Analysis of Performance of Threads vs Processes in Linux

By
Andrew Burian
BCIT COMP 8005
Assignment 1

2015-01-10

Abstract
The purpose of this experiment is to use multiple processes and threads on the Linux operating system and determine the efficiency of each mechanism. This will enable the selection of the more efficient one when designing high performance software solutions.
It was determined that processes had performance advantage over threads, were more efficient to set up, and were more consistent in their execution time.
It was also found that if the number of workers being created is equal to or less than the number of cores in the machine, nearly no difference exists between threads and processes.
Threads are also clearly more prone to random delays in time than processes are, although the cause of this is unknown.
It was concluded that, if possible, processes should be used to improve performance.

Introduction
This experiment aims to examine the performance of a program running in Linux, which spawn workers to accomplish a task. While threads and processes can be used almost interchangeably in most circumstances, the mechanisms to create and maintain them are different at the operating system level. These differences will show up when measuring the performance between threads and processes while varying the number of workers and the load they are working on.
In the Linux operating system, virtual memory support and copy-on-write semantics implementation lowers the overhead for process creation. Thread creation will not benefit from this.
It was predicted that if the performance between worker processes and worker threads is compared in such a Linux system, the performance will be better in the process variant in all cases.

Methods and Materials
This experiment used a single desktop computer under light load, running a Linux operating system (Ubuntu 14.04), running kernel 3.13.0-44-generic.
The processor of the computer had four 64 bit cores, each running at 1100.00MHz, and each core was listed as supporting one thread.
A custom program was written, called A1, which was used to create the workers and test them.
A1 could be invoked given arguments that specified running either worker threads or worker processes, and a target integer.
The operations performed by A1's workers were to decompose every number from 1 to the provided target integer. After decomposition, the results were then written to a file. Each worker would perform the same amount of work. This set of tasks was selected so that each worker would have to perform processor intensive operations during its decomposition, and then would need to perform I/O

operations writing the results to a file. With both the chance of being pre-empted off the processor, and being blocked waiting for file I/O, this is a reasonable analogue to normal program behavior.
The Linux time program (/usr/bin/time) was used to time the execution of the program, and the time measured was total (real) execution time of the entire program.
In the context of this report, one 'operation' is the decomposition of an integer into its prime factors.

Procedure

For experiment 1, A1 was first executed repeatedly with a constant number of workers, increasing the number of operations each worker performed. This was performed identically between thread workers and process workers.
Then for experiment 2, A1 was then executed repeatedly with a constant number of operations, increasing the number of workers created each time. This was once again done over both threads and processes.

Results

When running with a constant amount of workers, and considering only a small section of operations range, the total execution time was erratic, but both threads and processes fell within the same range. Neither was definitively faster than the other in any case. (Figure 1)

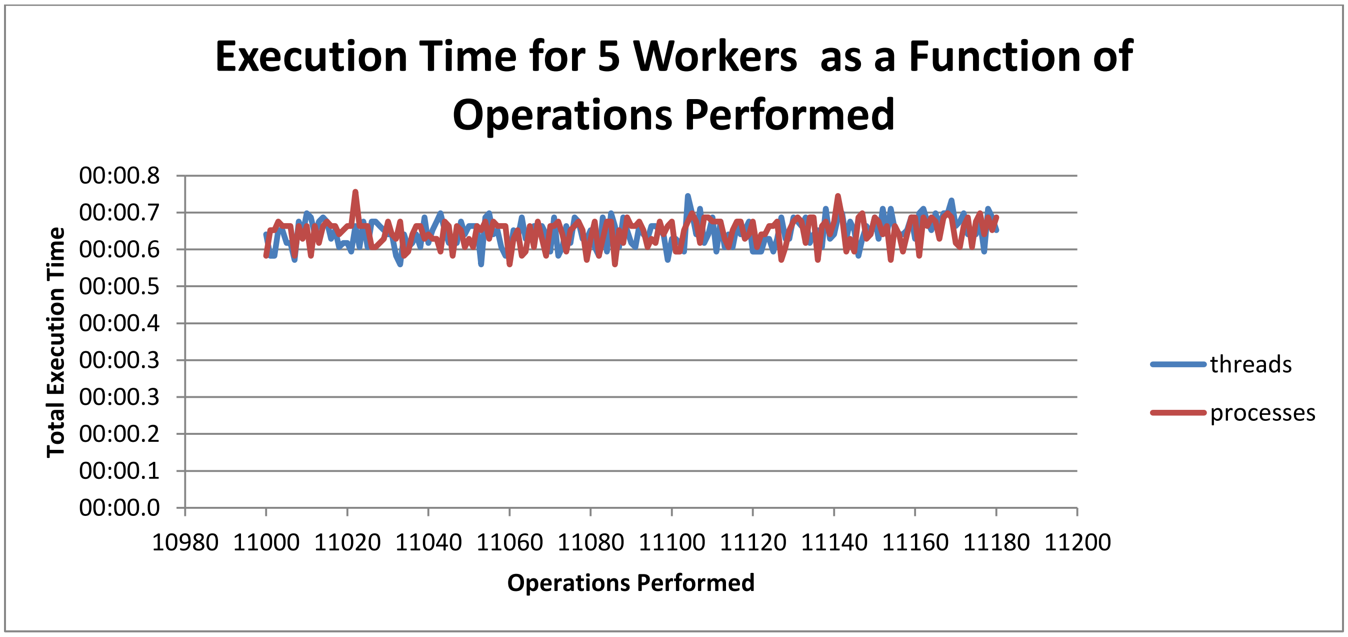

Figure 1 – Constant workers with variable operations, 11000 – 11200

The results of comparing a much larger section of operations (Figure 2), showed a second order polynomial trend in both the execution time of both threads and processes.
While the exponential term was fairly close, only 5.6% greater in processes which could be within expected error for the experiment, the linear term was 52% greater in threads than processes. Both these indicate that as the number of operations continues to increase; the execution times are likely to continue to diverge.
The coefficients of determination on the polynomial fit lines also varied between the two mechanisms. Processes were 2.3% closer to perfect fit than threads, with threads showing increased scatter towards longer execution times

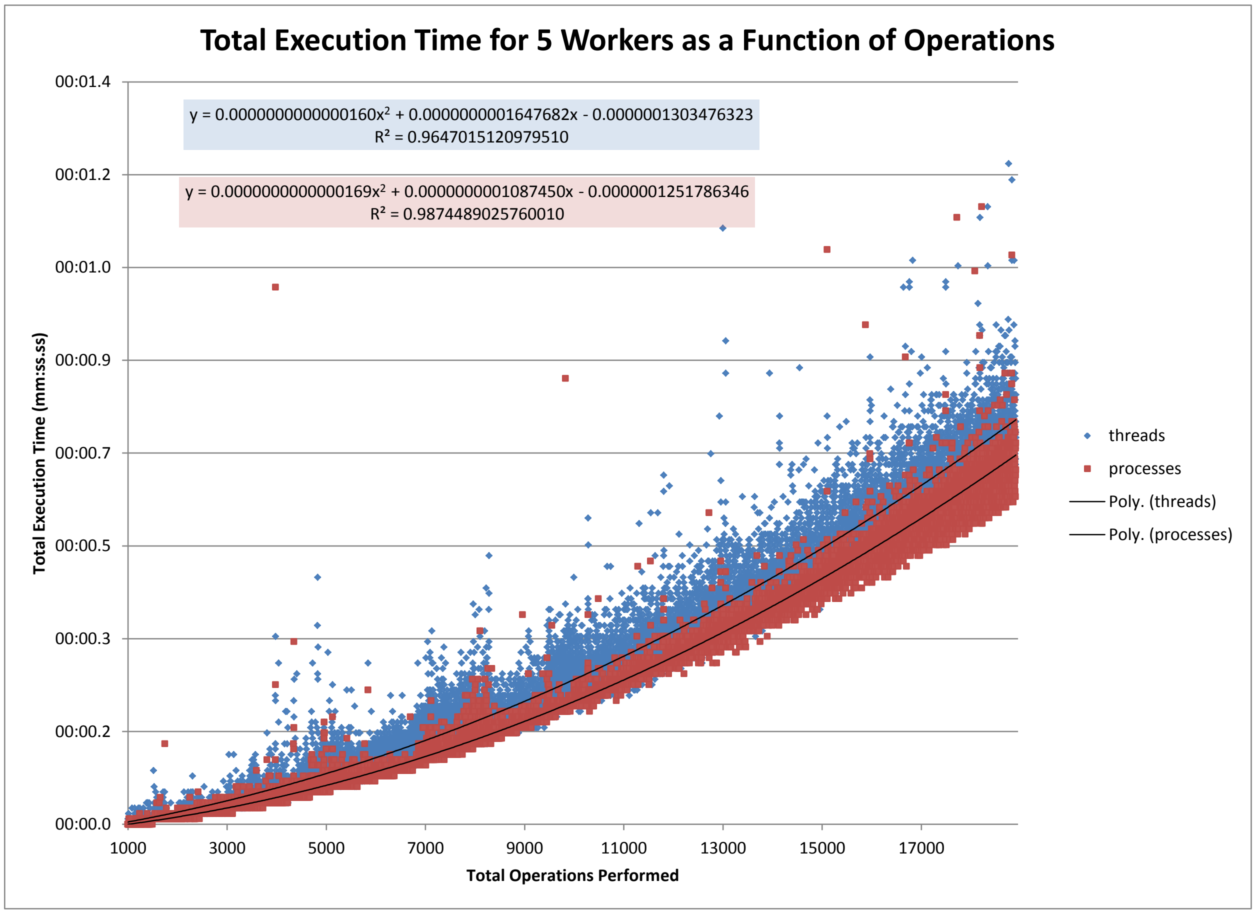

Figure 2 - Execution time for 1'000 - 19'000 ops

Polynomial Equations on Figure 2:

Threads

$$y = 1.60 * 10^{-14}\, x^2 + 1.65 * 10^{-10}\, x + 1.30 * 10^{-7}$$
$$R^2 = 0.9647$$

Processes

$$y = 1.69 * 10^{-14}\, x^2 + 1.09 * 10^{-10}\, x + 1.25 * 10^{-7}$$
$$R^2 = 0.9874$$

When the number of operations per worker was kept constant, and the number of workers created to perform the operations varied, a clear set of trends emerged. (Figure 3)
Time difference between threads and processes was minimal from 1 to 4 workers; from 5 onwards the treads option was consistently slower

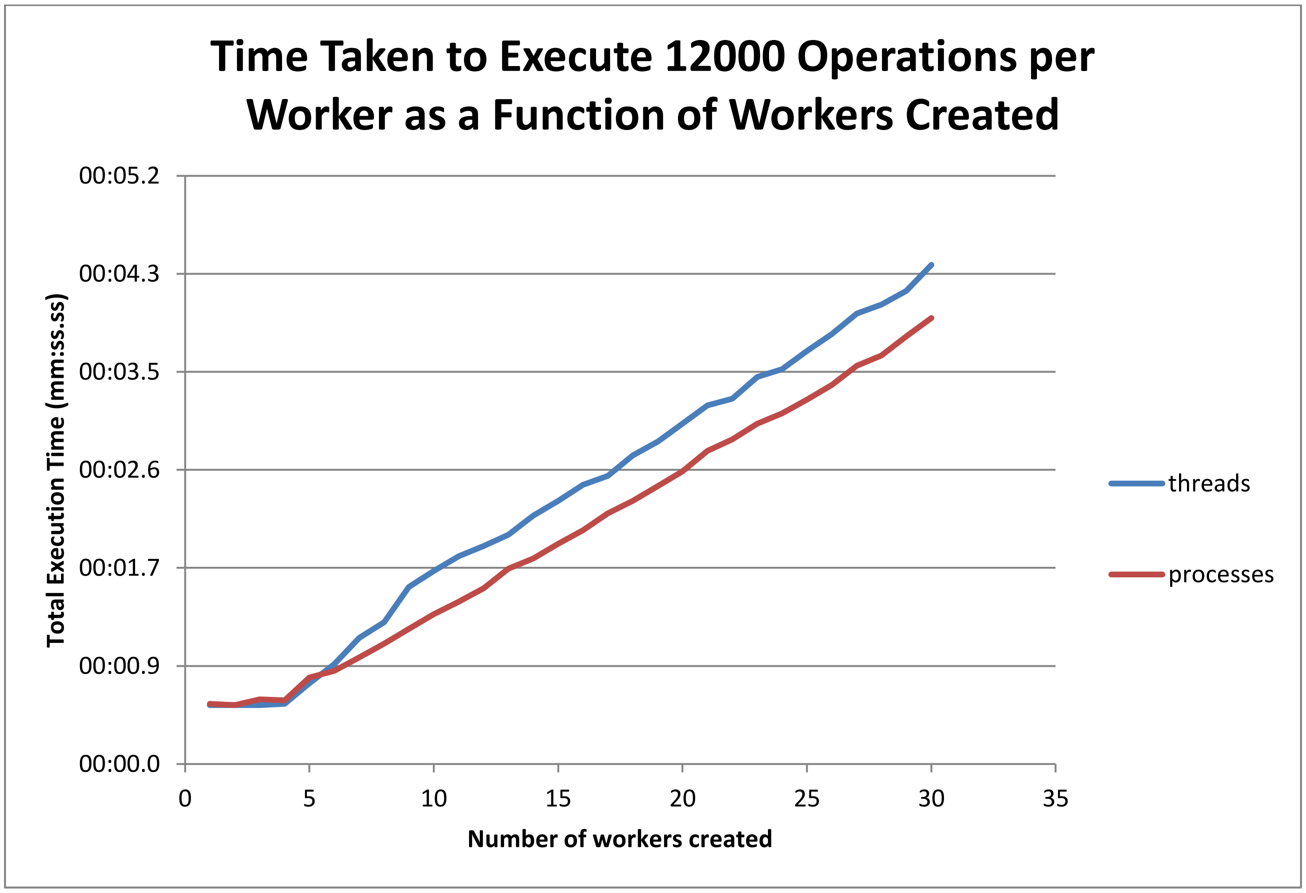

Figure 3 – Constant operations per worker with variable workers 1 - 30

When the data from 9 workers onward is considered in isolation, the trend is found to be a linear increase in both threads and processes. (Figure 4)
These trends are quite accurate, with coefficients of determination nearing 1 in both cases.
The slope of the threads trend is seen to be 3% steeper than that of the processes, indicating that these trends will continue to slowly diverge as workers created increases.
The hypothetical intercepts of the trend which indicated the fixed overhead in creating the worker are found to be 236% greater in threads compared to the intercept in processes.

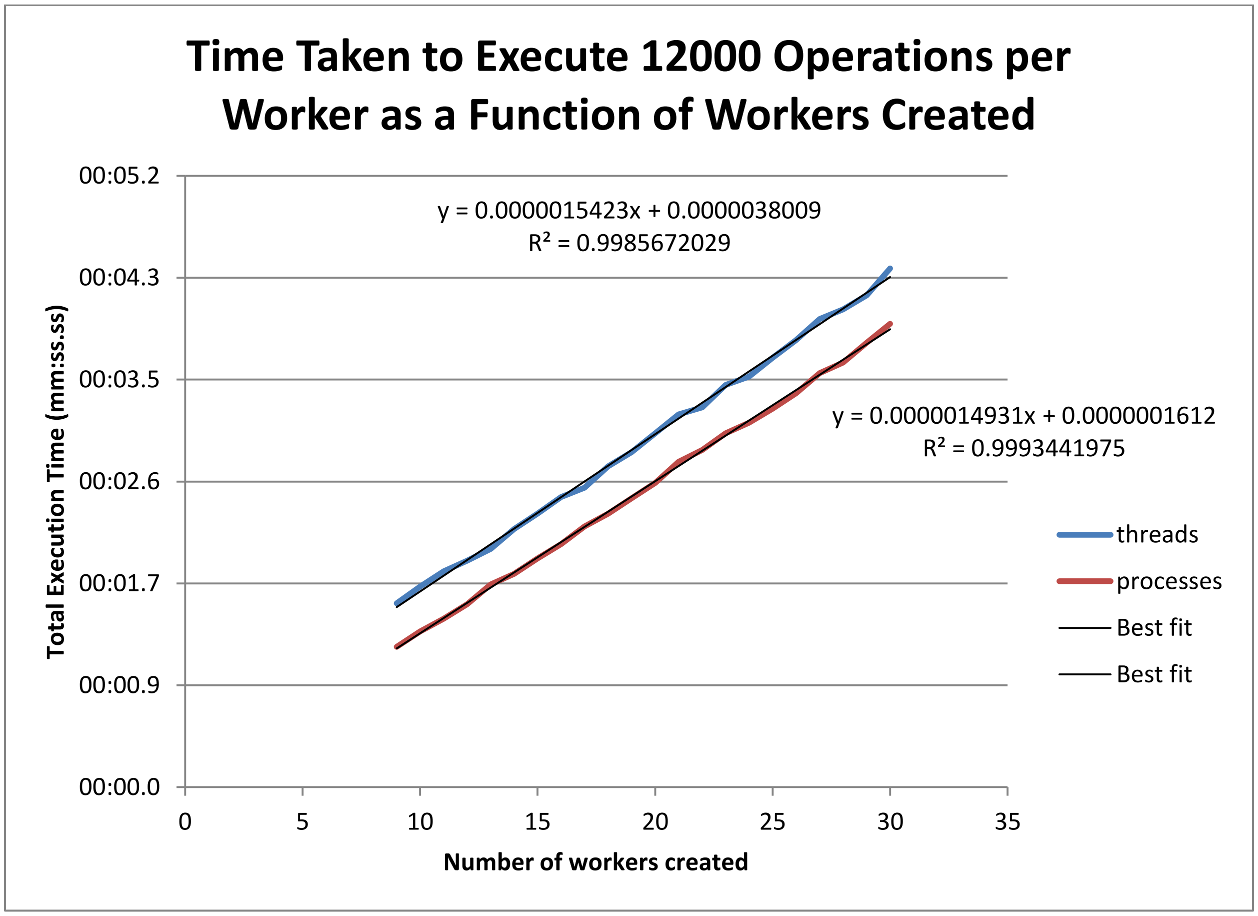

Figure 4 – Constant operations with variable workers, 9 – 30 with trend line

This same data (Figures 3, 4) can be shown differently as the total execution time taken, divided by the number of workers created. As each worker is given its own identical workload, this becomes a measure of total time taken per operation as a function of workers created.
The data shown here (Figure 5) indicates a dramatic increase in efficiency from 1 to 4 workers, followed by stabilization of the trend and more constant efficiency from 9 workers onwards.
Processes are stabilized after 6, while threads decrease in efficiency between 5 and 9, before slowly returning the same range as processes.

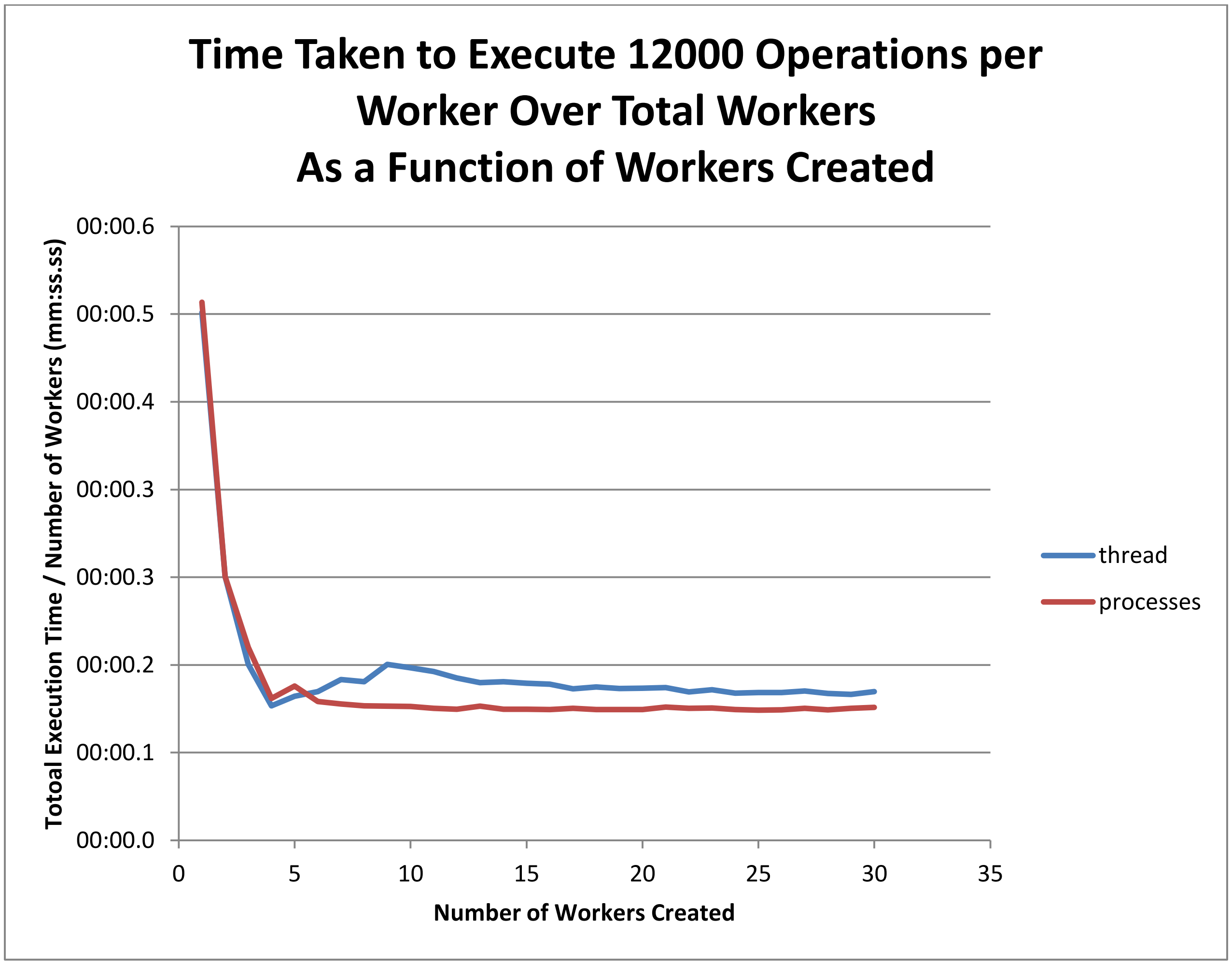

Figure 5 – Total work done against time

A clearer trend also appeared when sampling a larger number of workers created (Figure 6). The diverging trend gradually became more pronounced and the determination of the threads fell as the time increased, becoming more erratic.

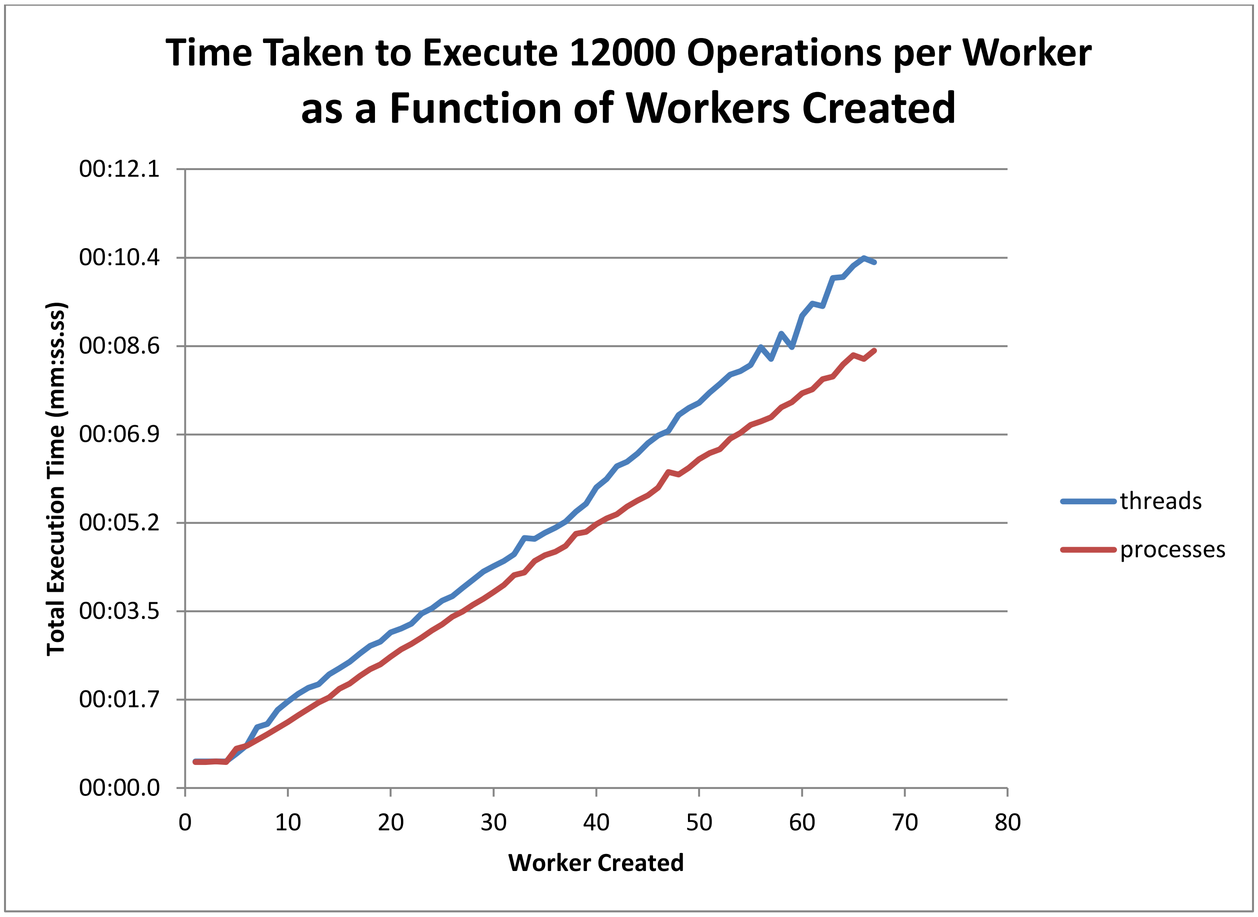

Figure 6- Time Taken with 1 - 65 workers

Discussion

These results clearly indicate that processes outperform threads in most cases, though often the variation in the system environment may cause fluctuations in the time greater than the average difference between them.

When working on similar work loads, it is difficult to see any difference in execution time between the two mechanisms, such as in Figures. The deviations in both are such that both often appear to experience advantages over the other, only to suddenly reverse places. To get a clearer picture of what is happening; analysis was performed over a much larger data set, and trends began to emerge. This data shown in Figure 2 begins to give grounds to make conclusions on. It is seen clearly that both mechanisms follow a polynomial increase, but that threads increase more rapidly compared to processes. It is also notable that threads are far more prone to sudden increases in execution time than processes are. When comparing the determination coefficients, processes have less than 2% deviation from the trend line, whereas threads come closer to 5%.

The experiment also ran a variable number of workers doing the same number of operations in an attempt to quantify the difference caused by setup times. As seen in Figure 3, the execution time remained nearly constant in both threads and processes when the number of workers created was four or less. This is likely due to the abilities of the operating system to distribute over the test machine's four processing cores. As such, no worker was ever unable to be assigned to a processor due to another being created. This is evidenced by the sudden jump at five, when workers would need to be suspended while the parent created other workers. The trends then begin to quickly diverge until more than eight workers are created, after which the trends run nearly parallel. During this section between five and nine, threads rapidly increase in execution time, while processes increase more slowly. The cause of this is unclear, but the cause is possibly the machine's architecture once again. As this oddity only exists when the number of workers is greater than the number of cores, but only up to twice that, it suggests that the architecture is relevant. Hyper threading may be a potential cause, and processes may be more efficient at being hyper threaded than threads. Once past this threshold, a very precise linear trend emerges. Threads are shown once more to be less efficient, with a constant overhead far greater than that of processes, and the slope of their trend continues to rise at 3% more than processes. At higher numbers of workers still, depicted in Figure 6, threads prove to once again be more susceptible to time scattering, and the trend slope rises quickly compared to the relatively constant predictable slope of processes.

Significant questions remain that are outside the scope of this experiment. It is unclear why threads are more prone to large variations in execution time than processes are. The divergence in worker creation time between five and nine workers also remains largely unexplained, though a link to the architecture is still strongly suspected.

Conclusion

In comparing the performance of processes and threads, processes were found to be the superior mechanism in three ways. Process introduced less time overhead in their creation, they were able complete large numbers of operations faster, and they were more consistent in their execution time and less prone to large delays.