solution is to traverse the matrix in a different fashion. Figure(7.2) showcase a relationship pivotal to the parallel solution formalized in this project. Looking at the empty cell in the bottom right corner, it is dependant on cell X and Y, which lie in a diagonal. Both X and Y are independent of each other and are dependant on values lying on the previous diagonal of the matrix. We hypothesized a solution which takes advantage of this relationship. It is possible to compute different cells in parallel as long as they are independent and have their dependencies satisfied. Therefore, cells lying on the same diagonal can be computed in parallel.

Figure(7.2)

		G	C
	0	-1	-2
G	-1	1	X
A	-2	Y	

An important step to utilize this relationship is to compute the cell indices lying on a diagonal while keeping track of the diagonals id in order to access these cells by different threads appropriately. Figure(7.3) showcases an algorithm derived to compute the diagonal cell indices. The first code block computes diagonals until the diagonal visiting the last element in the first row is reached. That diagonal represents and inflection point where the algorithm must proceed in a different manner in order to compute the remaining diagonals of the matrix, represented by the second code block. Although diagonal traversal is a common technique with simpler implementation, it is only effective when the matrix length and width are equal. Most sequences used as input in the algorithm are not of equal length and require matrices rectangular in shape requiring a more complex solution.

Figure(7.3)

```
for(int i = 1; i < mGridWidth; i++){
    int r = 0;
    int count = 0;
    for(int j = i; j >= 1; --j){
        r = r + 1;
        int c = j;
        if(r > mGridLength){
            break;
        }else{
            i_indices[count] = r;
            j_indices[count] = c;
            count++;
        }
    }
```

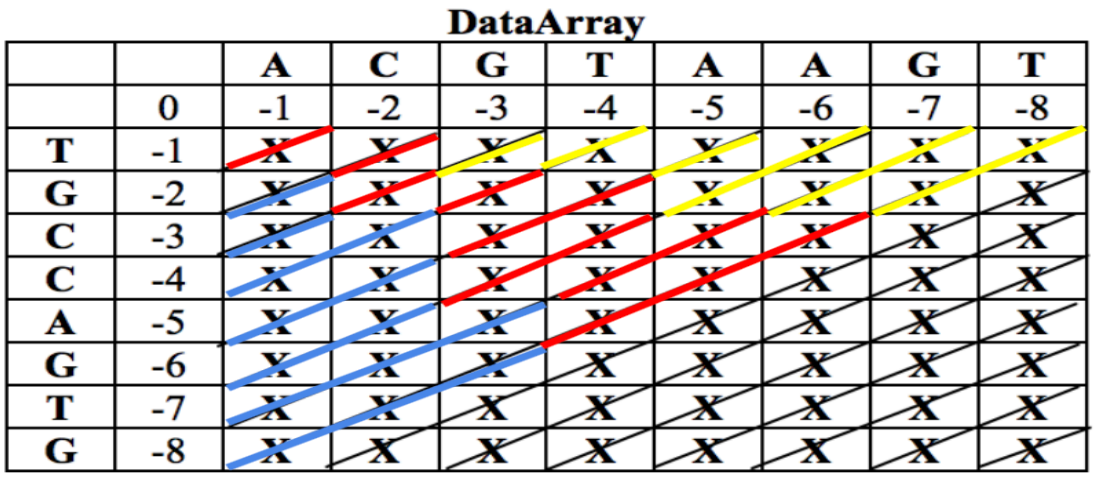
```
for(int k = 1; k <= mGridLength; k++){

    int j = mGridWidth;
    int count = 0;
    for(int i = k; i <= mGridLength; ++i){
        int r = i;
        int c = j;
        --j;
        i_indices[count] = r;
        j_indices[count] = c;
        ++count;
    }
```

Figure(8.2) showcases memory distribution used in the parallel algorithm where each colored line represents an individual thread running in parallel. Arrays containing diagonal indices and the score and pointer matrices are global variables shared by all threads. Diagonals are visited serially and during each iteration, pthreads are initialized to compute the values of the designated cells. Using blocked distribution, each thread receives a range of cells to compute which increases or decreases depending on the size of the diagonal. Once a diagonal is computed in parallel, all initialized threads are synchronized. Another approach is to eliminate synchronization in each serial threads such that threads will act independently throughout the entire processing step while utilizing mutex and conditional variables for locking individual cells and commencing computation. Although the second methodology shows promise, during implementation it did not show potential of speedup compared to the initial method. It is possible that a proper implementation of it could offer increased speedup. Figure(8.1) is pseudo code of the algorithm used in the processing step of the algorithm.

figure(8.1)

```
- Do for (l = 0; l < A; l++){
-     Do for (j = 0; j < B; j++){
-         i_indices[] = //i-indeces of diagonal cells for lth iteration
-         j_indices[] = //j-indeces of diagonal cells for lth iteration
-     }
-     Do for all threads(c){
-         For(k = (N/P)*c; k < (N/P)*(c+1); k++){
-         //compute score & pointer of cell[i_indices[k]][j_indices[k]]
-         }
-     }
      Barrier()
-
```

Figure(8.2)



Experimental Setup:

For the experiment, different sized sequence pairs, in FASTA format, were used as input. The files were downloaded from https://www.ncbi.nlm.nih.gov/. Each pair represents a different

test case. The test cases and their respective file sizes are as such:

- **Small Test:** 11 x 7 DNA characters
- **TAS2R16 Test:** 1,010 x 888 DNA characters
- **Ebola Test:** 19,231 x 19,161 DNA characters
- **Sapien Test:** 87,501 x 36,676 DNA characters

An output file was generated once the two sequences were processed that aligned the sequences optimally. The output files were compared to correct reference outputs. Other parameters were the number of threads, the processor speed (3.1 Ghz), the RAM size (8GB) and the number of cores used (4). The execution time for each tests and parallel approach was given 10 trials. The results below show the average execution times for these trials.

Results & Analysis - Parallel Initialization of Matrices

Figure (9.1)

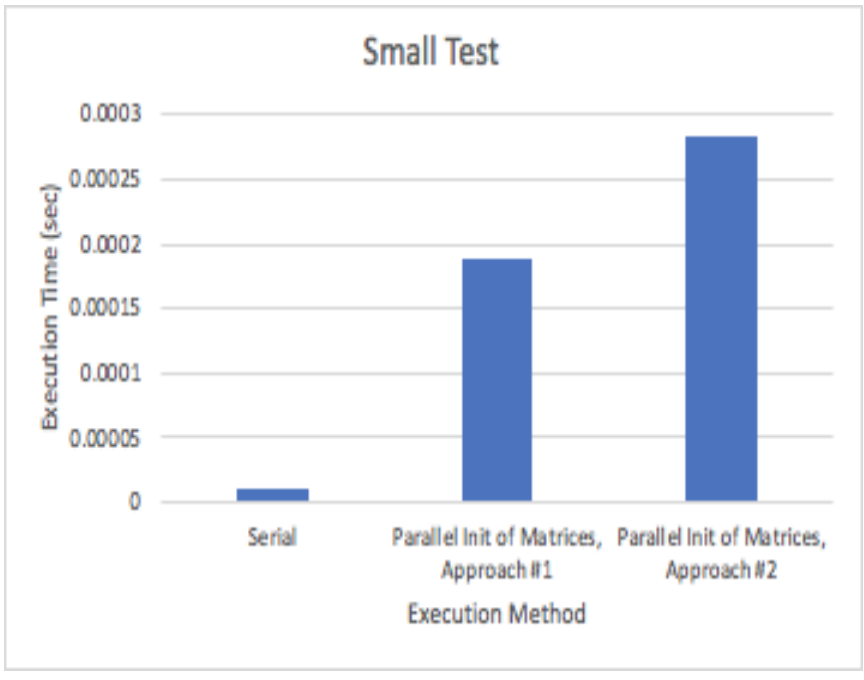


Figure (9.2)

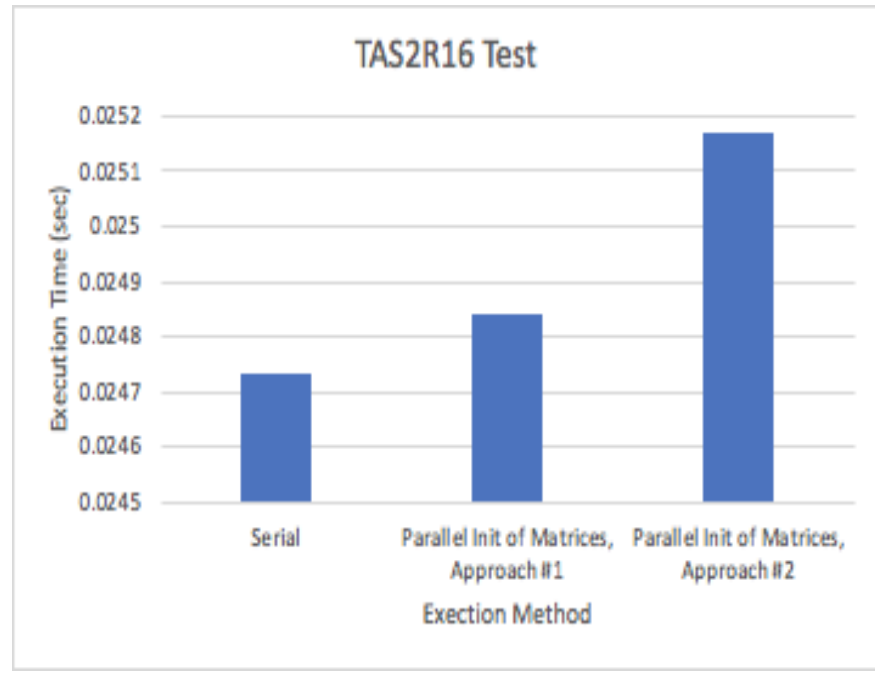
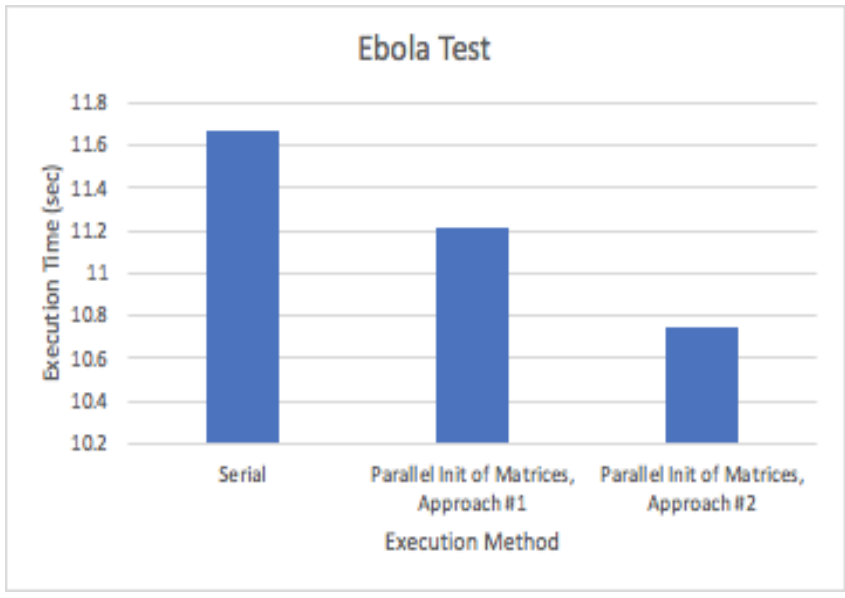
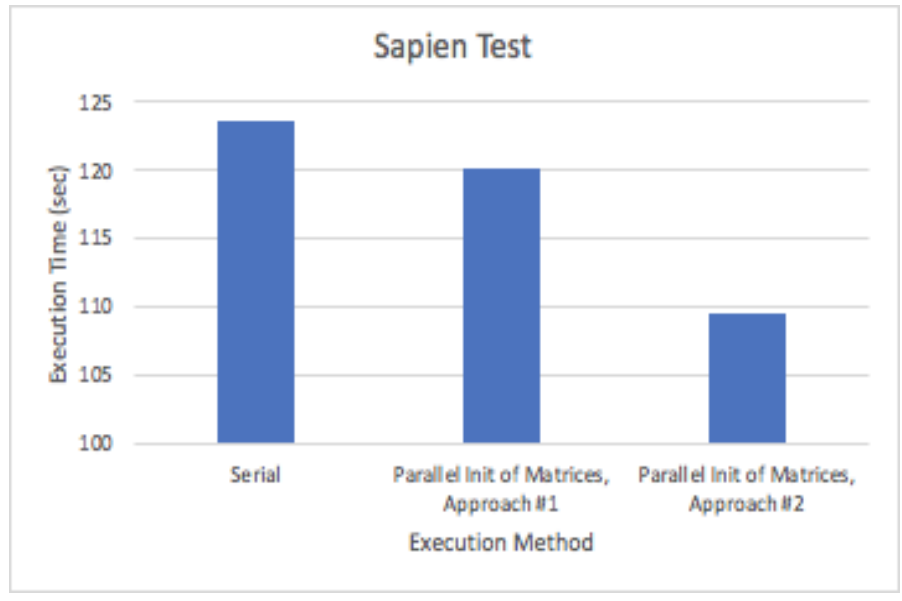


Figure (9.3)

Figure (9.4)

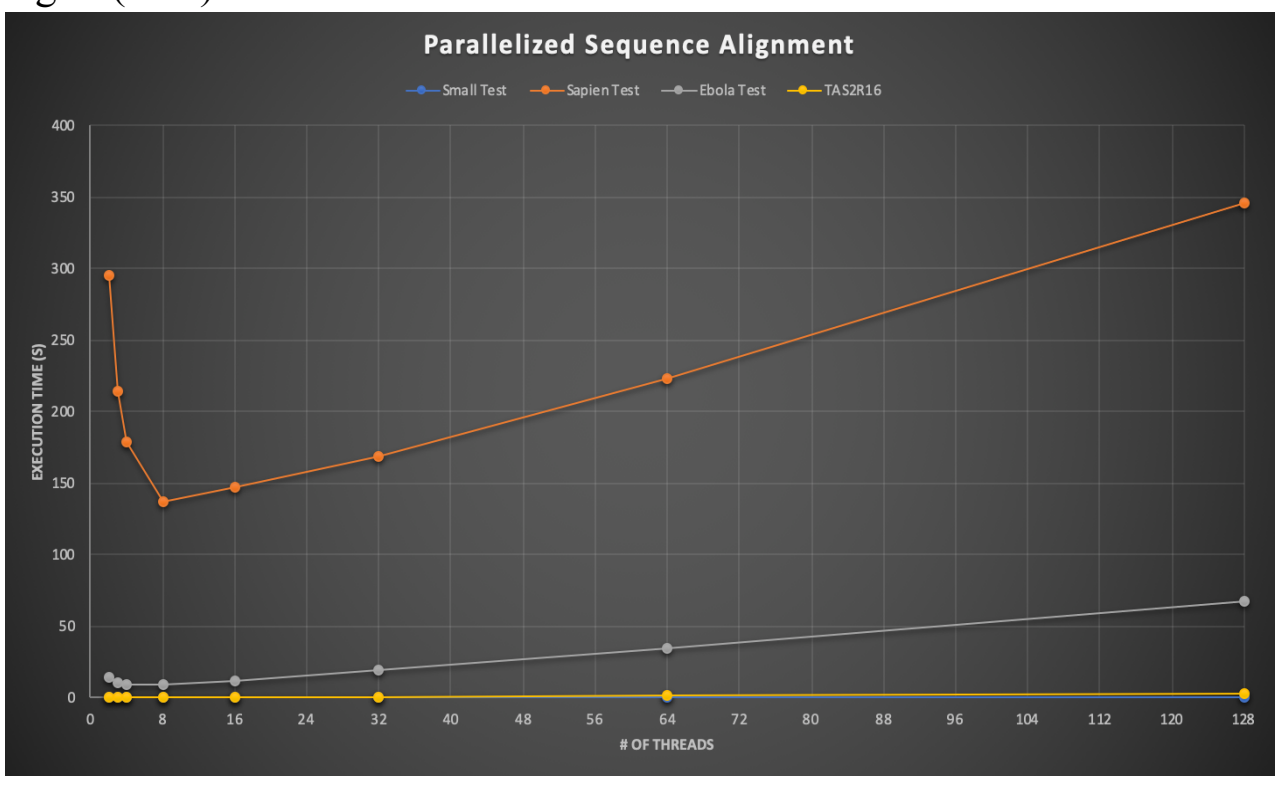




As shown by the figure above, the parallel initialization of the matrices did show an improvement in execution time but only for the tests with larger file sequences (Ebola and Sapien). With smaller sequences, the serial performed much better as expected. Also, the second approach was also significantly better than the first. In the largest sequence test (Sapien), the second approach saw a 1.13x speedup while the first approach only saw a 1.08x speedup from serial.

Results & Analysis - Parallel Computation of Matrices

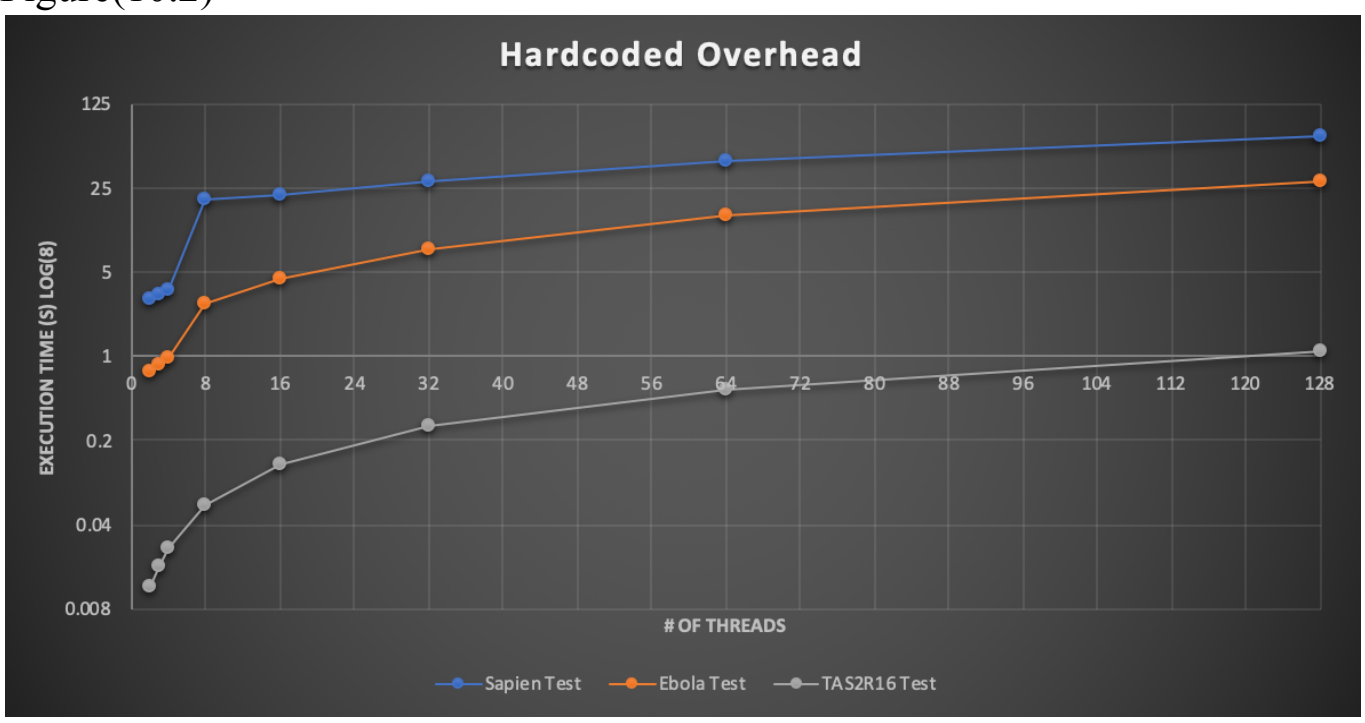
Figure(10.1)



Figure(10.1) shows results from running the Needleman-Wunsch algorithm with parallel computation using diagonal traversal and blocked distribution. The plot shows the total execution time using various numbers of threads. It is notable that there is a decrease in execution up to one half going from two threads to eight threads. After using eight threads the execution time increases linearly at a lesser rate with more added threads. This can be attributed to the hardware resources available on the device used for testing. The testing was conducted on

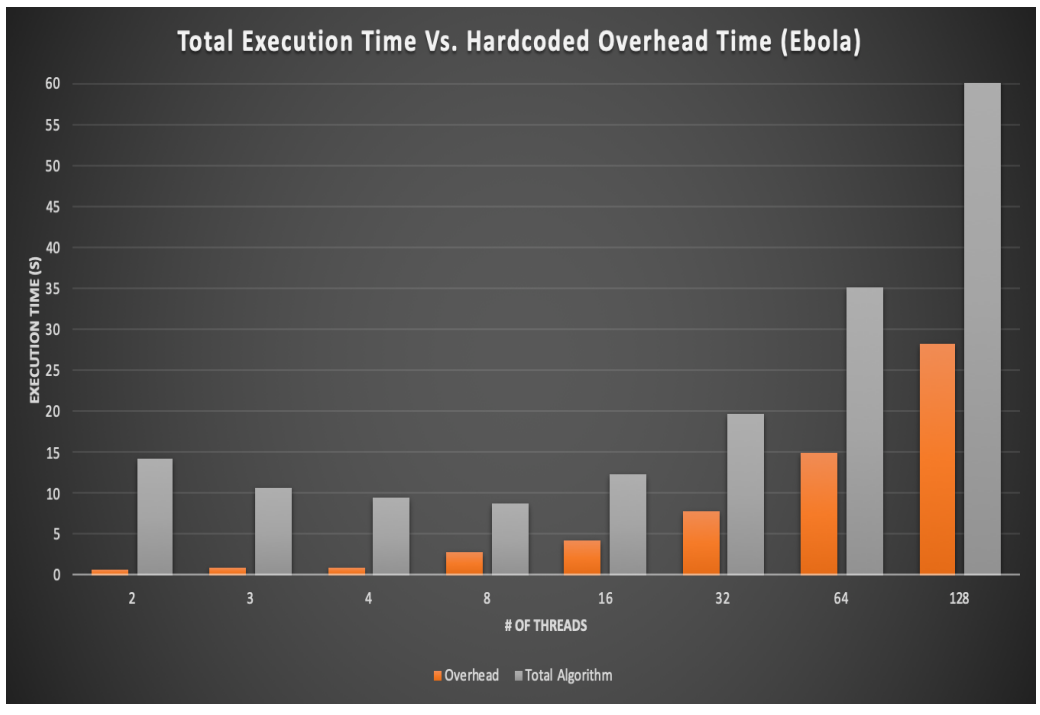
a Macbook Pro 2019 with four CPU cores and hyperthreading technology producing eight threads. This explains the improvement in performance and reaching the best execution at eight threads. When more threads were used in the program, there were no hardware resources to accommodate the extra threads. The excess threads were instead queued.

Figure(10.2)

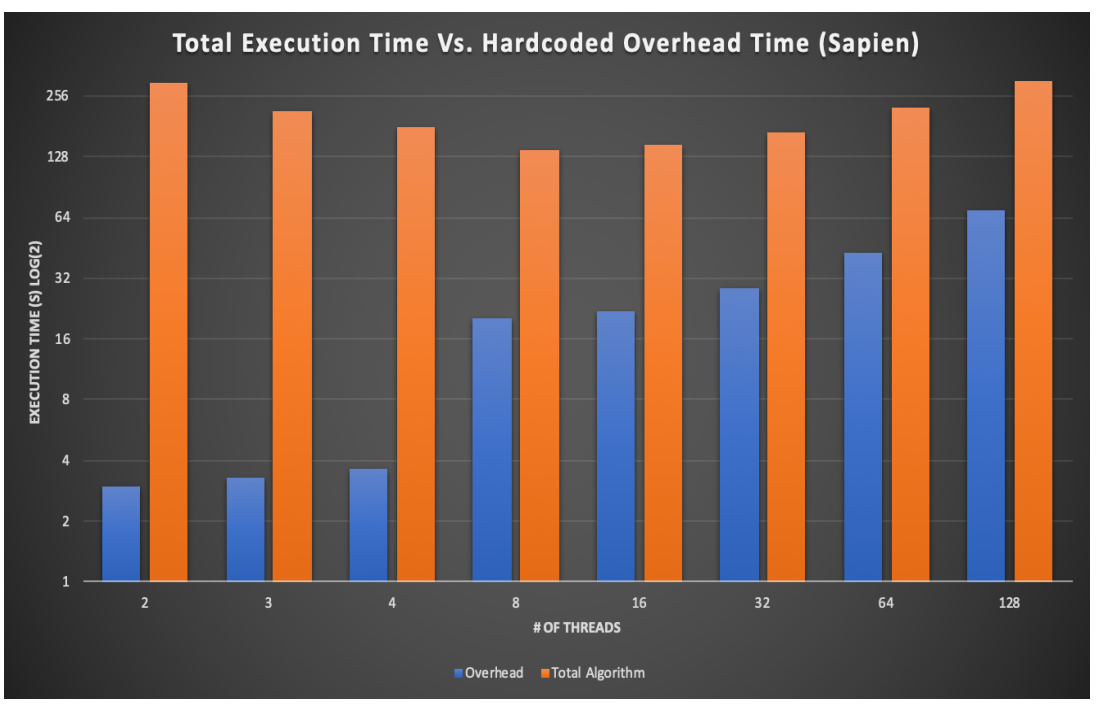


Figure(10.2) shows the execution time of thread initialization in addition to computing the diagonal indices and other variables necessary for the threads' operation. It is notable that the execution time of overhead increases with increasing number of threads due to the increased number of variables and iterations to accommodate the increasing number of threads.

Figure(10.3)



Figure(10.4)



Figure(10.3) and figure(10.4) show a comparison between total execution time of the algorithm and the execution time thread overhead and variable initialization required. Overhead and initialization requires increasing time with more thread. Between two and eight threads,

although initialization time is increasing, the total execution time decreased meaning the speedup from parallel computation compensated for the increased initialization time. However, as aforementioned the execution time past eight threads kept increasing. The speedup from parallel computing at eight threads did not change and the increase in total execution time is due to the increase in initialization time without compensation from the excess threads which were queued.

Conclusion:

The parallelization of the Needleman-Wunsch algorithm did show an improvement in execution time with large enough sequences but not as significant as expected. This was likely due to several limiting factors, the main one being the introduction of the diagonal finding algorithm in our Matrix Computation. This algorithm had to be performed serially and therefore produced a large overhead costs that increased with file size. Another significant limiting factor was the hardware used in the experiment. The processor used had only 8 thread execution. Our data showed that execution time kept decreasing when introducing more threads up to 8. However, when using any more that 8 threads, the execution time increased as overhead continued to increase exponentially. A better approach would be to utilize a larger core processor that can handle the overhead costs and therefore continue to lower execution time by increasing the thread counts. Hopefully future solutions can continue to optimize algorithms such as Needleman-Wunsch so that we can further the understanding of the human genome.

Appendix:

Github Repo: https://github.com/ebrahimAlhaddad/Parallel_Needle.git