Performance Analysis Report for MPI Based Decryption

Lucas Sudol
Sebastian Kula

Table of Contents

Correctness Analysis

*Compiler optimizations were not used in compiling this assignment, as for repeat tests with the same iteration sizes, the compiler would optimize the algorithm for computing permutations. This would drastically change the timings of the program, which would make it impossible to analyze the results of changes to program execution timings.

Testing Correctness of MPI Application

Test Cases: Checking serial results with varying unique character lengths, with 5 Processes.

Word Bank: "cat", "code", "apple", "garden", "freedom", "computer", "attention", "impression"

This test goes through each word in the word bank, encrypts it, and then compares the results of decryption from both serial and multi-process programs. After passing the string through the encryption algorithm, the program will decrypt it using the serial implementation, and MPI implementation with 5 processes. It will compare the results, and check if all decryptions are found in both implementations. The increasing unique character count is used to test task distribution among parallel processes, as the program must ensure all possible permutations are calculated. It then goes over the number of processes in the pool to check if processes are assigned multiple permutations.

To Run Test Case

All test cases can be run by executing the correctnessTest.py program in the root directory of the repository, ensuring it is built with “make all”. It will run all the appropriate tests with increasing unique character length, and print the results of the test to stdout. Sample results are shown on the next page.

Test Passed

Analysis

With the results of the prior test cases, it can be concluded that the results of the multi-process decryption algorithm are correct. Since the processes in the program simply use the serial algorithm and function calls, errors can only result from faults in the message passing or any failures in task distribution. Using different task sizes with thread numbers tested the messaging system, and the corresponding distribution of permutation tasks. These tests all confirmed that the decryptions of the varying string sizes were correct, and all processes were receiving the correct branches of permutations to be calculated. Since the calculations are dependent on the same algorithm used in the serial version, no further testing was required to validate the parallelization of the tasks.

Results

```
Starting tests...
Testing word: cat
Serial Output (filtered):
cat
act
MPI Output (filtered):
cat
act
Test passed for word 'cat': Outputs match.


Testing word: code
Serial Output (filtered):
coed
code
MPI Output (filtered):
coed
code
Test passed for word 'code': Outputs match.

Testing word: apple
Serial Output (filtered):
apple
MPI Output (filtered):
apple
Test passed for word 'apple': Outputs match.

Testing word: garden
Serial Output (filtered):
ranged
danger
garden
gander
MPI Output (filtered):
ranged
danger
garden
gander
Test passed for word 'garden': Outputs match.

Testing word: freedom
Serial Output (filtered):
freedom
MPI Output (filtered):
freedom
Test passed for word 'freedom': Outputs match.
```

```
Testing word: computer
Serial Output (filtered):
computer
MPI Output (filtered):
computer
Test passed for word 'computer': Outputs match.

Testing word: attention
Serial Output (filtered):
attention
MPI Output (filtered):
attention
Test passed for word 'attention': Outputs match.

Testing word: impression
Serial Output (filtered):
impression
MPI Output (filtered):
impression
Test passed for word 'impression': Outputs match.

Testing complete.
```

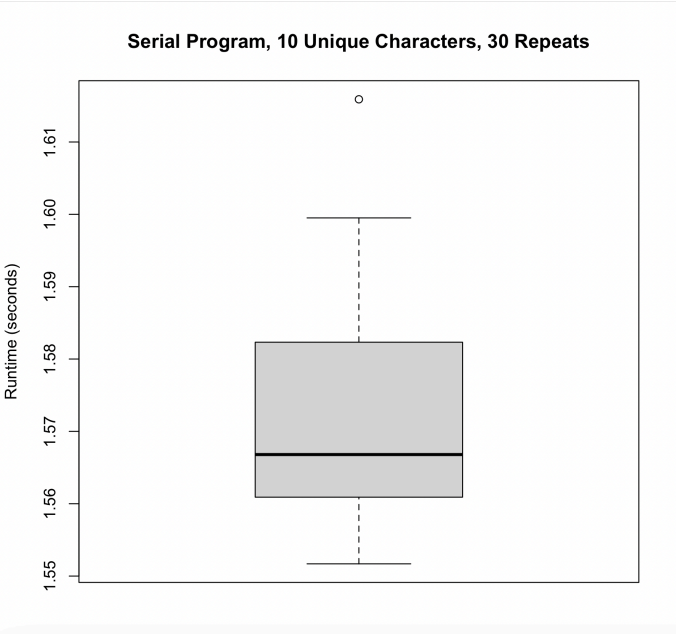
Performance Analysis

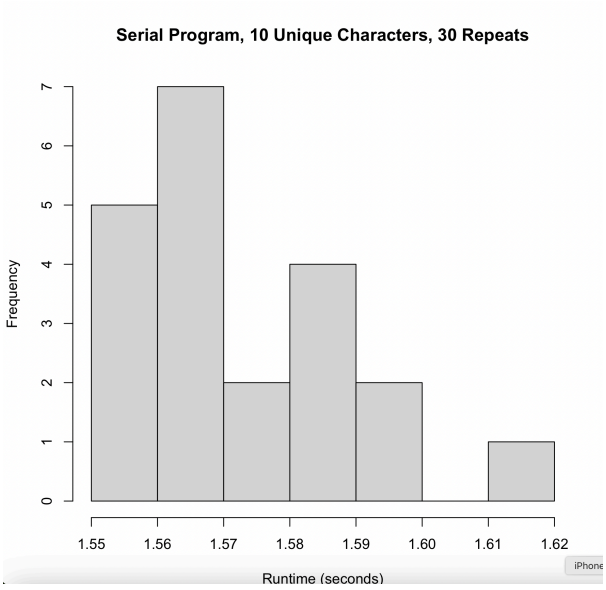
Computer Processor Information

Processor: AMD Ryzen™ 7 5800X
Core Count: 8

Each test was performed using 30 repetitions, and then the median of the sample was taken as a result. This is due to the fact that the data was skewed, leading to the median being a better representation of the data, as mean results are affected by outliers.

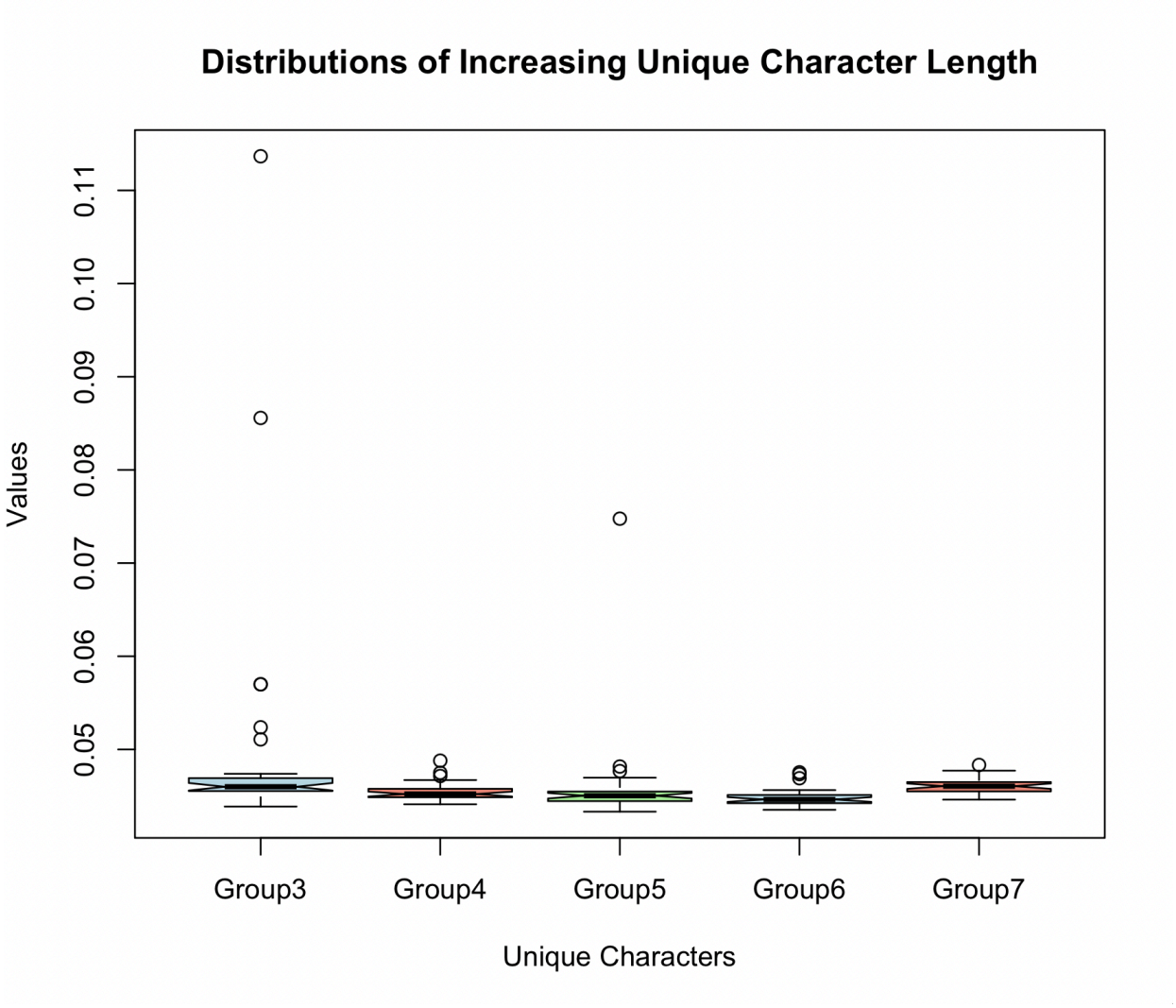
Example: Variation in Serial Program Sample





It is important to consider skewness when timing programs, as it results from there only being factors that can "slow" the program down, not speed it up. This leads to timings being for the most part near their maximum, and outliers when things start to slow down the CPU scheduler. These considerations are taken into account when using the median instead of the mean, as its result is not as skewed due to the presence of outliers, making it a more accurate representation of the data.

Analyzing Distributions with <u>Increasing Unique Character Count</u>



When analyzing the distributions of the data sets of a single process application with increasing unique character counts, it is apparent that the dataset is almost always skewed. This is once again a result of program iterations being fixed to permutation length, and having variation in timings as a result of the CPU scheduler; causing this skewed distribution.

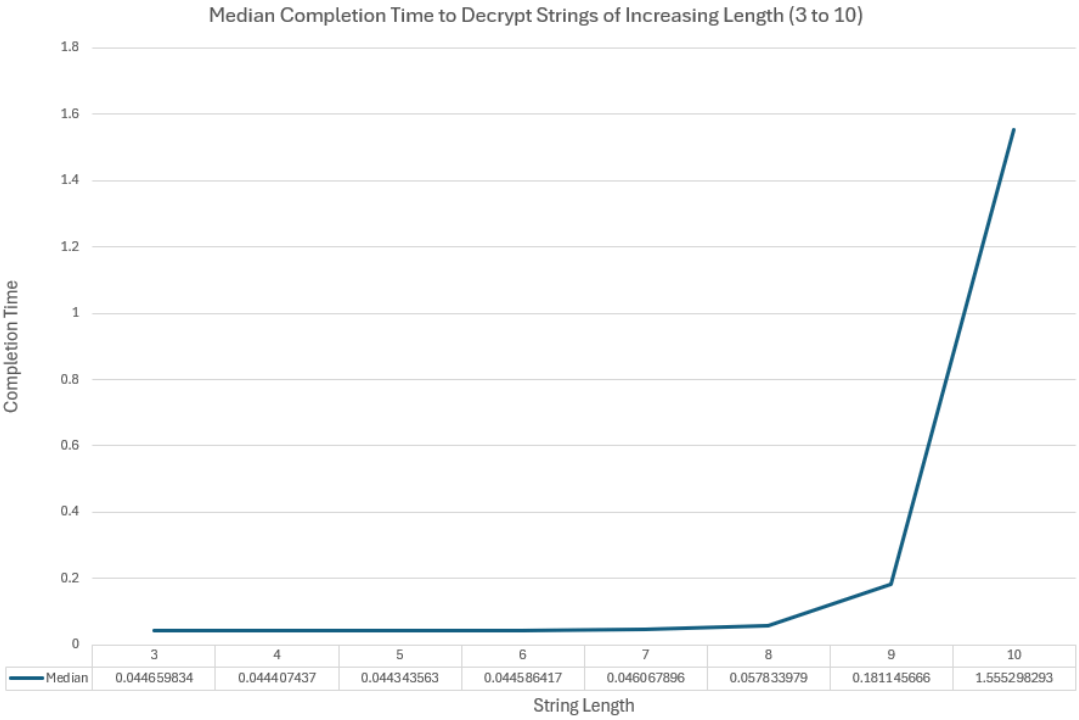
Analyzing Performance in Serial Program with <u>Increasing Unique Character Count</u>

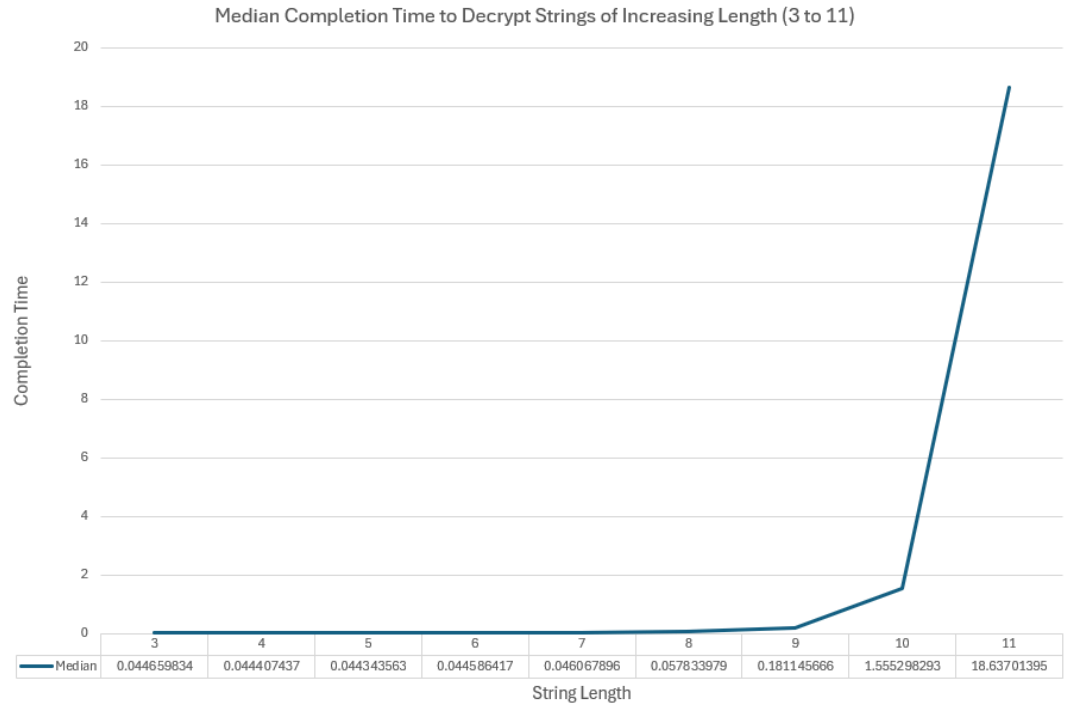
Case 1: One Process with Character Counts from 3 to 11

Unique Character Count: 3,4,5,6,7,8,9,10,11

“Abc”, “abcd”... to “abcdefghijk”

Repeats = 30, Plot median results





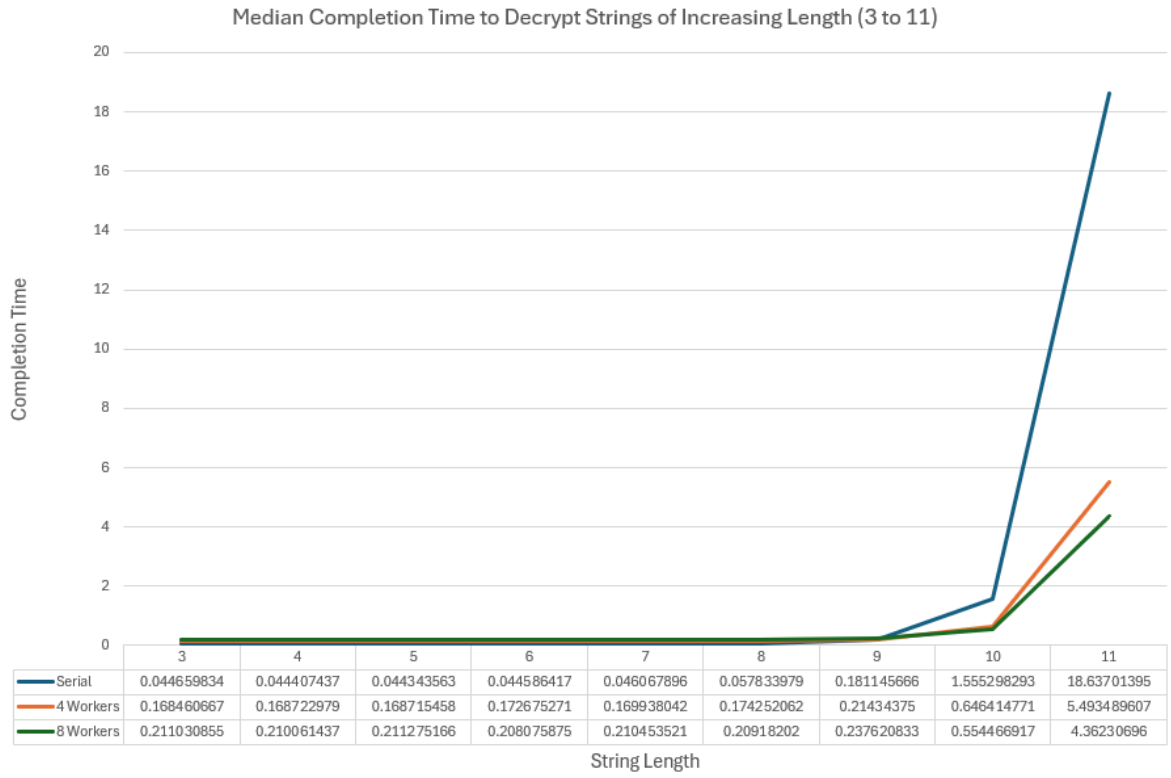
Case 1 Analysis

This test was used to analyze the time complexity of the overall algorithm as unique character count increases. Increasing amounts of unique characters were passed to the serial program to analyze its effect on run time. Since this program uses brute force to calculate all possible permutations, it follows a factorial time complexity. As shown in the graphs above, the execution time drastically increases past 10 unique string characters. As a result of this runtime complexity, unique character counts past 11 become too time-consuming to analyze since their runtimes increase by factorial orders of magnitude. The median time for 11 unique characters is 18.6 seconds, multiplied by 30 repetitions results in a test time of 9.6 minutes.

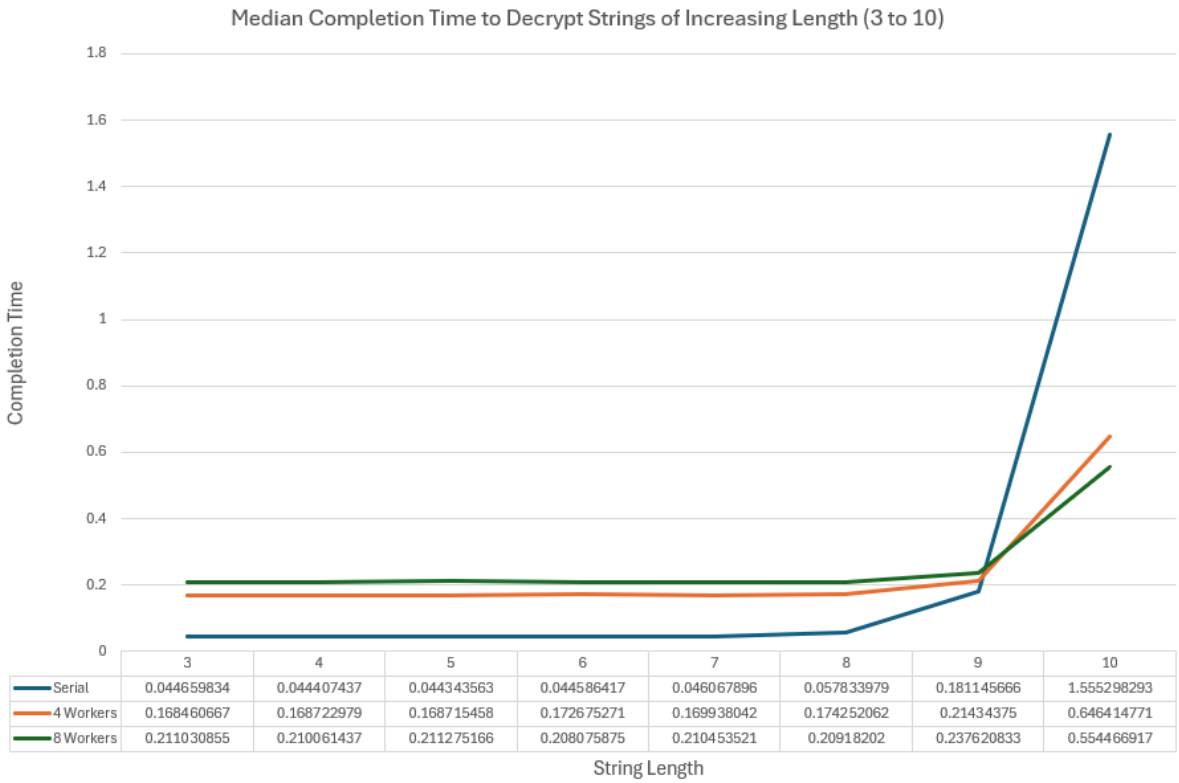
Since the program calculates all possible permutations based on unique characters, the data distributions follow these skewed distributions. This would be different if the program exited after finding the result, as its completion time would be dependent on where the correct permutation falls in the computation. This would lead to a more uniform distribution, affecting the distribution of program timings.

Analyzing Performance in MPI Program with <u>Increasing Unique Character Count</u>
Case 2: Character Counts from 3 to 11, with 1, 4 and 8 processes

Unique Character Count: 3,4,5,6,7,8,9,10,11
"Abc", "abcd"... to "abcdefghijk"
Process Count: 1,4,8
Repeats = 30, Plot median results



	3	4	5	6	7	8	9	10	11
Serial	0.044659834	0.044407437	0.044343563	0.044586417	0.046067896	0.057833979	0.181145666	1.555298293	18.63701395
4 Workers	0.168460667	0.168722979	0.168715458	0.172675271	0.169938042	0.174252062	0.21434375	0.646414771	5.493489607
8 Workers	0.211030855	0.210061437	0.211275166	0.208075875	0.210453521	0.20918202	0.237620833	0.554466917	4.36230696

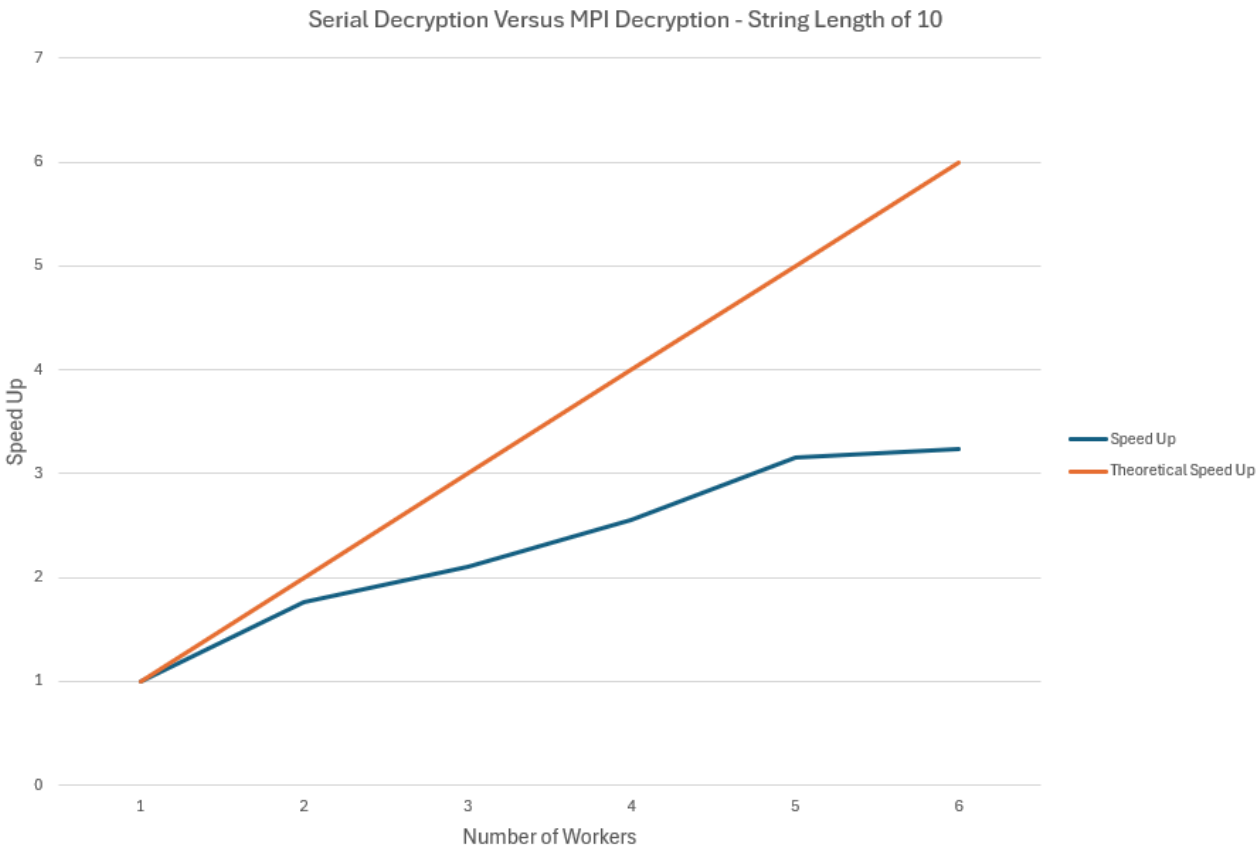


	3	4	5	6	7	8	9	10
Serial	0.044659834	0.044407437	0.044343563	0.044586417	0.046067896	0.057833979	0.181145666	1.555298293
4 Workers	0.168460667	0.168722979	0.168715458	0.172675271	0.169938042	0.174252062	0.21434375	0.646414771
8 Workers	0.211030855	0.210061437	0.211275166	0.208075875	0.210453521	0.20918202	0.237620833	0.554466917

Case 2 Analysis:

This experiment examines the impact of string length on decryption performance between serial and parallel approaches using MPI with 4 and 8 cores. The graph shows that for strings with 3 to 9 unique characters, the serial implementation consistently outperforms the parallel MPI implementations, as the overhead of managing multiple processes outweighs any parallelization benefits. Both the 4-core and 8-core versions perform similarly within this range, remaining slower than the serial version. However, when the string length reaches 10 unique characters, the serial completion time spikes dramatically due to the factorial time complexity of the decryption process, while the parallel versions increase more gradually, with 8 cores providing a slight improvement over 4. This shift indicates that MPI parallelization becomes beneficial only when the workload is complex enough to offset its inherent overhead.

Analyzing Speedup in <u>Increasing Process Count</u>
Case 3: 10 Unique Characters, 1 to 6 Processes
Tasks = 1,2,3,4..100 (Iterations evenly distributed to tasks)
Thread Count = 16
Repeats = 30, Plot median results



Case 3 Analysis:

This experiment was conducted to determine the performance impact of increasing the number of MPI workers on the decryption of a string with 10 unique characters. The graph compares the actual speedup achieved with the theoretical speedup, showing that performance gains do not scale linearly as the number of workers increases. While the theoretical line suggests an ideal linear speedup, the actual speedup plateaus after about 4 workers, with minimal improvement beyond that point. This could be due to the fact that this test only utilized strings with 10 unique characters, if more were used then the plateau would likely occur at a larger number of workers. The test did not include running 7 or 8 workers because it forced the system to utilize its efficiency cores, which skewed the results by providing significantly lower performance. This indicates that the overhead of MPI communication and process management grows as more workers are added, meaning certain tasks may not always benefit from utilizing additional cores. The diminishing returns after 4 workers highlight the impact of MPI overhead, as increased coordination among processes restricts further gains in speedup.

Performance Analysis Conclusions

Overall, these results show both the benefits and limitations of using MPI parallelization for decryption tasks. While MPI can offer solid speedup compared to the serial version, it doesn't scale perfectly because of the overhead involved with managing multiple processes. For smaller strings with fewer unique characters, the serial approach often runs faster since the MPI overhead such as communication and management negates any parallelization gains. However, as the task complexity increases (as seen with 10 or more unique characters), the workload finally becomes large enough to make MPI worthwhile, and we start seeing the advantages of parallel processing. Even so, the speedup will level off after a certain number of workers (as in case 2, with 9 unique characters having 4 processes was faster than 8 processes), as the added cost of managing more processes reduces the gains from additional cores. This experiment highlights that MPI is most effective when the task complexity justifies its overhead, and it’s essential to find a balance between the number of processes and the workload to get the best performance.