CMPT431 Project Report

Introduction

This report discusses the parallel possibility of the “Longest Palindromic Subsequence” problem. A subsequence is a sequence that can be derived from another sequence by deleting some or no elements without changing the order of the remaining elements. This problem is to find the length of the longest subsequence which is palindromic.

To provide a better understanding, there are two examples:

Eg1.

If given input string s as "bbbab"
‘a’ can be deleted, so the remaining string becomes “bbbb”
Then the final result is the length of the “bbbb”, which is 4

Eg2.

Another given input string s’ is "cbbd"
After delete ‘c’ and ‘d’ from s’, the remaining part is “bb”
So the final result is 2 (the length of “bb”)

In this problem, we only take the ***alphanumeric*** (i.e. numbers and 26 English letters) as input, and letters are ***case-sensitive***.

“PinetreeCareGroup515Tower” is a valid input
“Pine tree Care Group 5.1.5 Tower” is NOT a valid input

The output will be an integer to indicate the longest length.

This project is a deep exploration of this problem. The algorithms of serial, multithreaded, and MPI will be discussed, implemented, and tested. The thoughts and insight will be discussed at the end.

Background

This Longest Palindromic Subsequence is a typical dynamic programming problem. If the implementation is in brute-force, the programmer needs to generate all possible subsequences of the string and check each one to see if it is a palindrome, keeping track of the longest palindrome found. Since there are 2^n possible subsequences of a string of length n, and checking if each one is a palindrome takes O(n) time, it will have a time complexity of O(n * 2^n).

This is a terrible time complexity for a program. So we use the Dynamic Programming strategy as the key to solve this problem and adapt the key idea to parallel versions.

Idea:

A simple way to solve this problem is to generate all the possible subsequences of the given string and then find the longest one that is a palindrome. Since there are 2^n possible subsequences for a string of length n, we can use recursion to do this.

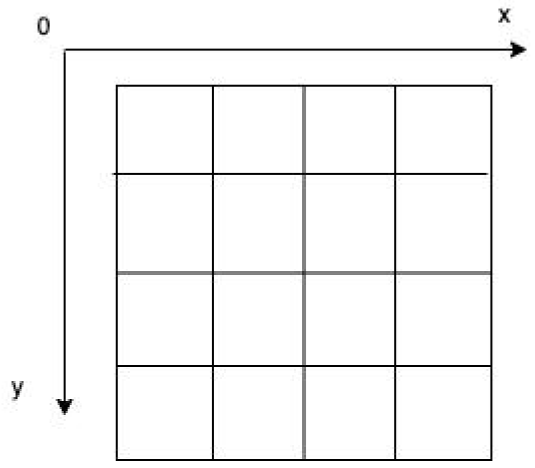
To break it down, if the first and last characters of a substring are the same, they are guaranteed to be part of the final palindrome. In this case, we add 2 to our result and then use recursion to work on the substring left after removing the first and last characters.

If the first and last characters are different, they can't both be in the palindrome. So, we use recursion twice—once after removing the first character, and again after removing the last character. Then, we take the maximum result of the two, since we're looking for the longest palindrome.

For this recursion, we use two pointers: i and j. Pointer i marks the start of the substring, and j marks the end. The recursive relation can be expressed like this:

If s[i] == s[j], we set the result = 2 + LPS(i + 1, j - 1).
If s[i] != s[j], we set result = max(LPS(i, j - 1), LPS(i + 1, j)).

If we build a 2-D table, as this:



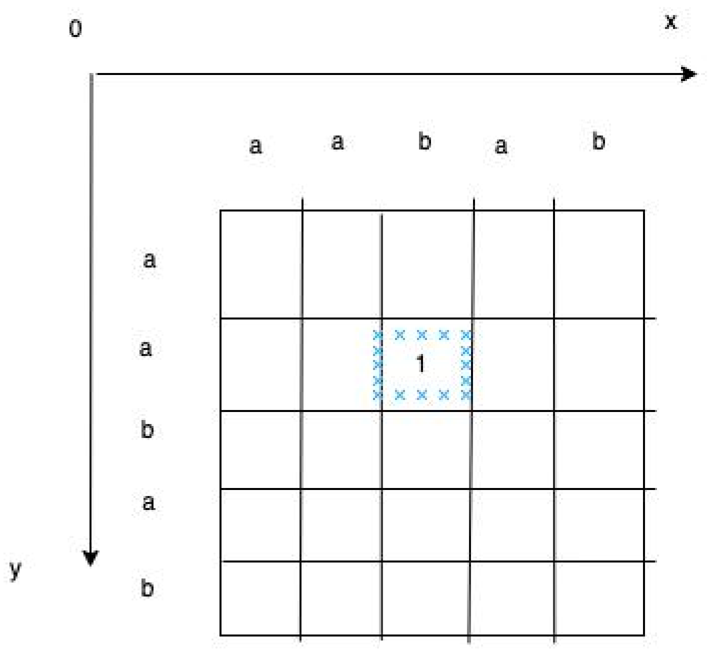
[1]

this set of equation can be adapted to:

If s[x] == s[y], table[y][x]= 2 + table[y+1][x-1].
If s[i] != s[j], table[i][j] = max(table[y][x-1], table[y+1][x]).

Following the new equation set, we can fill the table in the bottom-up version.
For a string: aabab, we start the checking from any character. For now, let's start from the first 'b', which has an index of 2.

string:	aabab
index:	01234

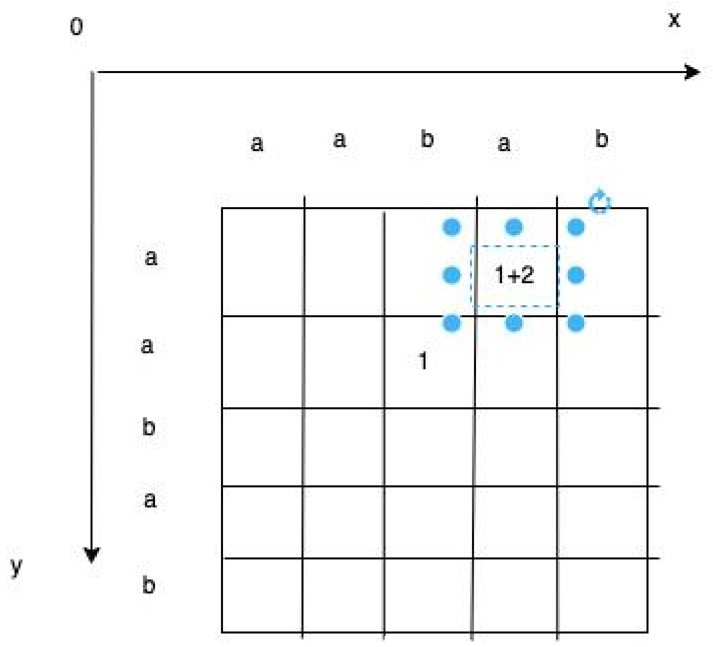


[2]

Let's put two pointers y and x at the corresponding axis, and only consider the string between these two pointers.

Now, we have the longest palindromic string "b" with a length of 1. We record length = 1. Next, we move y up by one and move x to the right by one.

```
string:        aabab
index:         01234
```



[3]

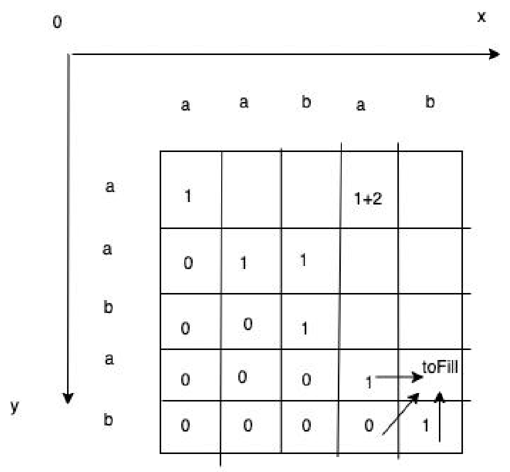
We check if the values pointed by y and x are the same. Luckily, this time they are the same. Now we have a new string "aba" based on the original string "b" and it has two ends with the same character "a", so our current longest palindromic string has a length by length + 2 = 3.

Similarly, the operation can be done for all other cells. For table[y][x] where y = x, the value will always be 1.

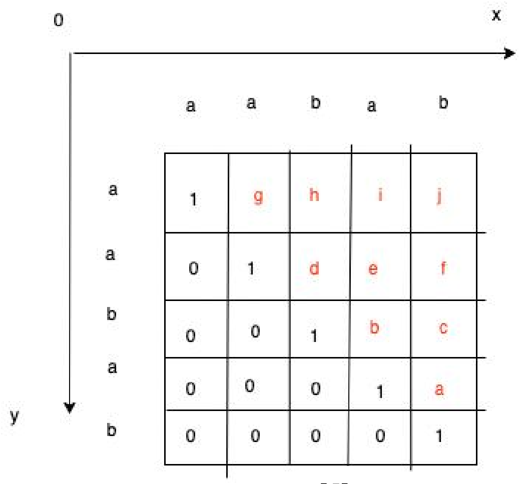
(Note, for table[y][x] where y > x, the semantic meaning is that a string with its left bound is greater than its right bound. Then it has no length, so we fill it as 0)

After all the cell is filled, the ***right-up corner is the answer***.

Since each cell needs three dependencies: its left neighbor, bot neighbor, and left-bottom neighbor, a serial implementation is to fill the table bottom up and from left to right. In this route, all the dependencies will be computed before filling the current cell.



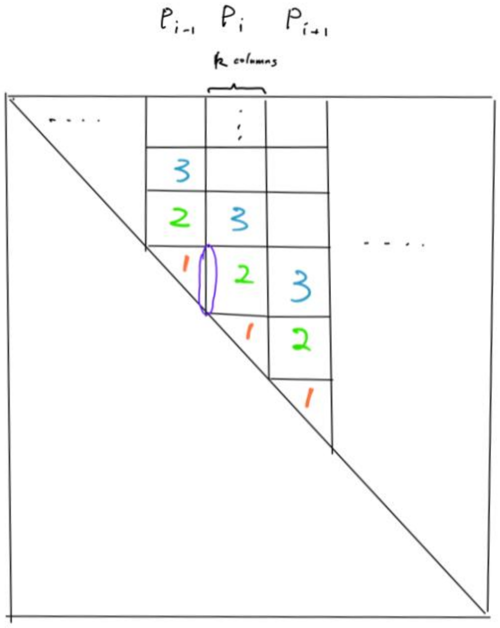
[4]



[5]

The red text in the alphabet represents the order of filling the table (from a to z).

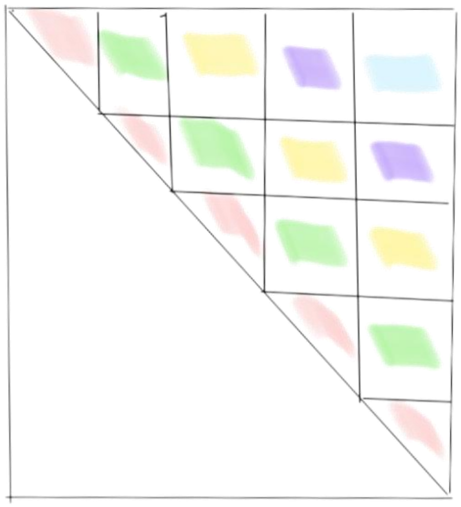
Parallel idea:



[6]

Each process is assigned with k columns, and each of them will start from the triangle area shown in fig. The same coloured area can be calculated in parallel by each process. After an area is done, a process can use the data on its left edge as dependencies to calculate the next area. Our multi-thread and MPI methods will be based on this approach.

Possible speedup:



[7]

We can see from the above graph([7]), assuming the time to calculate one square area needs time t, a serial program needs (1+2+3+4+2.5) t = 12.5t to finish the computation. If we use 5 processes/threads to do the calculation and ignore the communication time, we only need (1+1+1+1+0.5)t = 4.5t to finish the computation. We got 2.8 speed up if used 5 processes.

If we have an N process, the max possible speedup is (1+n)n/2n = n/2. n is the number of processes.

Implementation Details

Serial

Since the algorithm has been discussed in the background section. This section will quickly go through some key sections of the serial implementation.

1. We created a class DPTable, which has key members like:
    a. table: a 2D vector of int to store the DP table
    b. read(): read the value from the DP table, from a particular position
    c. assign(): fill specific value to a specific position in the table
2. We implemented the key function fillTable(), this function will finish the final table by using the given input string as a reference. It uses the logic as discussed in the background section: If s[x] == s[y], table[y][x]= 2 + table[y+1][x-1].
   If s[i] != s[j], table[i][j] = max(table[y][x-1], table[y+1][x]).

The whole process is straightforward: parse the input, build the DP table based on input, fill the table, and return the right top cell value as the final result.

Multi thread

The program achieves parallelism by dividing the dynamic programming (DP) table into column ranges, assigning each range to a separate thread. Each thread is responsible for calculating values for its assigned columns while ensuring correctness through synchronization. Threads operate independently but occasionally need to synchronize when their calculations depend on values computed by neighboring threads in adjacent columns or rows.
Communication between threads is achieved implicitly through the shared DP table, which is accessible to all threads. Dependencies between columns, such as values from the left or bottom-left cells, are handled using busy-waiting. A thread checks whether the required values are ready and yields execution (std::this_thread::yield()) if they are not, allowing other threads to proceed. This approach avoids explicit locks or condition variables but ensures data consistency by waiting for dependencies to be resolved.
Each thread iterates over its assigned range of columns and processes rows from bottom to top. It calculates the DP value for each cell based on its dependencies: For diagonal cells (base case), the value is straightforward. For other cells, the thread checks whether the left and bottom dependencies are ready before proceeding. The computations rely on simple comparisons and maximum operations, and results are written back to the shared DP table. By carefully assigning column ranges and managing dependencies, each thread can compute its values independently for most of the computation, minimizing contention and ensuring correctness.

MPI

Since the algorithm idea has been discussed in the background section and serial section. This section will discuss the details of:

1. split the workload
2. Communication with neighbor process

To split the work we evenly assign the same number of continuous columns to the same process. For extra k ones, assign them to the first kth process.

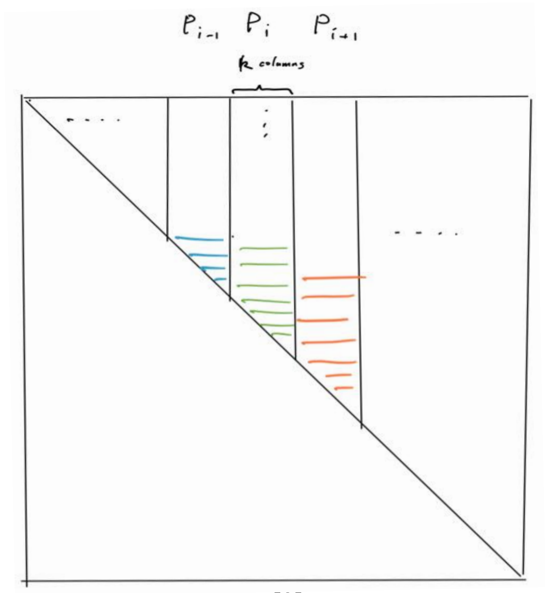
```
uint subCols = T->sLength / worldSize;
uint extraCols = T->sLength % worldSize;

 if(worldRank < extraCols){  // --- assign extra column to first extraCols-th process ---//
         startCol = (subCols + 1) * worldRank;
        endCol = startCol + subCols;
}
 else{
         startCol = (subCols + 1) * extraCols + (worldRank - extraCols) * subCols;
         endCol = startCol + subCols - 1;
 }
```

To communicate with neighbors, we use MPI_Send() and MPI_Recv().

1. Each process will finish its own area - either a triangle or a rectangle
2. When finished its area, process the edge data to be sent
3. then depend on its rank and current turn, even p sends first then receives in even turn, odd p receives first then send
4. In an odd turn, odd p sends first then receive, even p receives first then send

This rule will avoid the deadlock during communication and safely let the early-finished process quit. In addition, it limited the number of communication operations to achieve a better speed.

After implementation, there is a new found: the test phase shows this communication method is slower than expected. So we tried a new method to communicate as the graph shown below:



[8]

In the graph([8]), each color line is a row. Every process will:

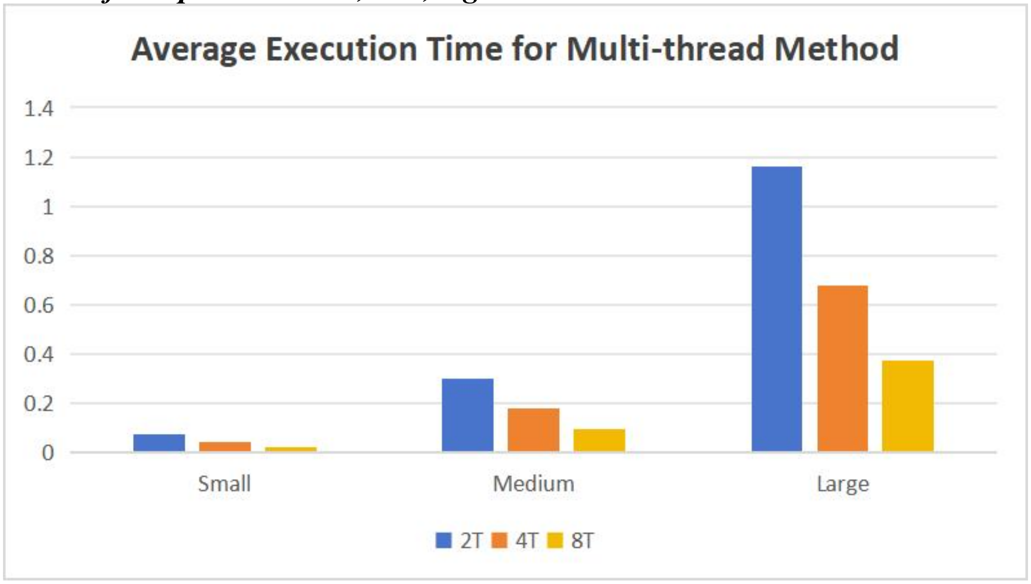
1. finish one of the rows first and then wait the data from the left neighbor
2. and the send its rightmost cell data to the right neighbor

For every row, it will do a MPI_Send and MPI_receive. The communication operation is increased a lot than the method before, however, the total processing time is 30% faster than before.

The reason may be due to the large non-continuous data exchange in the memory being slower than a single exchange. The large data may not be stored in the memory at the same time. It will frequently be exchanged. That may be the reason for slowing down the processing time.

Evaluation

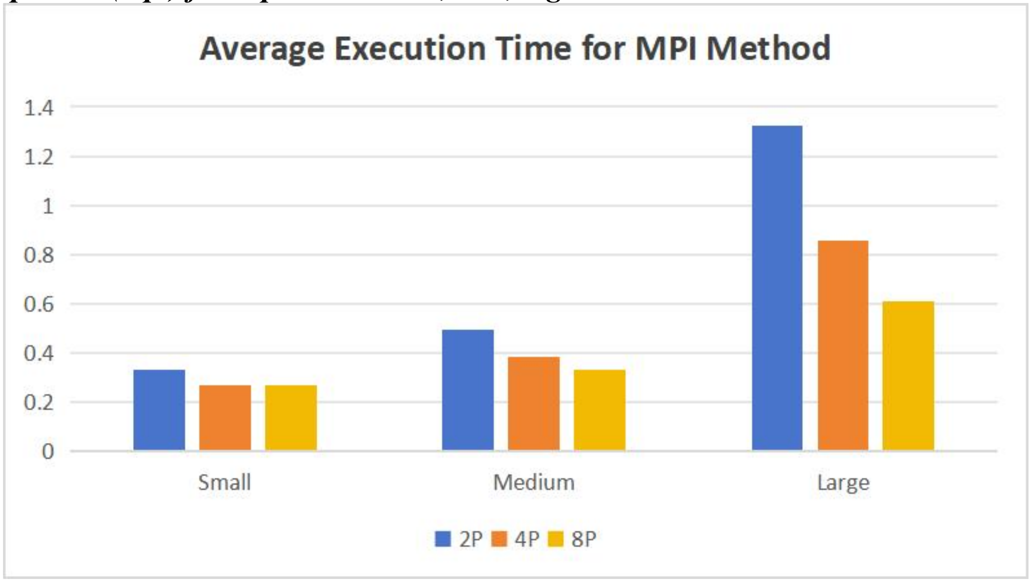
1. 2, 4, 8 thread for input size small, mid, big



[9]

This chart([9]) illustrates the average execution time of a multi-threaded method for small, medium, and large input sizes using 2, 4, and 8 threads. For small inputs, the execution times are similar across all thread counts, indicating limited benefit from parallelism due to low computational intensity. As the input size increases to medium, the execution time decreases slightly with more threads, showing that parallelism starts to improve performance. For large inputs, the execution time significantly decreases as the number of threads increases, demonstrating effective workload distribution and scalability. This highlights the efficiency of the multi-threaded approach for larger inputs while suggesting limited advantages for smaller ones.

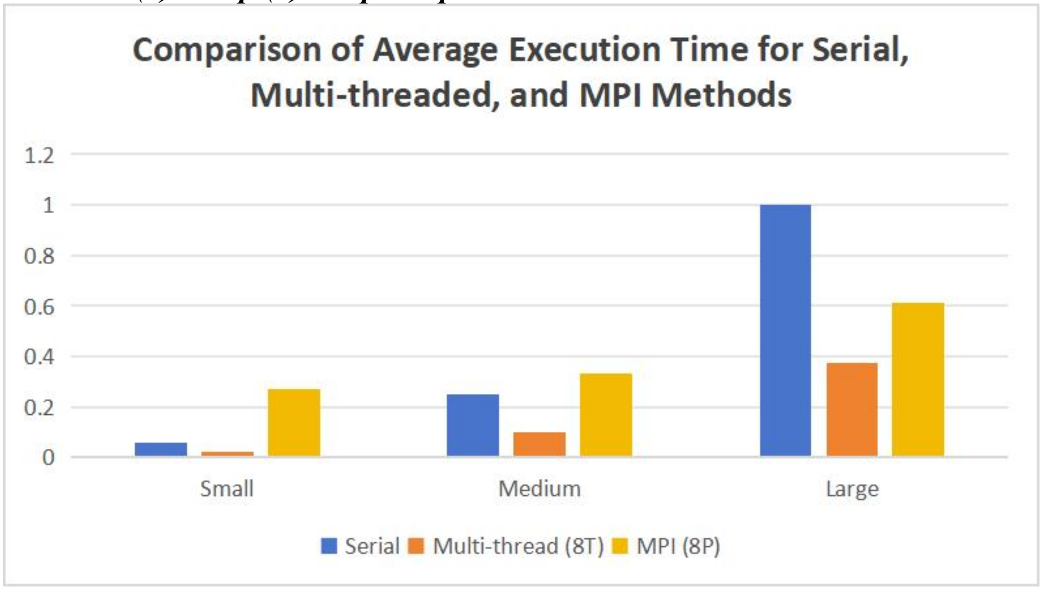
2. 2, 4, 8 process(mpi) for input size small, mid, big



[10]

This chart([10]) compares the average execution time of the MPI method for small, medium, and large input sizes using 2, 4, and 8 processes. For small inputs, the execution time is relatively consistent across different process counts, indicating that the overhead of inter-process communication is significant compared to the computation. For medium inputs, the execution time shows a noticeable decrease as the number of processes increases, reflecting improved efficiency. For large inputs, the differences between 2, 4, and 8 processes are more pronounced, with 8 processes achieving significantly better performance. This indicates that the MPI method scales well with larger inputs but has limited advantages for smaller datasets.

3. Serial vs thread(8) vs mpi(8) → speedup



[11]

This chart([11]) compares the average execution times of serial, multi-threaded (8T), and MPI (8P) methods across small, medium, and large input sizes. For small inputs, MPI is notably slower than both the serial and multi-threaded methods, likely due to the communication overhead outweighing the benefits of parallelism. For medium inputs, the execution time ranks as multi-threaded < serial < MPI, with the multi-threaded method being the fastest, showcasing its effectiveness for medium-sized workloads. For large inputs, the differences between the methods are significant. The serial method has the highest execution time, while both parallel methods are much faster. The multi-threaded method exhibits a slight advantage over MPI, likely due to the reduced overhead of inter-process communication compared to MPI's distributed memory approach. The 8 threads parallel has 2.5 speedup and the 8 process MPI parallel has 1.67 speedup.

Conclusion

This report first provided a detailed explanation and analysis of the algorithm for solving the Longest Palindromic Subsequence problem, illustrating why it works and how it guarantees correct results through dynamic programming. The parallelization strategy was explained, focusing on how dependencies are managed to ensure correctness during parallel execution. Detailed implementations for the serial, multi-threaded, and MPI-based methods were then presented. Finally, performance evaluations were conducted using execution time graphs, demonstrating that multi-threading achieves the best performance for medium and large inputs due to efficient resource utilization, while MPI excels in scalability for large datasets despite its communication overhead.

From this project, we learned how to design and implement parallel solutions for intensive problems. We gained hands-on experience with multi-threading and MPI, understanding their strengths and limitations. Additionally, analyzing performance metrics provided valuable insights into how parallelism can be optimized for different input sizes and environments, reinforcing the importance of balancing computation and communication overhead. The 8 threads parallel has 2.5 speedup and the 8 process MPI parallel has 1.67 speedup.