Jonathan Alexander Gibson

jagibson44, T00198998

Program 4 (MPI Matrix Multiplication)

During this programming assignment I went through several steps of success and setbacks. I started by writing a basic C++ program that would populate a matrix, with 16 (4x4) hardcoded values, and print it to the screen. After I got that working, I looked up a square matrix multiplication algorithm on Stack Overflow, familiarized myself with what was going on, and generated a correct matrix multiplication with the known values (I used Maple to confirm the result of the multiplication).

Next, I went back to some code that I got from Dr. Brown (from CSC 2100), to generate random values for the matrices, instead of hardcoded ones. Throughout this time, I also generated and deleted the matrices via dynamic allocation. But, when I couldn’t figure out how to generate random ***double*** values for the dynamically allocated matrices, I simply reverted to static allocation.

After I confirmed that my code was working with small randomized matrices, I decided to test auto partitioning parallelism using ***#pragma omp parallel for*** on slightly larger matrices. I used the HPC server to test the automatic partitioning and confirm that the values were still correct. At this point, I was a bit stuck. I wasn’t sure how to manually partition the work, so I consulted Steven and Ahsan for insight. Steven showed me a rough draft of how he thought the auto partitioning should behave based on the ***matrix\_size***, ***thread\_ID***, and calculated ***chunk\_size***, ***start***, and ***end*** values.

After modifying his notes to work with my code, I made some progress, but also ran into some problems. My calculated ***end*** value was “off by one”, and my final matrix was only generating values down the diagonal of the matrix, equal to that of the current ***chunk\_size***. I then drew out two matrices on some engineering paper, to walk myself through the steps, for troubleshooting. I soon figured out that my indexing scheme didn’t match my calculations for ***end***(using ***<*** instead of ***<=***) and I had manually partitioned the work for both the columns and the rows. Once I fixed these two issues, my code worked as expected and generated correct results.

Although my code was functioning properly, I noticed that my execution times weren’t changing much, despite the number of threads I was using. I generated nearly 80% of my output files before I realized that I needed to increase the ***–cpus-per-task***sbatch value to see any accurate speedup. Once I rectified this problem, my execution times started decreasing proportionally with the number of threads, and I started generating sound data, shown in Table 1. I then used the data to generate plots with MATLAB, shown below.

Figure 1: *Execution time for 1 process.*

Figure 2: *Execution time for 4 processes.*

Figure 3: *Execution time for 8 processes.*

Figure 4: *Execution time for 16 processes.*

Figure 5: *Execution time for 28 processes.*

Figure 6: *MPI execution time surf plot based on varying matrix size and process count.*

Figure 1 through Figure 5 show scaled representations of the execution times for each thread count, based on varying matrix sizes. These bar graphs all looks very similar because they are scaled to the maximum execution time values for each set. I concatenated the data into a three-dimensional graph to get a more relative representation of the data. Figure 6 shows the gradient curve of the execution time for the matrix multiplications based on varying matrix size and thread count. It is very clear that execution time drastically decreases with decreases in matrix size and increases in thread count.

Table 1: *Collected Data.*

Using the average execution time, of two runs per thread count, I calculated the speedup for each thread count, with constant matrix size. Then, I averaged these speedup times to get an approximate speedup curve, when using additional threads.

To my surprise, the speedup curve shows that the speedup is super-linear. Although this is non-intuitive, there are a few reasons why this might be the case. First, matrix multiplication has no data dependency; none of the threads need to wait on each other for any calculations. This significantly reduces both overhead and execution time compared to dependent data sets. Second, the code was operating on simple, low-level computations; the complexity did not increase as the size of the matrices increased. Finally, the executions could have taken advantage of the cache effect; with an increase in CPU resources, there is also an increase in the amount of available cache. This could potentially reduce the data retrieval time from memory, by storing larger amounts of data in cache, instead of RAM, with the increasing number of CPUs. The speedup results are shown in Figure 7 below.

Figure 7: *Average Speedup.*

The MATLAB code used to generate the MPI graphs is shown below.