CSC209 Assignment 3 Report

July 17, 2016

Name (and cdf): Lino Lastella(c5lastel) - Keirn Munro (g5atreus)

Questions:

Would you expect the program to run faster with more than one process? Why or why not? Why does the speedup eventually decrease below 1? Did the performance results surprise you? If so, how?

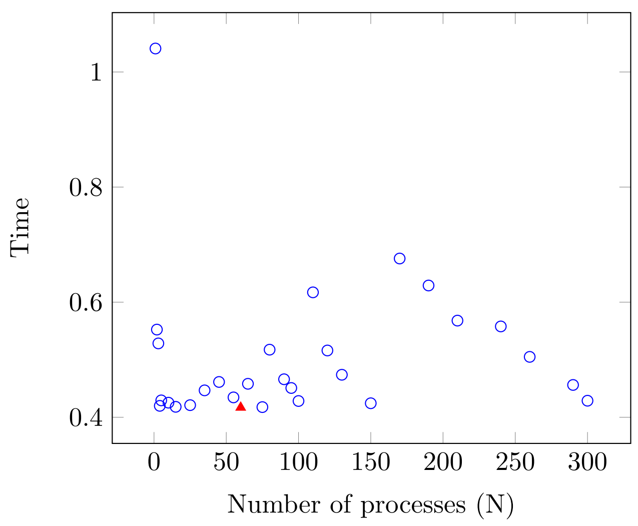
Answers:

We partially expected the program to run faster with more active processes based on results of several divide and conquer algorithms that we have seen in other CS courses (such as: quicksort, merge sort, and dynamic table expansion). With that said, we had a slight suspicion that, with a large enough number of processes, our performance would decline since we knew that creating new processes in a Linux operating system was rather costly. Further, we knew that creating pipes, writing to them, and reading from them would greatly slow our performance since each operation is both costly and occurs in a loop of considerable size. In particular, the algorithm used for reading from the pipes (simply iterating over them) required a loop that iterated N (the number of processes) multiplied by 45 (the number of word sizes to tally).

Indeed, these operations took their toll on our running time (as can be seen clearly in the graph) but we were very surprised to see such a pronounced Gaussian bell shaped distribution. Our suspicions were just and more or less confirmed, but I think we witnessed a much deeper concept of Linux programming at work. As described in section 24.2.2 of The Linux Programming Interface by Michael Kerrisk, the Linux kernel uses a copy-on-write technique with multiple processes, where virtual memory of a process is copied from the parent only if the child attempts to write to said memory. When the number of processes in our experiment was in the interval $(100, 200)$, each was made to compute word counts of around $1-3$ files. Every child process copied a large amount of virtual memory (to store each line when reading from files, to change their file table, etc.) and with that added to the run time cost of forking, we approached the worst case. When $N > 200$, it seems that each process used less virtual memory and so most of the runtime came from forking and processes blocking each other. So it seems the runtime tradeoff is clear; to ensure optimal performance processes must either use less memory and exist in much higher numbers or exist in much lower numbers and use more memory.

Overall, it seems we've learned quite an important lesson in system programming: blindly dividing and conquering problems might not be entirely effective and careful consideration of computational theory must be done before beginning a larger scale project with goals of efficiency.

Data report graph


Time
Number of processes (N)