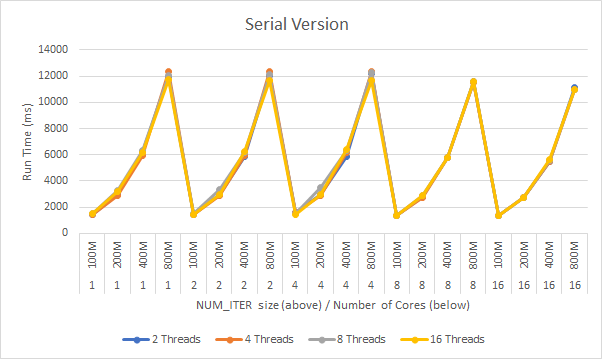
The Experiment:

In order to thoroughly test the effectiveness of multithreaded code. A series of trials was conducted on three different versions of the same file. The base file named pt1.c was written using single threaded code and served as the control. The files pt1-omp.c and pt1-pthread.c were ran using 2, 4, 8, and 16 threads respectively. The difference between these two is pt1-omp.c is implemented using openmp and pt1-pthread is of course running pthreads. All three files were ran using a NUM\_ITER size of 100, 200, 400, and 800 million respectively. Finally, all three files were ran using 1, 2, 4, 8, and 16 cores respectively. The idea is to observe how the multithreaded code behaves under different loads with different numbers of threads and determine at what point, are they most effective. In general multithreaded code can only get so efficient. Too many threads working on a task can cause logistical problems that lead to extended runtimes.

The Environment:

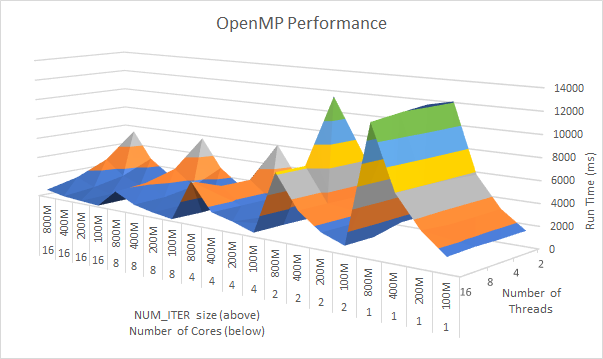
All of the code in the following tests was ran on the Elf class of nodes. Tests on cores 1, 2, and 4 ran on the 1st tier between Elf 1-56 while tests on cores 8 and 16 ran on the second tier between Elf 47-72. These nodes have two 8-Core Xeon E5-2690 processors with 64GB of RAM. At runtime, the system was running Linux eos 3.10.0-862 with the GCC compiler version being 4.8.5. The client side of the tests was a Dell Latitude E7470 running Windows 10 Pro with a 2.45GHz Intel Core i5 with 16gb of RAM. The SSH client was MobaXterm v10.7 build 3650.

The Results

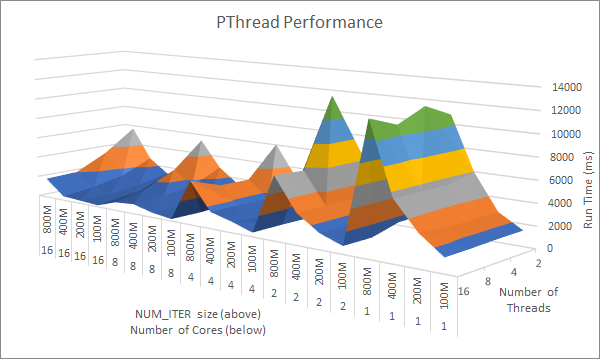


The serial code consistently reported an almost exponential growth in terms of runtime, regardless of the number of threads. This is to be expected, by definition, serial code can’t take advantage of using more than one core. That said, there was a slight reduction in runtime as the number of cores increased. Most likely this was due to anomalies in the system such as less

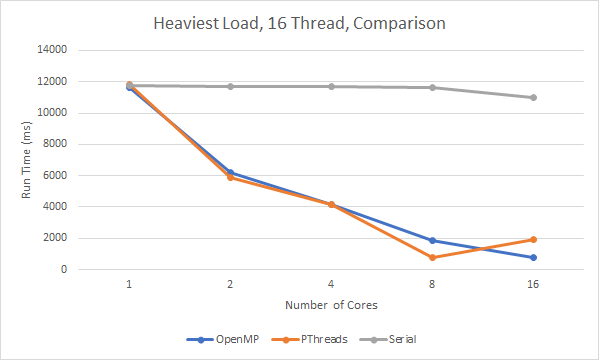
load on the servers or simply coincidence.



openmp produced some interesting results. Most Notably, the performance increase when running sixteen threads, on sixteen cores, under the heaviest load is 1483% faster than running the same load on a single core. The single core performance roughly mirrors that of the serial code. This makes sense. It doesn’t matter how many threads are created if there’s only a single, non-hyperthreaded, processor to execute the code. In general, as the number of cores and threads increases, the runtime decreases with diminishing returns. An outlier here is the test running two cores, with two threads, under full load. Here the runtime skyrocketed to just shy of 12000 milliseconds. Most likely this was caused by a race condition. If the code were to get hung up on trying to access the same set of data, one thread would have to wait significantly driving up the runtime. At its worst, the code performed on par with the serial code. Due to the significance between it and the other tests, this was most likely not caused by an anomaly in the system.



Rather surprisingly, the code utilizing pthreads performed almost identically to that of the openmp tests. Being that pthreads is a lower level of abstraction and doesn’t utilize system calls like openmp does, I figured that the performance boost would be even greater. In this case, this didn’t seem to matter. This benefit would most likely be more evident if these tests were conducted on even larger sets of data. The slowdowns caused by openmp, at least in this set of trials, was not enough to cause a major difference. In regards to performance, pthread code was most optimal when running on only 8 cores. This resulted in a 1517% speedup when compared to the same load on a single core. When given 16 cores, this slows down to only a 623% speed up. It is at this point where the pthread code hits the floor in regards to how efficiently it can run. Metaphorically, there’s too many builders on a job site and logistically it’s slowing the project down.

Conclusion:

In the event that code can be parallelized in such a way that race conditions are avoided, doing so with either openmp or pthreads creates a significant boost in code runtime. That said, there is a clear difference between these two methods. Running code using pthreads is most effective when hardware is limited to a smaller number of cores. It “bottoms out” quicker so to speak. openmp meanwhile operates better at a higher number of cores. Its performance continues to decrease runtime with diminishing returns through the 16-core mark. Most likely, openmp would see a spike in runtime beyond this point in a similar fashion to pthreads, however, further tests with higher numbers of cores and threads would be required to test this. Furthermore, If I was to run this experiment again, I repeat all of the above tests a minimum of three times in order to get data averages instead of data points.