**Kody Bennett**

**CSC 4760**

**Assignment: Program 3**

**Due: November 16, 2014**

The objective of Program 3 was to rewrite the Game of Life program from Program 2 to use the MPI distributed computing library on CSC’s remote server, Tiny. The program was run using a 40,000 x 40,000 grid wrapping borders. The program was run to simulate 100, 200, and 400 generations with 2, 4, 8, 16, 32, and 48 processes. All attempts to run the program with a single process resulted in the ssh session on the remote server being terminated, likely due to perceived inactivity, which in turn caused all process from that session to be terminated. For purposes of comparison, single process execution time was assumed to be twice the time it took for 2 processes.

Work was portioned between processes in a similar fashion to the one used on Program 2. Processes 0 through n-2 are allocated rows each with the last process getting the remainder. The main difference is that since each process could potentially be on a different machine, each process needs its own individual copy of the section of the array that it is working on. This was accomplished by using **MPI\_Scatterv** to distribute the array to each process. Before the computation of each new generation, each process requests a row from the processes that are a rank above and below its own rank. This accomplishes 2 things. First, it allows each process to calculate the state of the cells on the top and bottom border of its allocated grid section. Second, it synchronizes all of the processes so that no one process can start computing the next generation without receiving data from and sending data to its dependent processes. After all generations are computed, each process sends its final sub-grid to the master thread, which in turn reconstructs the final 40,000 x 40,000 grid, using **MPI\_Gatherv**. After the grid is reassembled, parameter and timing data is displayed on screen and written to a .csv file.

Figure 1 below shows the comparison between calculation time and the number of processes used. Note: the time used for this graph only measures the parallel section; it does not include the setup and teardown time. As stated earlier, data for the calculation time for a single process could not be collected and was assumed to be twice the calculation time for 2 processes. The graph shows a near exponential decay in process time as expected.

Figure : Calculation Time vs. Process Count

Figure 2 and Figure 3 below show the speedup and efficiency achieved by increasing the process count respectively. The graphs show that the runs using 100 and 400 generations maintain a speedup efficiency close to 1 when using up to 16 processes. The sharp drop in efficiency for all runs when using 32 or more processes was likely due in part to having to share resources with other users on Tiny since data for these runs was collected closer to the due date of this assignment. At the end of this report is the spreadsheet that contains all of the data collected for this assignment. It shows a significant increase in the difference between the calculation time and the total execution time of the 2 runs with 200 generations and 8 and 16 processes (228 and 432 seconds compared to 45 – 60 seconds). If this was caused by a significant increase in the delay between intra-process message passing, it could explain why these two runs had a significant drop in efficiency compared to the runs with 100 and 400 generations. Overall, most runs maintained an efficiency of approximately 70% or better.

Figure : Speedup vs. Process Count

Figure : Effeciency vs. Process Count

