Project 4 Report – MPI Hotplate

## Kyle Pontius

# Introduction

Parallel processing is a powerful tool that requires careful planning. If we want to tackle large problems, with lots of data, we have two general approaches that we’ve touched on this class. First are multithreaded applications leveraging OMP. This consists of many threads running simultaneously on a single machine, preferably with a multi-core processor. An alternative we’ve studied is networked, distributed processing using MPI. This distributes pieces of a problem to multiple machines, which then return a piece of the result to be merged.

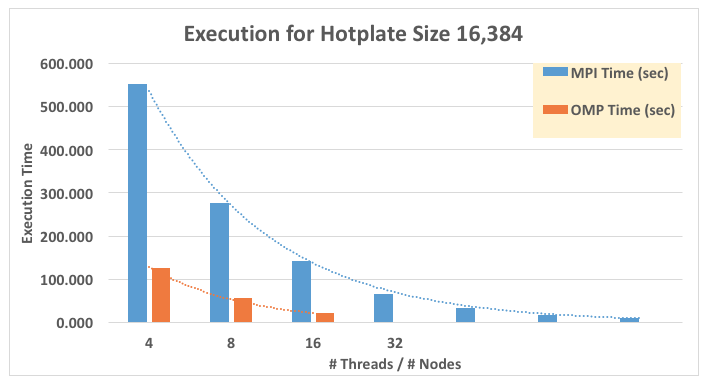
For me, this problem posed several interesting questions, some of which were: How will MPI’s communication time stack up against OMP’s nearly instant performance? Do separate nodes with a given problem size perform better than separate cores (working on a thread) with that same problem size? Which type of parallelism’s performance degrades more quickly as problem size significantly increases? While not each of these questions will specifically be answered, there were many that were. This was a fun and meaningful learning activity for me.

# Approach

Please note, as I haven’t finished the p-threads project my results will focus on the performance results for OpenMP and MPI. My approach to comparing these two programs is primarily looking at the number of nodes for MPI and the number of threads for OMP. While this is not an excellent apples-to-apples comparison, I did find it enlightening to view the results with this perspective. Obviously the communication time with MPI was significantly higher than that of OMP, relatively speaking, the computation power seems reasonably comparable.

# Experimental Setup

For my tests, I’ve decided to run both the OMP code and MPI code on the FSL Supercomputer. In an effort to minimize performance noise and outliers from the results, I’ve run every test twice and reported the average. In my experience, the results were very consistent, always falling within a few seconds of each other. I believe this will help the results more accurately reflect the real execution time of the program. Additionally, in order to get a good feel for the performance of MPI, in particular, I also ran up to the highest number of nodes FSL would allow me to do within my scheme. Since my nodes had to be divisible by 8, I was able to run the program on up to 256 nodes. OMP, however, was limited to up to 16 threads as the FSL didn’t offer available nodes with a higher thread count. The last thing worth pointing out is that I limited the lowest result on OMP to 4 threads because MPI’s results timed out at 10 minutes below 4 nodes. This felt like a reasonable low water mark as the trends as easily visible before going below that point, and we covered execution time below 4 threads on our Project #1 report.



# Results

My hotplate’s relaxation took about 360 iterations. Looking more closely at the graph there are three elements that really jump out. First, the similarity between the two trend lines seems very telling. Both decrease at a rapid rate; with MPI decreasing a little faster than OMP (that fact is difficult to see from the graph, so I included the numbers themselves after the report’s conclusions). The trend line placed on both graphs is exponential, with MPI following the trend line much more closely. If this doesn’t make much sense, please let me clarify. As you can see from the code, my start and end time for the MPI Hotplate implementation begins after much of the “serial” code has been executed. This execution time does ***not*** much of the code that initializes MPI itself. In essence, this means that the execution times really represents the speedup in the parallelization portion of the code, not the serial portion.

Second, the execution time’s growth pattern of both OMP and MPI’s graphs is very similar. This leads me to believe that the speedup achieved by both, when we minimize or disregard communication time altogether, is effectively very similar for some problems. This is interesting to me because it answers one of my initial questions about which performance degrades more quickly, mentioned previously in the introduction. Also, it concretely depicts to us that while MPI and OMP effectively work differently, they scale in a very similar way. Understanding this, and the effect scaling has on either implementation, can really help in determining which is the better tool to use for a given situation. After using both, problems with large datasets and relatively few chunks lend themselves well to multithreading with OMP; however, when working with datasets that can be broken up into hundreds of pieces, MPI is very robust.

Third, the differences between OMP and MPI’s performance speak volumes. One difference is there ability to scale. OMP effectively can only go up to the number of threads the hardware supports. Generally, this is reached very quickly at scale. However, the communication time between threads is extremely small, relative to MPI, and lends itself well to small, but parallelizable tasks such as image editing. MPI, on the other hand, can scale to immense proportions. I was able to stretch to 256 nodes, which is genuinely impressive. Imagine what the potential could be given a much higher number of nodes to work with. Obviously the problem size would need to increase significantly to make a higher number of nodes worthwhile, but the potential computation power is simply exciting!

# Conclusions

Three points really stood out to me in reviewing the data from my application. First, speedup with this embarrassingly parallelizable problem is enormous, both for OMP and MPI. Second, the execution time’s growth is very similar for both MPI and OMP, leading me to believe that both are similarly robust given the proper problem size and type. Third, both OMP and MPI must be used properly, using the right tool for the problem and hardware available.

# Additional Results

