CS 5500 HW1 : MPI

Marshal Taylor

January 26, 2022

Abstract

A concise report describing my implementation of Parallel Pachinko

1 Introduction

To keep the number of parameters passed between each of the processors as small as possible, I opted to merge the layer that the ball was on, and the iteration of the simulation into a single variable. I multiply the number of iterations by the number of layers to get the total layers in the whole simulation. Essentially, I have created a really really really long pachiko board. Every time the ball hits a layer that should be the end of an iteration, we record where the ball landed in a text file. Then, based on command line parameters, we either pick the ball up and set it into a specific column, or out it into a random column. This happens continuously until the ball reaches layer 0.

2 Commands

1) Compile: VS code Build

```
2)mpiexec -np {number of processsors}
.\HW2_Parallel_Pachinko.exe {layers} {iterations} {starting_processor}
```

3) optional parameters: 0 GT starting processor LT number of processors Don't pass a parameter, or pass an invalid parameter for the program to randomly select a new column each iteration.

3 Implementation

1) To start the simulation, we need to start an iteration. This method will start an iteration, decrement the layer, and log the layer that we ended it. This method will process the command line argument to see where it should start the next ball. It will either start it randomly, or in a specified column.

```
int newIteration(int layer, int new_game_column) {
    //cout << layer << "New iteration" << endl;

    int rank, size;
    MPI_Comm_rank(MCW, &rank);
    MPI_Comm_size(MCW, &size);
    int dest;
    layer--;
    dest = rand() % size;

    //if new_game_column is valid, use it. Else, randomly pick in the while loop
    if (new_game_column >= 0 && new_game_column < size) dest = new_game_column;

    while (dest == rank) {
        dest = rand() % size;
```

```
    }
    //cout << dest << endl;
    cout << rank << ": " << "boom!" << endl;

    // log data
    string filename("data.txt");
    fstream file;
    file.open(filename, std::ios_base::app | std::ios_base::in);
    if (file.is_open())
        file << rank << endl;

    MPI_Send(&layer, 1, MPI_INT, dest, 0, MCW);
    return 0;
}
```

2) This method ends the simulation. It tells all processors that the simulation has ended, and we can finalize the MPI program.

```
int endSimulation(int layer) {

    int rank, size;
    MPI_Comm_rank(MCW, &rank);
    MPI_Comm_size(MCW, &size);

    layer--;
    for (int i = 0; i < size; ++i) {
        if (i != rank) {
            MPI_Send(&layer, 1, MPI_INT, i, 0, MCW);
        }
    }
    return 0;
}
```

3) In pachiko, we need to send the process to a new processor to the left or to the right of my current position. sendDirection is the method we call to send that process over. If a process is send to a processor that doesn't exists, we bounce it the other direcction instead.

```
int sendDirection(int layer) {

    int rank, size;
    int dest;
    MPI_Comm_rank(MCW, &rank);
    MPI_Comm_size(MCW, &size);

    bool valid = false;
        //cout << rank << ": " << "received the potato." << endl;
    dest = rank;
    int direction = rand() % 2;
    if (direction == 0) dest--; else dest++;
    layer--;
        //cout << layer << ": " << "layer." << endl;
    if (dest >= 0 && dest < size) {

    }
    else {
        //bounce back to center
        if (dest < 0) dest += 2;
        if (dest >= size) dest -= 2;
```

```
    }

    MPI_Send(&layer, 1, MPI_INT, dest, 0, MCW);

    return 0;
}
```

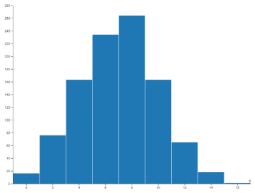
4) Finally, the main loop. It calls all other methods when needed. It processes command line arguments, creates and removes the log file. Kicks off MPI, and processes what we nee to do at each layer of the iteration.

```
int main(int argc, char** argv) {
    int rank, size;
    int layer;
    int layers_per_game = atoi(argv[1]);
    int dest;
    int iterations = atoi(argv[2]);
    int new_game_column = -1;

    remove("data.txt");
    if (argc > 3) new_game_column = atoi(argv[3]);
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MCW, &rank);
    MPI_Comm_size(MCW, &size);

    if (!rank) {
        layer = (layers_per_game * iterations) + (layers_per_game-1);
        newIteration(layer, new_game_column);
    }
    while (1) {
        MPI_Recv(&layer, 1, MPI_INT, MPI_ANY_SOURCE, 0, MCW, MPI_STATUS_IGNORE);
        //Sleep(10);
        if (layer <= 0) {
            endSimulation(layer);
            break;
        }
        else if (layer % layers_per_game == 0) {
            //cout << "new: "<< layer << endl;
            newIteration(layer, new_game_column);
        }
        else {
            sendDirection(layer);
        }
    }
    cout << rank << ": " << "done." << endl;
    MPI_Finalize();
    return 0;
}
```

3.1 Normal Distribution

Command to reproduce this

```
mpiexec -np 16 .\HW2_Parallel_Pachinko.exe 10 1000 7
```



Graph built with https://rawgraphs.io/