Bilkent University

Fall 2023-2024

CS342 Operating Systems

Project 1

Concurrent Processes, IPC, and Threads

Section 1

Deniz Tuna Onguner 22001788

Alper Göçmen 22002948

Oct 20, 2023

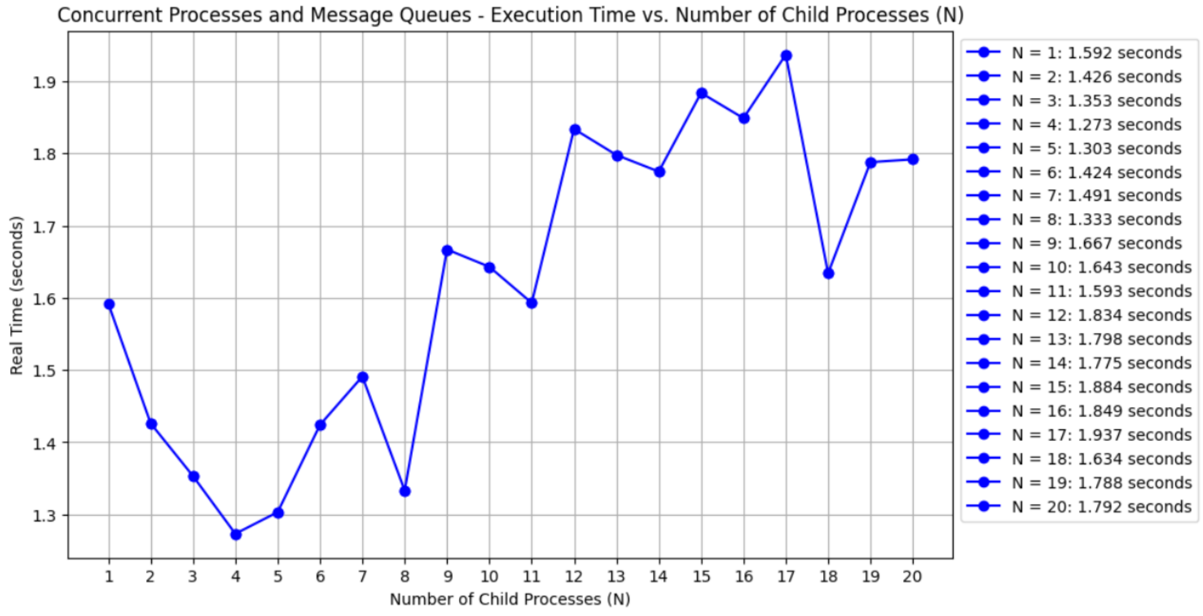
Introduction

The objective of this report is to document and compare two different programming approaches for an application that finds the prime numbers in a given text file. Thus, we have developed two separate programs; one is implemented via multiprocessing, while the other uses multiple threads. In the first part, we focused on multiprocessing and tried to understand how the program's speed changed as the number of child processes changed. Also, in this part, we changed the number of prime numbers sent to the message queue at a time aimed to observe whether the program's total runtime depended on the message size. In the second part, we used threads to achieve the same thing. At this time, we tried to understand whether the program's speed depends on the number of threads working. The following sections will discuss these results and the programs' speed.

Part A: Concurrent Processes and Message Queues

1- Different N Values

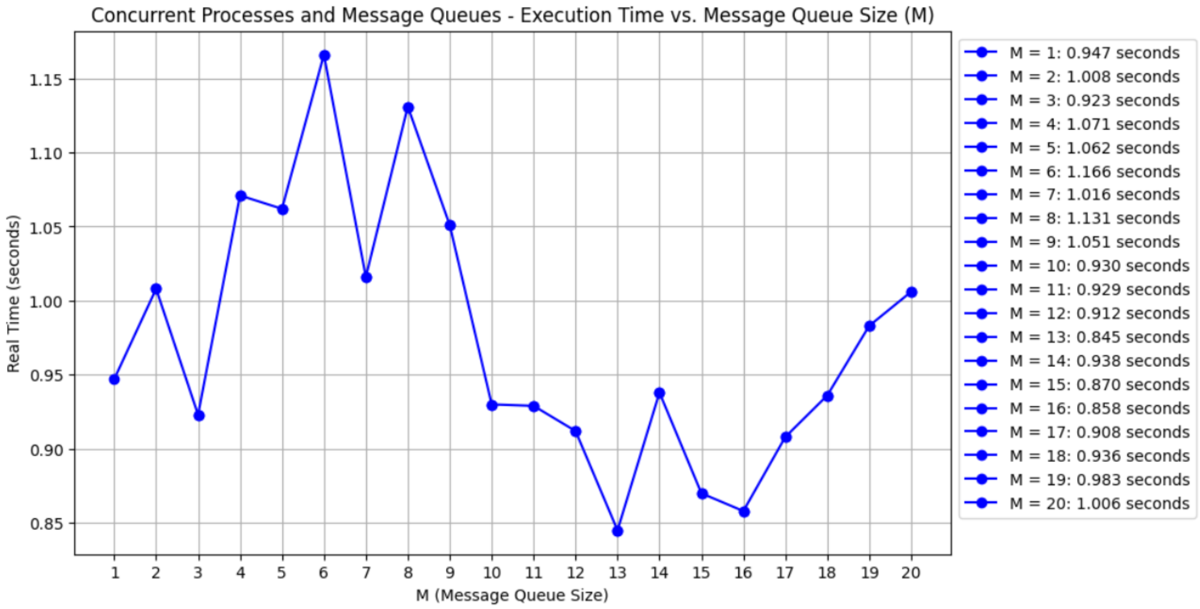
In utilizing multiple child processes for the prime number finding application, we notice that the overall time it takes to complete the task improves as we increase the number of child processes. In simpler terms, having more processes helps the program run faster until a certain point. For instance, with only a single process (N=1), the real-time execution takes approximately 1.592 seconds, but real-time steadily decreases as more processes are used, reaching its lowest point at N=4 with 1.273 seconds. However, beyond N=4, there is a slight rise in execution time, suggesting potential overhead associated with managing numerous processes. Thus, it is essential to consider resource limitations while working with the child processes since, as we increase the number of child processes, there is a greater demand for system resources such as CPU, memory, and I/O. Beyond a certain point, these resources can become scarce. Thus, excessive child processes may make the program run slower, even worsening it, like in N=17. Overall, these results indicate that using multiple child processes can substantially boost the performance of prime number finding application up to a point where diminishing returns or slight performance degradation might occur due to the management of a large number of processes.



Note: input.txt includes 500.000 numbers in a range from 1 to 100.000.000.

2- Different M Values

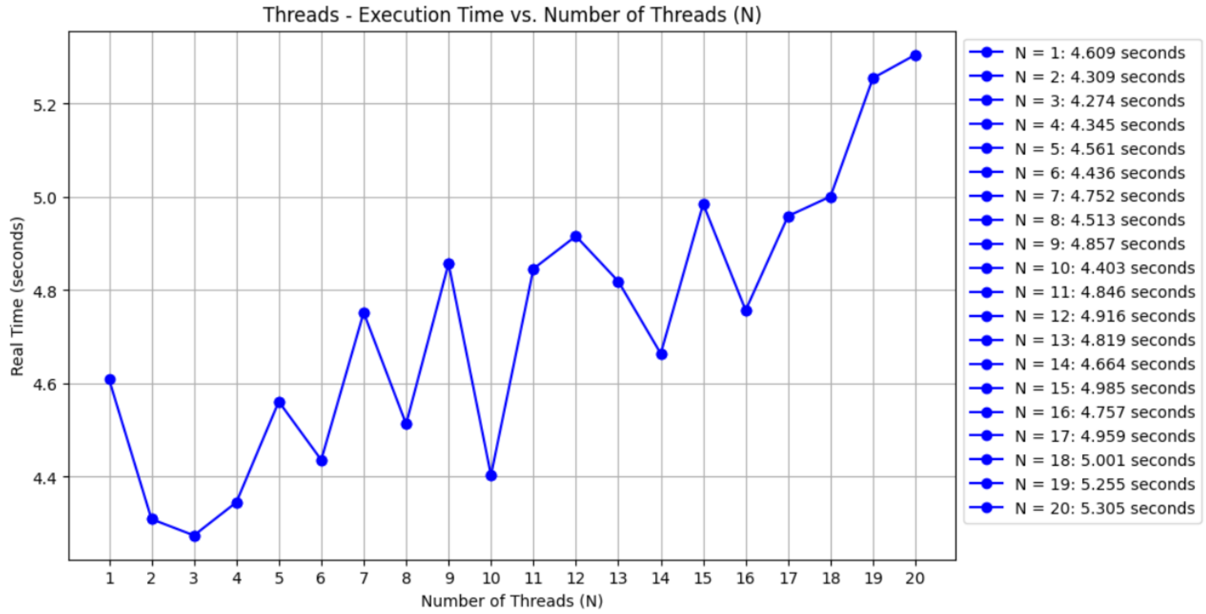
We explained how N changes the program’s speed in the previous section. Now, the N will be fixed as 5 (N=5) for all runs, but the M will be changed, which is the number of primes sent in one message. The results show that when M is set to 1, the real-time execution is remarkably fast, taking only 0.947 seconds. However, as M increases, the execution time rises, with the slowest run recorded for M=7 at 1.166 seconds. Interestingly, there are instances where a smaller M value, such as M=14, results in faster execution, taking only 0.845 seconds. This shows that finding the right balance between the number of prime numbers in a message and the message queue size can make a difference in how well the program works. Before the experiment, we thought that the program’s speed should not depend on the M value since the total job done will be the same, even if M is small or not. For instance, with the M=2 value, there will be 50 messages; with the M=20 value, there will be just five larger messages. In the process, the total data sent will be the same. However, we were wrong since the larger (M>9) values make the program faster than the smaller values (M < 8). The overhead associated with processing multiple smaller messages may be higher than processing a single larger message. Thus, with the larger M values, there are fewer messages to manage, which could reduce this overhead.



Note: input.txt includes 500,000 numbers in a range from 1 to 100,000,000.

Part B: Threads

In evaluating the prime number finding application using multiple threads, it is essential to note the initial performance improvement achieved by moving from a single thread to multiple threads until five or six threads. However, the graph shows that the performance decreases as the number of threads (N) increases. As observed, the execution time for processing a sequence of positive integers grows from an initial value of around 4.609 seconds for a single thread (N=1) to 5.305 seconds for threads (N=20). The unexpected increase in execution time with more threads suggests that too many threads can make the program slower, even worsening it. Considering the resource limitations and overhead associated with managing multiple threads is essential, as an excessive number of threads may only sometimes lead to improved processing speed. We thought that it had the same reason as the child processes. As we increase the number of threads, there will be a higher demand for system resources after a certain point. Also, when multiple threads work in parallel, they must synchronize their work to avoid conflicts and ensure data consistency. For instance, our program has a global linked list, so every thread waits for the other threads to enter this critical section. This synchronization can introduce overhead, which becomes more pronounced as the number of threads increases. Overall, until one point, using multiple threads is beneficial. However, using excessive multiple threads makes the program run slower after that point.



Note: input.txt includes 2,500 numbers in a range from 1 to 10,000.

Conclusion

Our project gives us insight into two crucial aspects: using child processes and message queues (Part A) and evaluating threads (Part B) to improve the prime number finding application.

In Part A, we witnessed the impact of varying the number of child processes (N) and the number of primes packed into a single message (M) on application performance. While N initially offered performance gains, there was a tipping point beyond which diminishing returns and overhead set in. The unexpected dependence of program speed on M highlighted the significance of message size and message queue management.

On the other hand, while using threads initially improved performance in Part B, we observed a decline in performance as the number of threads increased. This behavior was primarily influenced by resource contention and synchronization requirements.

Comparing the two parts, Part A proved to be more efficient in enhancing the program's performance. As you can see in the notes below the graphs, very high numbers are used in Part A, while significantly small numbers are used in Part B. There is a massive difference between the programs in terms of program speed. The main reason could be that child processes work simultaneously in Part A, whereas the threads wait for the other threads in part B.

Overall, these findings teach us that finding the right balance between a number of processes or threads and how they are managed is crucial for making our programs work as fast as possible.