The Experiment:

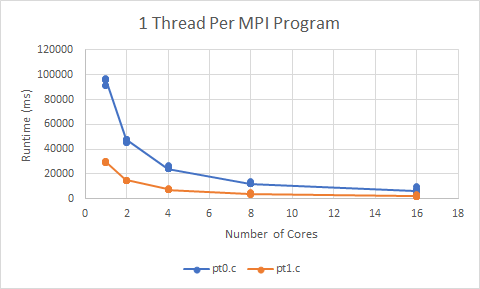
The purpose of this experiment was to test a more complex, hybrid style of parallelization and compare the results. This was accomplished using a combination of OpenMPI calls and OpenMP compiler commands on two files, pt0.c and pt1.c respectively. The first file calculates a numerical integration equation using the following function: f(x, y) = -cos(x)\*sin(y)\*exp( -((x - PI)\*(x - PI) + (y - PI)\*(y - PI))). This is of course parallelized using MPI. The second file tests a number of numerical integration optimizations using the following function: f(x) = cos(x)+(pow(fabs(6.0-x), 2.0/15.0))+2\*(pow(fabs(5.0-x), 4.0/35.0)). The MPI program uses broadcast to calculate the result using a number of different methods to determine the most effective method. The value and run time from this method is returned. In order to thoroughly test this, both files were subjected to tests running 1, 2, and 4 threads respectively, as well as 1, 2, 4, 8 and 16 cores respectively. All of the above tests were repeated 10 times each in order to produce a reliable value. Each test was given exactly 1GB of RAM per core.

The Environment:

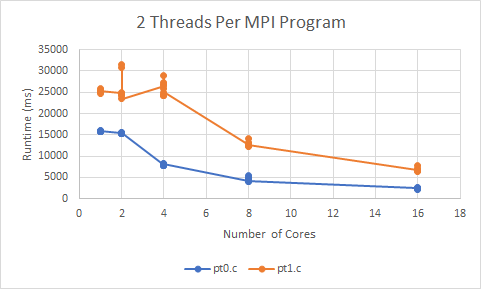
All of the code in the following tests was ran on the Elf class of nodes. More specifically, the tests were ran on the first two tiers, elves 01-72. These nodes have two 8-Core Xeon E5-2690 processors with 64GB of RAM. At runtime, the program was compiled and ran using OpenMPI v3.1.1. The client side 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

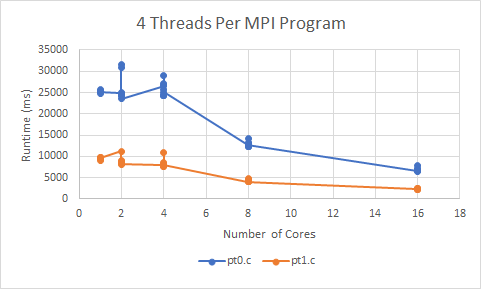
Given one thread per MPI program, the two programs performed roughly as expected. As the number of cores increases, the overall runtime decreases exponentially. I graphed all three sets of data using line-connected scatter plots in order to highlight any potential race conditions and/or anomalies in the data. Given that only one thread was allocated to each MPI program, race conditions were minimal to non-existent. This makes sense. If each program only has a single thread, it's impossible for two threads to become locked into a race condition slowing down the program. In general, pt0.c returned roughly a 15.7 times speed up and pt1.c returned roughly a 15 times speed up when comparing the 16 core tests to the single core tests.



Upon allocating two threads to each MPI program, the results became a lot more interesting. In general, the programs follow the same general trend. As the number of cores increase, the runtime decreases albeit at a small rate than the previous test. In the 2 core tests, there was a weird spike of almost 5000 milliseconds. This could’ve been caused by a race condition, however, given the data around it, it most likely falls within the margin of error for the general trend. One thing I was not expecting was pt1.c to perform worse than pt0.c. Given two threads, both performed exceptionally better than single thread test. I’m not 100% certain what caused this. This was most likely caused by either a bug in the code or some minor race conditions. Load on the servers was fairly light upon runtime.



Unlike the previous test, the 4 threaded tests performed roughly as expected. pt1c, much like in the single threaded test, out performed the single numerical integration. This makes sense. By design, pt1.c should determine the most efficient numerical integration method resulting in shorter runtimes. Another interesting thing regarding this test is improvements in runtime have effectively stalled. The results for pt0.c almost mirror that of the previous test (give or take a few hundred milliseconds). pt1.c performed better than the previous test, but the improvements could easily be disregarded as being within the margin of error. This complete stall in performance enhancement is most likely due to too many things occurring at once. Essentially having “too many cooks in the kitchen” leads to a decreased return on investment and eventually, runtime begins to increase.



Conclusion:

To be perfectly honest, the biggest pitfall to this experiment was time. A number of projects in my backlog from various classes meant that a number of things could’ve been tested more effectively. I could’ve ran another test running 8 threads, I could’ve spent more time compiling data into a better, cleaner format than just connected scatter plots, and most importantly, more time could’ve been spent in the creation of this report. That said, these tests of the OpenMP and OpenMPI hybrid style of parallelization highlight one key aspect. This combination of parallelization techniques can handle infinitely more in less time than what either style can accomplish by itself. That in itself would make for an interesting future test. A runtime and memory usage comparison between standard OpenMP, OpenMPI, and the hybrid model.