# Parallel Programming HW2 Write Up
# Due Monday February 1, 2021, 11:59 pm

Connor Osborne
A01880782

Monday February 1, 2021

## 1 Description

For homework 2 I implemented an MPI program version of a pachinko game where each process simulates one column of pegs, so that a ball arriving at a process will be sent either left or right. When the ball arrives at the lowest level, its location is recorded. When the last ball hits the lowest level, the simulation ends. Processes at the extreme edges of the pachinko board always send the balls back toward the interior.

## 2 Implementation

The implementation starts with setting variables to keep track of how many balls there will be, what row level the ball is on, what the rank of the process is, and how many processes there are.

```
int numBalls = 1000;
int ball = 1;
int rank, size;
```

The next step was setting up the MPI instance and seeding the random number generator so that later on I would be able to have the ball pass randomly from column to column.

```
MPI_Status mystatus;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MCW, &rank);
MPI_Comm_size(MCW, &size);

srand(time(NULL));
```

The rest of the code for this assignment is split into two main sections for a master and slave system for the processes. Process 0 is set in charge of dropping balls to the center column process. It also waits to receive word that the ball it dropped made it to one of the bottom holding bins before dropping another. Once it receives this message it decreases the number of balls it has and increases the count for keeping track of how many balls are in the holding bin. I keep track of these results using vector so that the number of bins can depend on the number of processes. Once process 0 has stored the current state of the game it drops another ball. It continues this pattern until there are no balls left. Once this happens it sends a ball of value -1 to the other processes in order to end the game and stop all processes. Once this has happened it then prints the final results to the console.

```
if (rank == 0) {
    // Hopper drops ball to random peg
    vector<int> results;
    for (int i = 0; i < size; ++i) {
        results.push_back(0);
    }

    int peg = size / 2;

    MPI_Send(&ball, 1, MPI_INT, peg, 0, MCW);

    while (numBalls > 0) {
        MPI_Recv(&ball, 1, MPI_INT, MPI_ANY_SOURCE, 1, MCW, &mystatus);
        numBalls -= 1;
        results[mystatus.MPI_SOURCE] += 1;
        ball = 1;
        MPI_Send(&ball, 1, MPI_INT, peg, 0, MCW);
    }
    ball = -1;
    for (int i = 1; i < size; ++i) {
        MPI_Send(&ball, 1, MPI_INT, i, 0, MCW);
    }

    for (int i = 1; i < size; ++i) {
        cout << "Holder " << i << " has: " << results[i] << endl;
    }
}
```

The second major part of the code determines the actions of the slave processes or all processes other than 0. The current process will receive a ball and check to see if it has a value of -1. If it does the process will break out of the while loop, allowing the game to end. Next it will check to see if the ball's value is 5 and if it is it will send a message to process 0 saying the ball made it to the bottom. If the ball has any other value from 1 to 4 the process will increase the ball's value and then try to drop it down to the next level. The process will first check if it is the far left or far right column. If it is one of those columns then it will pass the ball to the next column towards the center.

```
else {
    //Columns
    while (1) {
        MPI_Recv(&ball, 1, MPI_INT, MPI_ANY_SOURCE, 0, MCW, &mystatus);

        if (ball == -1) {
            break;
        }
        else {
            if (ball == 5) {
                MPI_Send(&ball, 1, MPI_INT, 0, 1, MCW);
            }
            else {
                ball += 1;

                if (rank == 1) { // far left column
                    MPI_Send(&ball, 1, MPI_INT, 2, 0, MCW);
                }
                else if (rank == size - 1) { // far right column
                    MPI_Send(&ball, 1, MPI_INT, rank-1, 0, MCW);
                }
```

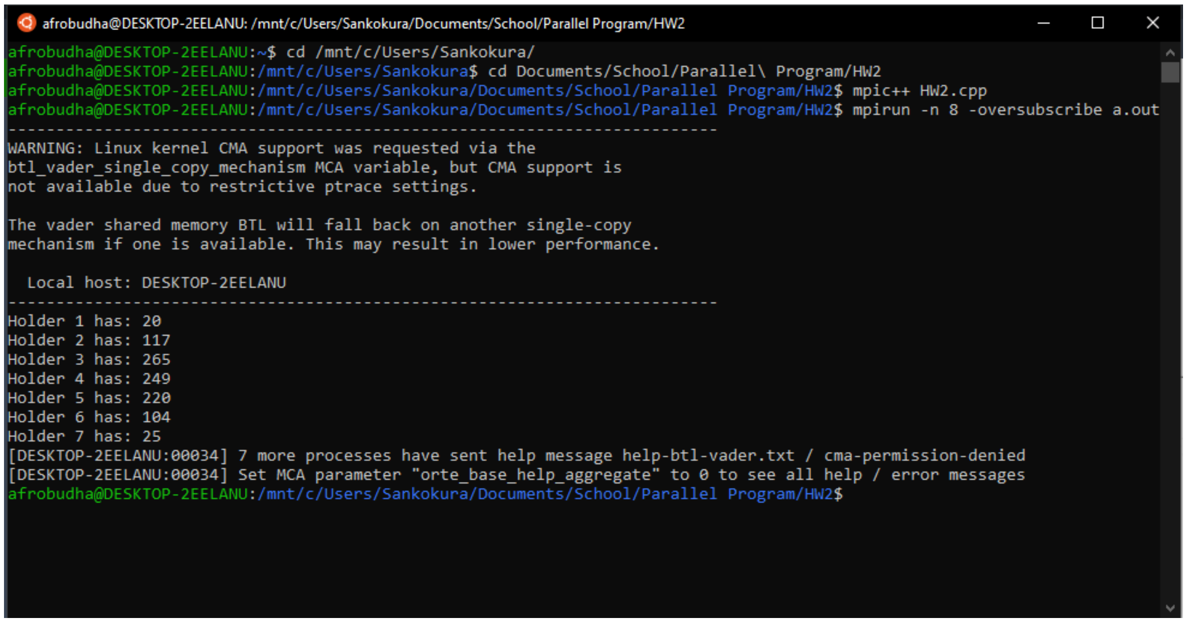
If the current process isn't one of the extreme edges it will go through a complex process to determine where the ball will go next randomly. This is how we end up with a normal distribution the more balls the program starts with. Because a pachinko board actually has a row of offset pegs below each row represented as a column the ball will have a 50/50 chance of going left or right from its current column but it then will have a 50/50 chance of being passed back to it's current row. To emulate this the code first gets a random number of 0 or 1 to decide if it's going to make it to a different column or stay in it's current one. Then we generate a random number of 0 or 1 in order to determine whether the ball would have gone left or right. Then based on the results the ball is sent to the corresponding process.

```
else { // determine whether the ball will move to different column then determine whether to
    pass to the left or right
                        int moveOrNot = rand()%2;
                        int move;

                        if (rand()%2 == 0){
                            move = 1;
                        }
                        else {
                            move = -1;
                        }
                        if (moveOrNot == 1) {
                            MPI_Send(&ball, 1, MPI_INT, rank+move, 0, MCW);
                        }
                        else {
                            MPI_Send(&ball, 1, MPI_INT, rank, 0, MCW);
                        }
                    }
```

## 3 Output

The output of my code is shown below.



```
afrobudha@DESKTOP-2EELANU: /mnt/c/Users/Sankokura/Documents/School/Parallel Program/HW2
afrobudha@DESKTOP-2EELANU:~$ cd /mnt/c/Users/Sankokura/
afrobudha@DESKTOP-2EELANU:/mnt/c/Users/Sankokura$ cd Documents/School/Parallel\ Program/HW2
afrobudha@DESKTOP-2EELANU:/mnt/c/Users/Sankokura/Documents/School/Parallel Program/HW2$ mpic++ HW2.cpp
afrobudha@DESKTOP-2EELANU:/mnt/c/Users/Sankokura/Documents/School/Parallel Program/HW2$ mpirun -n 8 -oversubscribe a.out
--------------------------------------------------------------------------
WARNING: Linux kernel CMA support was requested via the
btl_vader_single_copy_mechanism MCA variable, but CMA support is
not available due to restrictive ptrace settings.

The vader shared memory BTL will fall back on another single-copy
mechanism if one is available. This may result in lower performance.

  Local host: DESKTOP-2EELANU
--------------------------------------------------------------------------
Holder 1 has: 20
Holder 2 has: 117
Holder 3 has: 265
Holder 4 has: 249
Holder 5 has: 220
Holder 6 has: 104
Holder 7 has: 25
[DESKTOP-2EELANU:00034] 7 more processes have sent help message help-btl-vader.txt / cma-permission-denied
[DESKTOP-2EELANU:00034] Set MCA parameter "orte_base_help_aggregate" to 0 to see all help / error messages
afrobudha@DESKTOP-2EELANU:/mnt/c/Users/Sankokura/Documents/School/Parallel Program/HW2$
```