ECE 420 Lab 4 Report

Andrew Juwon Park - 1442834
John Tyler Beckingham - 1500702
Zhijie(Max) Shen - 1494084

Description of Implementation:

Our program begins with initialization of MPI. For all of the processes running, it starts by reading the data generated by the datatrim program given to us in the labkit. Then, the data are used to initialize the nodes using node_init() in the Lab4_IO program. Since we are running multiple processes, each process processes only a chunk of data. In init_values(), we initialize the result array and a local array which will be used to hold the different chunks of nodes that are unique to the process. Once initialization is completed, it executes solve(). In the solve function, we iterate through our chunk sized local result array and calculate the contribution for the chunks in each process. After calculating through all the contributions for all of the processes, MPI_Allgather is called for all of the chunk arrays. MPI_Allgather is a method that takes chunks of arrays from different processes, combines all of them and broadcasts them to all of the processes. By doing this, all of the processes will now have an updated array of nodes with the contributions. This process is repeated until the relative error is small enough. Once that is finished, we close the MPI and the program terminates.

Attempted Methods of Optimization:

We tried the same method above on the update_contributions loop by having each process update their local contributions before performing another AllGather to get those results to each process, but due to the relatively small amount of work being done in the update_contributions And the relatively large amount of overhead work required for executing the AllGather, this optimization only reduced runtime.

Experimental Observations

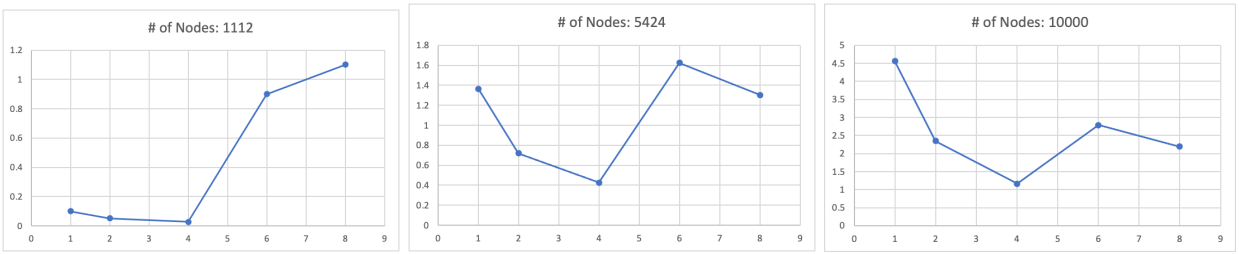
# of nodes	Single machine four process	4 machines
1112	0.026658	0.037511
5424	0.424616	0.416694
10000	1.164615	1.221434

# of processes \ # of Nodes	1112	5424	10000
1	0.099786	1.362168	4.559714

2	0.050849	0.720314	2.350447
4	0.026658	0.424616	1.164615
6	0.899876	1.623155	2.790325
8	1.100337	1.302902	2.194839

Performance Discussion

From the first table, we noticed that the time difference between using a single machine with four processes and using 4 machines with one process each is small. Overall, the single machine is slightly faster than using four machines, which is expected due to communication overheads between machines.



The above is a graph generated by the 2nd table where y-axis is the time and x-axis is the # of processors. All of the three graphs show that the fastest time usually happens with ~4 processes regardless of the nodes. However, when it goes past 4 processes, we see a sudden spike in processing time. This can be due to the fact that lab machines cannot handle more than 4 processes at once. Thus, any data gathered for more than 4 processes must be looked at critically and only be considered relative to these lab machines. Our parallelized optimizations caused a noticeable speedup as we increased the number of processes from 1 to 4, with a maximal speedup of 3.92 with 10000 nodes and 4 processes.

There were no partitions done to the data or graph as each computational step depends on an unknown amount of data (incoming links), so each process keeps track of the entire data set. However, work is divided among the processes as each computational step is dependent on static values.

For this lab, we used MPI's AllGather() function as our only mechanism of communicating from process to process. The way that we designed the algorithm and divided up the work between the processes available let us get away with only one communication mechanism while also being much more efficient that the serial implementation.