Jason Drain
CSCE 569

Assignment 3 Report

Implementation:

The implementation for this project fell into three main tasks, as delineated in the assignment description. The first task was to decompose the data and distribute it to the different processes. This was done in a uniform fashion whereby the matrix was divided up evenly into chunks of rows.

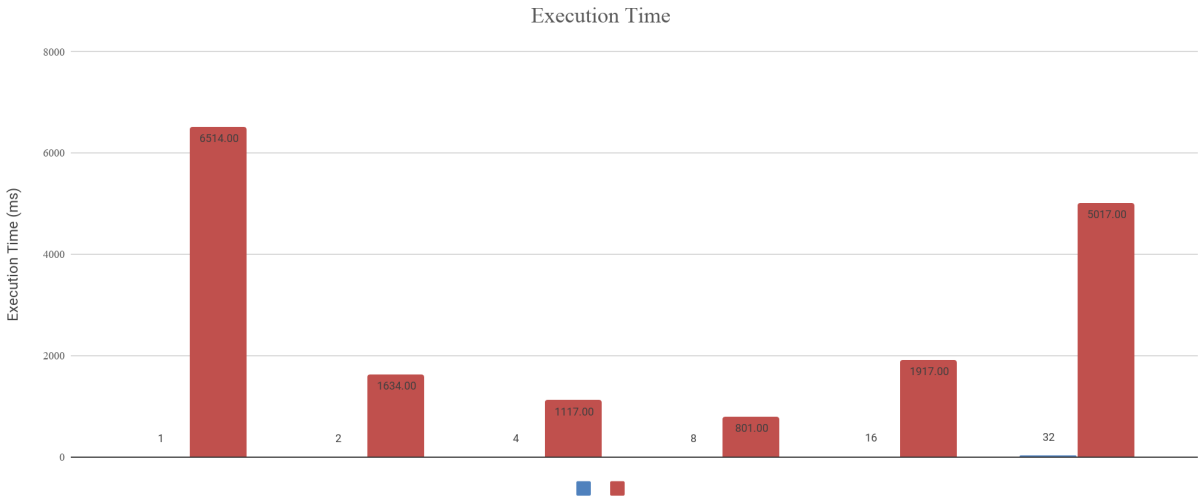
More concretely, my code performs this task via MPI_Send and MPI_Recv, depending on which process is executing the code at that time. If the process number is 0, the program iterates through each of the remaining process and sends out a pointer, via MPI_Send, to a memory address in the array that corresponds to the chunk that process needs to receive. If the process number is not 0, it calls MPI_Recv so that it can get the pointer it needs from process 0.

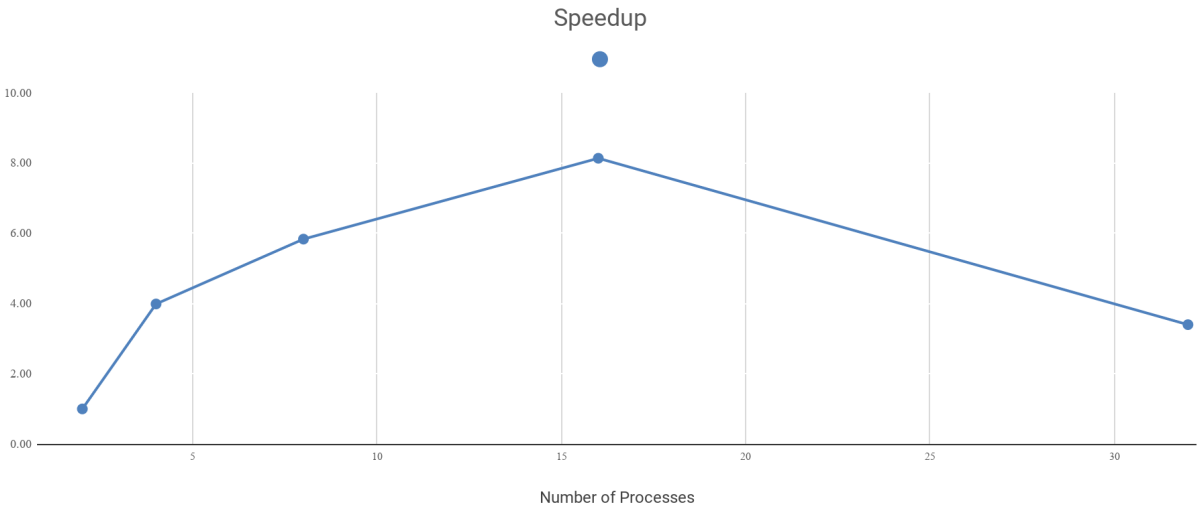
The second task was to parallelize the jacobi iterative method itself. Firstly, data from the last iteration needed to be copied over to a secondary array so that it could be used again in calculating the current state of the matrix. Since different processes contained different sizes of data (due to the overlapping nature of the subarrays), I needed to set the bounds accordingly.

After this was copied over, the next part of the program sent specific boundary data from one process to another. Furthermore, the first and last processes only need one row of boundary data, while the others need two rows, one from each of their neighbors. As in the first task, I used MPI_Send and MPI_Recv to pass the data back and forth between processes. I opted to create a pipeline of messaging, wherein the first process sends its data, the second one receives, and then the reverse happens. Next, the second process would send to the third while the third receives, and then the third would send to the second while the second receives. This sequence of function executions would continue in the same manner until all processes have exchanged data, and the pipelining ensured no deadlock occurred.

After the data passing occurred, the algorithm was run for each subarray. Then the global error was calculated via MPI_Allreduce. This took the local error from each process, summed it all together, and broadcasted the result back. This allows the processes to decide when to conditionally terminate the iteration. The final step after this was to use MPI_Gather and get all the final data from each process aggregated back to process number 0.

Figures and Interpretation of Results:





This first figure details the execution time in milliseconds as a function of the number of processes, while the seconds shows speedup as a function of number of processes. One may notice one odd thing, particularly that the speedup from one process to 2 is about 4 times. Unfortunately, I do not have an explanation for why this occurs. It definitely seems incorrect, since doubling the number of processes should yield at most two times speedup, especially with this problem, as decomposing the data does not really give any advantages in the amount of work that needs to be done -- the nature and amount of computation stays the same, it is just distributed among process.

However, other than this the data seems to trend as expected. Other than from 1 to 2 processes, the increase is sublinear, reflecting the overhead involved in spinning up new processes. There is a critical point somewhere between 8 processes and 16 where the overhead outweighs the benefits of increased parallelism. By the time we get to 32 processes, this is especially clear.

Another snag I ran into while testing my code was in using the bridges system. Unfortunately, I was only able to get results on the system for high numbers of processes (32, 56, and 112). Attempts to run the code with just 4, 8, or 16 processes were unsuccessful even with multiple tries. Therefore, it should be noted that the figures are derived from data I got running the code on my laptop.

While it was somewhat frustrating to be bewildered by certain results I was getting as well as the performance of the bridges system, I still learned a lot about how to make processes communicate, and why they might need to share data in the first place. If I were to implement another program like this again, it might be beneficial to implement the algorithm itself. This could yield some insight into why I am getting such an unexpectedly high speedup from 1 to 2 processes.