COMP 8005

Assignment 1 Report

By: Derek Wong

Instructor: Aman Abdulla

Due: 12 PM on January 25th, 2020

Table of Contents

Introduction

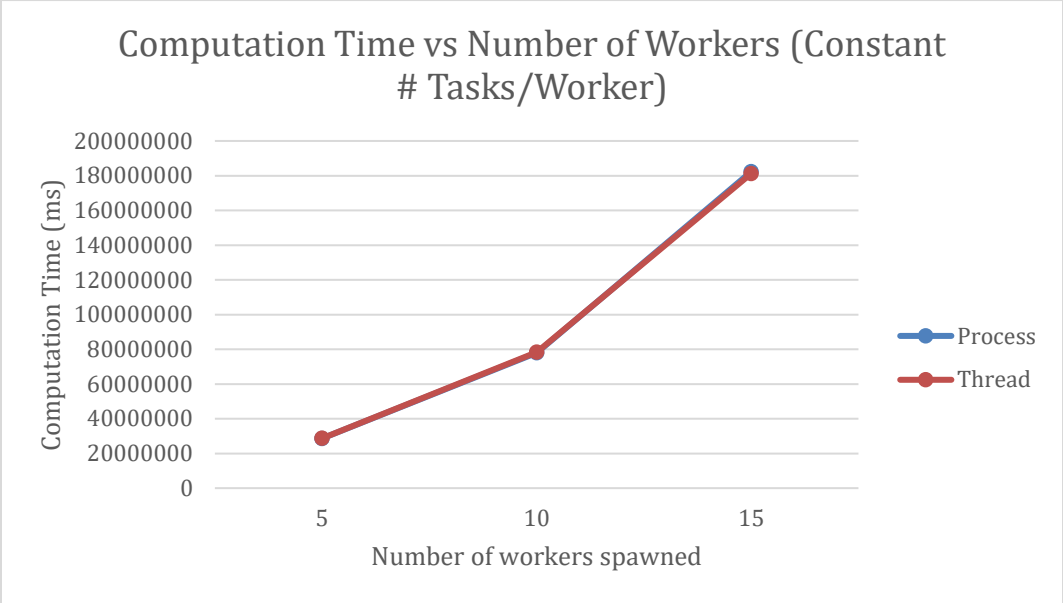
The main goal of this assignment is to obtain a measure of efficiency and performance of a system-intensive mathematical operation involving file input and output between threads versus processes. I gathered the computation time it took to do the same number of prime number factorization, varying both the number of workers and tasks done by each worker. Using the computation time as a measure of overall performance, we can see if there is a difference between the two on a Linux system.

Test Cases and Results

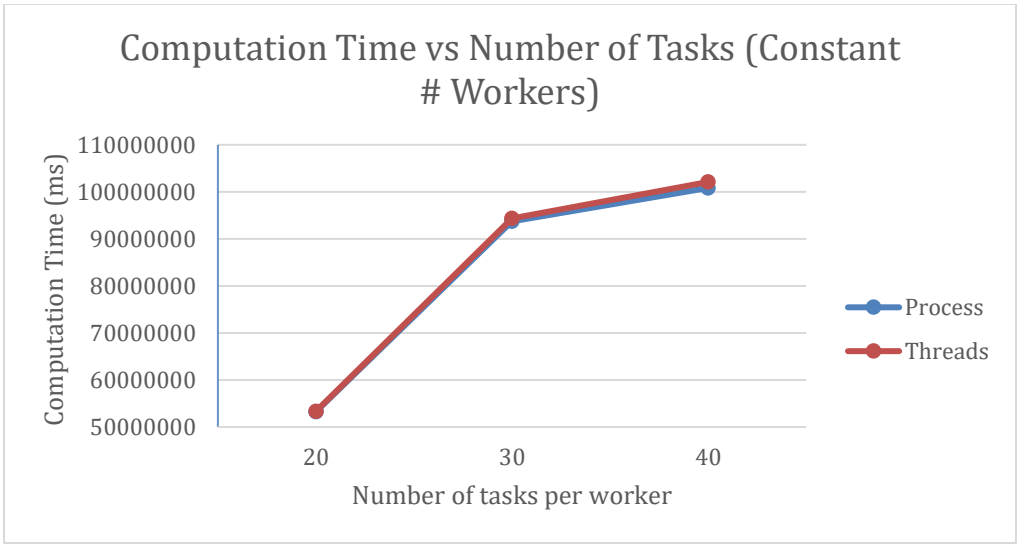
Test Case	Number of workers	Task/Worker	Number range to factorize	Total Elapsed Time (ms)
Process				
1	5	10	50,000,000 ~ 50,000,049	28642599
2	10	10	50,000,000 ~ 50,000,099	77927717
3	15	10	50,000,000 ~ 50,000,149	182337639
4	5	20	50,000,000 ~ 50,000,099	53272027
5	5	30	50,000,000 ~ 50,000,149	93716898
6	5	40	50,000,000 ~ 50,000,199	100834079
Thread				
1	5	10	50,000,000 ~ 50,000,049	28697517
2	10	10	50,000,000 ~ 50,000,099	78331794
3	15	10	50,000,000 ~ 50,000,149	181240044
4	5	20	50,000,000 ~ 50,000,099	53361133
5	5	30	50,000,000 ~ 50,000,149	94364259
6	5	40	50,000,000 ~ 50,000,199	102097029

Table 1: Corresponding test cases for both process and threads, each mirroring a set of test conditions

Analysis



There does not seem to be a significant visual difference between the total elapsed time between computations using process versus threads when the number of tasks per worker remains constant. If we the trend continues, we can assume that increasing the number of workers will exponentially require greater computation times. Based on the cumulative elapsed time for both threads and processes, we can see that the performance is significantly better when we keep the number of workers close to the number of cores on the CPU (4 in the case of the Raspberry Pi 3B+).



Similarly, to the previous diagram, there does not seem to be a significant visual difference between the total elapsed time between computations using process versus threads when the number of workers is kept constant. If the trend continues, we can assume predict that the computation time will remain relatively constant and will plateau out.

Test Case	Total Elapsed Time (ms) for Process	Total Elapsed Time (ms) for Threads	Computation Time Ratio $(\frac{\text{Process Total Elapsed Time}}{\text{Thread Total Elapsed Time}})$
1	28642599	28697517	0.998086
2	77927717	78331794	0.994841
3	182337639	181240044	1.006056
4	53272027	53361133	0.99833
5	93716898	94364259	0.99314
6	100834079	102097029	0.98763

Table 2: Computation Time Ratio between corresponding test cases for both process and threads

Overall, the computation time ratio between process and threads under each test case is close to 1. This indicates that there is no significant quantitative difference between the total time it takes for the tasks of prime decomposition split between multiple processes or threads; However, it should be noted that threads are slightly slower than processes which is surprising given that processes are often regarded as a "heavyweight" process while threads are a "lightweight" process. Since we created multiple threads to execute within the same process, we should have avoided context switching and allowed the sharing of code and data resulting in lower computation times. Furthermore, we did not implement synchronization of data (to access file I/O).

Conclusion

Based on the results, we observed that the computational difference between processes and threads is negligible. In fact, we see slower results when using a lightweight process (threads) for computation over what is supposed to be a heavyweight process (process). This can be attributed to the underlying implementation of POSIX threads on GNU/Linux from other implementations on many other UNIX-like systems. On GNU/Linux, threads are implemented as processes (https://unix.stackexchange.com/questions/364660/are-threads-implemented-as-processes-on-linux)/(https://www.informit.com/articles/article.aspx?p=370047&seqNum=3#:~:text=Linux%20implements%20all%20threads%20as,certain%20resources%20with%20other%20processes.), even though threads spawned from the same process share the same ID. In Linux, a thread is merely a process that shares certain resources with other processes such as an address space. Going forward, we should choose to use process over threads when our application does not depend on data isolation/protection between workers. On the other hand, if we require greater synchronization capabilities, we should use threads in our implementation.