Alexander Wall

MPI: The Game of Morra

This project focuses on using the Message Passing Interface (MPI) to run a game of Morra on a cluster of machines. The program utilizes a single program, multiple data (SPMD) parallel technique where a single executable file is distributed among multiple processors to execute similar results. Each process generates two random numbers, the number of fingers to extend to the playing field and a guess for the total number of fingers. The MPI collects all the number of fingers from each process, reduces it by summation, and distributes the result back to the processes. Then, each process compares its guess to the result, and if there is a match, the process will score itself a temporary win for that round. Next, the MPI collects data of the temporary wins from each process, reduce the data to a summation, and sends the summation back to all the players. If the sum of the data is zero, then one process, the first process for example, would print a message saying that nobody had won the round. If the sum of the data is one, that means only one process had won the previous and would score a point for itself and output a message of its winning. If the sum of the data is more than one, then the processes that scored the right guess would output a message of an almost win. After all the rounds, the MPI performs a final tally by gathering all the scores of each processes and puts them on a scoreboard for all the processes to calculates their final placement.

One observation that I should mention when I was simulating the trials is the output of the print statements can tend to mangle with the other print statements when run on a large number of processes. Due to the time it takes to initialize MPI for all processes and the delay of transferring data to buffer and output the message to terminal, the messages produced by all processes will be out of order, i.e. messages from previous round will mingle with other rounds if the number of players is 500.

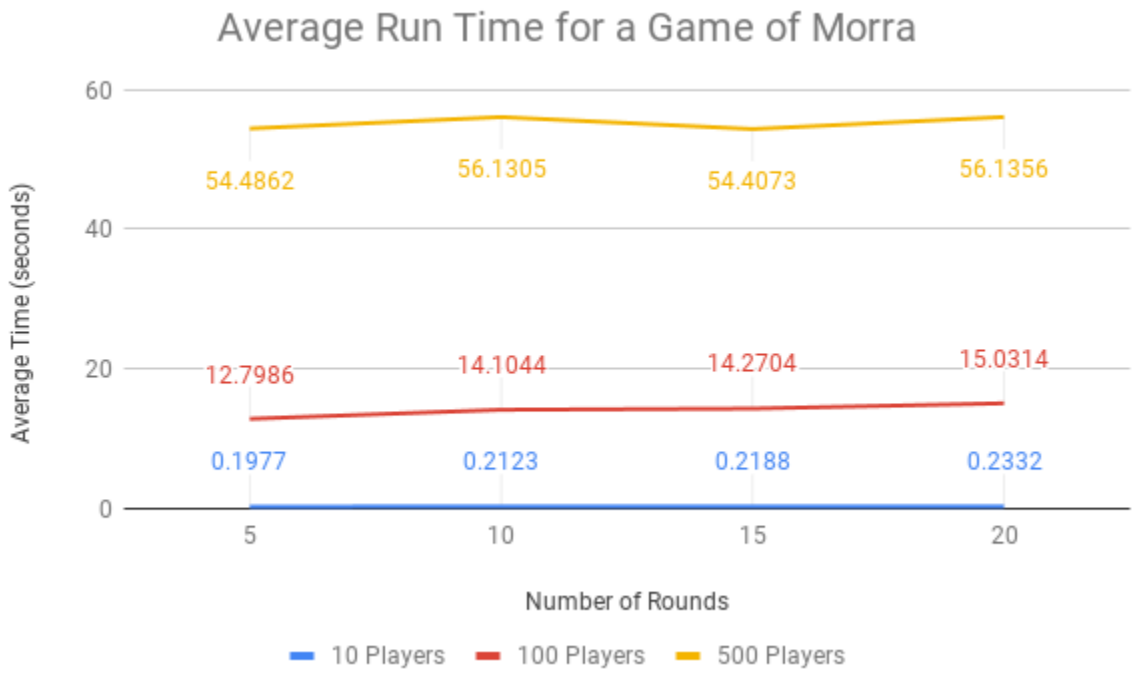
The program uses a broadcasting strategy to communicate the results between all processes. In the program, this strategy is seen when the MPI collectively gathers all the data from each process and distributes data back to the processes. Algorithmically, this approach of reducing and gathering is an effective method over the other communication types such as the ring strategy. According to this article on the optimization of the MPI collective functions, mathematicians at the Mathematics and Computer Science department at Argonne National Laboratory proposed methods of improving the runtime complexity by using two algorithms: doubling recursion, which exchanges data per processes based on the distance between them, and Bruck algorithm, which disperses data by sending data to one process and receiving from another process in a logarithmic passing. As a result, the reduce and gather methods, in the average case, takes about $\log(p)$, where p is the number of processes, to perform the functions with the algorithms described previously[1]. In comparison, the ring strategy would take $p - 1$ steps since data would be sent to the next process in a one-by-one procedure, thus making this approach inefficient.

The following observations have been made to analyze the runtime on the broadcast strategy with different numbers of players and rounds. Keep in mind that the time also reflects the procedure of creating the processes and MPI_Barrier times.

[1] https://www.mcs.anl.gov/~thakur/papers/ijhpca-coll.pdf (sec. 4-1 and 4-5)

Table 1 Average Time for a Number of Players to Play a Game of Morra with a Given Number of Rounds

Number of Players (Processes)	Number of Rounds	Time (seconds)
10	5	0.1977
	10	0.2123
	15	0.2188
	20	0.2332
100	5	12.7986
	10	14.1044
	15	14.2704
	20	15.0314
500	5	54.4862
	10	56.1305
	15	54.4073
	20	56.1356



The number of players is linearly proportional to the time it takes for the program to execute successfully. For example, 100 players take an overall average of 14 seconds, and 500 players take an average of 55 seconds, thus showing that the time it takes 1 process to execute the program is around 0.1 seconds. The number of rounds, in terms of the average times, illustrate no significant role to how the times are spread out evenly. For 500 players, the times are erratically dispersed between 54 and 56 seconds regardless on the increments of round. I did expect that it would take longer for the program to run when the number of players increases.