Project 2
The Game Of Morra

Prionti Nasir

This project runs a distributed program by means of a Message Passing Interface (MPI) to simulate the game of Morra.

The mechanism implemented in the program is outlined below:

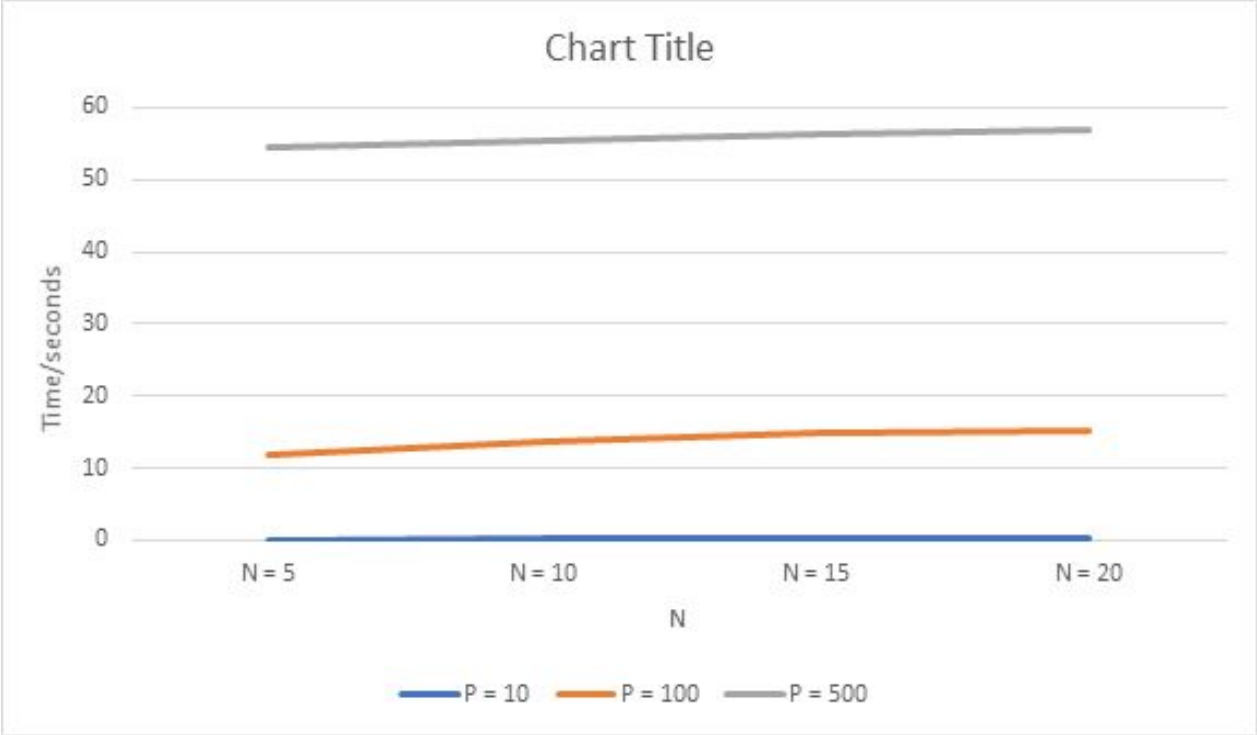
1. Each process produces two random numbers: the number of fingers it is extending, and a guess for the total number of fingers extended in the round.
2. The MPI collects all the number of fingers from each process, reduces them by summing the numbers, and relays the output back to the processes.
3. Each process compares the output to its guess and records whether it won a point for the round or not.
4. The MPI collects data of the temporary wins from each process, reduces them by summing the numbers, and relays the output back to the processes. Each process would then print a message declaring its score for the round (0 or 1).
5. After all the rounds, the MPI performs a tally by gathering all the scores of all the processes and produces a scoreboard/ranking.

Generally, since tasks are split up and run simultaneously on multiple processors with different inputs, the program runs an MIMD technique. More specifically, it actually utilizes a category of multiple instruction, multiple data (MIMD) technique, called the single program multiple data (SPMD) technique, where multiple processors operate on different inputs with the same executable.

In the program, the MPI gathers data from processes, processes the data and relays output back to the processes. This means the program is using a broadcasting strategy to communicate data between processes.A multicast strategy is, in theory, better than a broadcast because only intended receivers are communicated with. In the broadcast technique, data is transmitted to all the hosts connected to the network. So one can say a multicast would be better. However, since we don't have any receivers in the network of processors that we don't need to send messages to, i.e. all the processors involved are in fact intended receivers, in this case a multicast would be equivalent to a broadcast and so the maximum efficiency we can achieve in this program with the general technique of one-to-many communication (broadcast, multicast etc) has been achieved.

While executing a large number of processes, it was observed that the print messages are not always in the order that the guesses were placed. Sometimes there is also intermingling between other rounds while printing messages. One way to tackle this might be introduce synchronicity while using stdin. Also notable is the expected significant increase in speed when the program switched from 2 nodes to 4.

Temporal data in line plot and tabular format:



	P = 10	P = 100	P = 500
N = 5	0.2074	11.7699	54.4067
N = 10	0.2256	13.5055	55.5652
N = 15	0.2310	14.7787	56.3251
N = 20	0.2366	15.1205	56.9843

The pseudocode for the program is on the next page.

```
n = number of players
p = single player(processor)
if n < 2:
        print error message and exit
w= 0 if loss, 1 if win
total = total wins
for i=1 to rounds:
        f = number of fingers extended by p
        g = guess that a player makes for number of fingers for round
        w = 0
        MPI_reduce: get the sum of the number of fingers for round
        if sum of fingers == guess
                w = 1
        MPI_reduce: get the sum of all w's from each player
        if sum of winners == 0
                print that nobody won this run
        else if sum of winners == 1 and w == 1
                print player won
                total ++
        else if sum of winners > 1 and w == 1
                print player almost won
MPI_Allgather: scores[] hold the results of the number of wins to be relayed back
p = number of times w of player < w in scores[]
for j=1 to n:
        if w > scores[j]
                p++
print results
exit with success
```