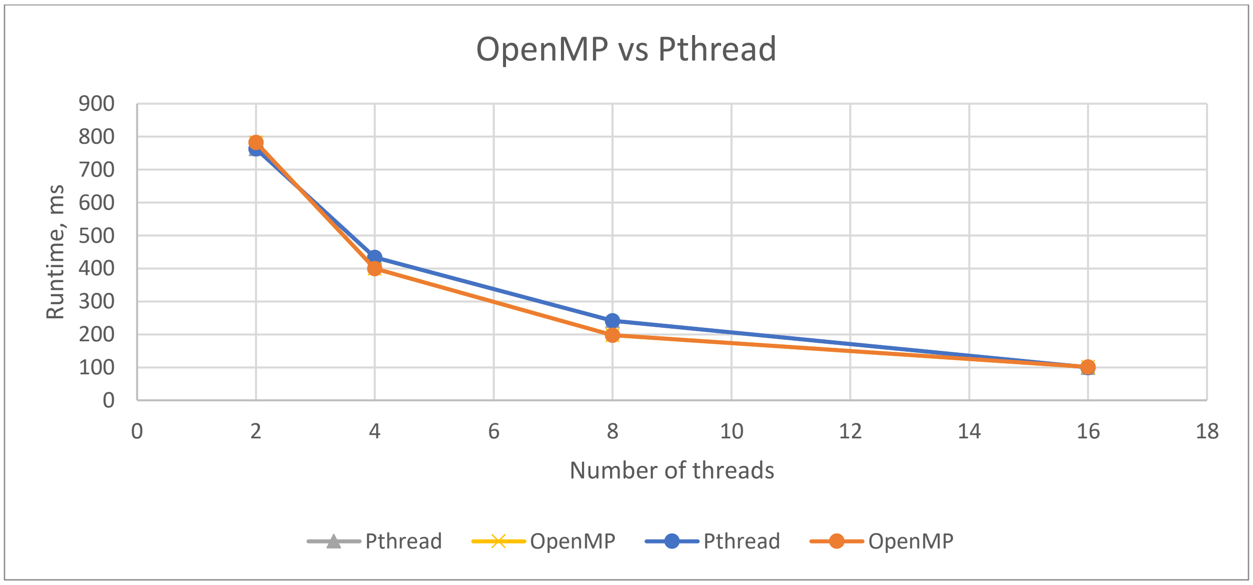
Alex Breault

CIS 625 Homework 2

Parallelism Performance Analysis

To begin, we were given the task of increasing performance by running code in parallel using OpenMP and pthreads. The code runs through a for loop and performs, not in this order, three additions, two multiplications, and a division which can be very computationally heavy. One iteration of the loop could take up to 67 cycles so splitting the program up in chunks and running on multiple threads can help performance greatly.

These tests were ran on the elf nodes with Linux version 3.10.0-862.11.6.el7.x86_64 (builder@kbuilder.dev.centos.org) and gcc version 4.8.5 20150623 (Red Hat 4.8.5-28) (GCC). I used a shell script to loop through the given number of threads (2, 4, 8, 16) and used 512 megabytes of processor memory. I compiled the C code with the GCC compiler. I ran 10 jobs with each number of threads to gather an average run time. I measured the time from just before to just after all the computation to gather the most accurate numbers possible. Here are my results:



This graph shows that, as the number of threads increased, the average runtime decreased on a smooth downward curve. Both methods were about the same in each thread category, so I will analyze the slower of the two, pthreading. Going off fastest times for each thread group in regard to pthreads, we went from 722ms to 364ms to 197ms to 97ms. This is cutting the run time in half each time, which is an incredible speedup for something this small scale. From my tests, I did see a small amount of race conditions. When it was 4 and 8 threads being used, there was one outlier that was much larger than the average. For 4 threads and 8 threads, it was 778ms and 533ms, respectively. I believe this was the case to where two or more threads were fighting over the same work or one thread was waiting on another to be able to continue its computation. I also tracked memory utilization. The average virtual memory used was 105,466KB and the average physical memory used was 796KB.

I received a two times speedup for OpenMP and Pthreads because it allowed the work to be chunked into separate partitions. When running in parallel, the work gets split between the active threads and each thread does what it can at that moment. In conclusion, my tests were successful because of the speedup I received. It showed that, in our case, the number of threads you throw at your tasks, the faster it will run. In other cases, more threads might not help because they could be fighting for work but in this example, it worked exactly as planned.