- Middle East Technical University

Department of Computer Engineering

CENG 478
Introduction To Parallel Computing
Spring 2019-2020

Report of Assignment 2

Mahmoud Alasmar
2126027

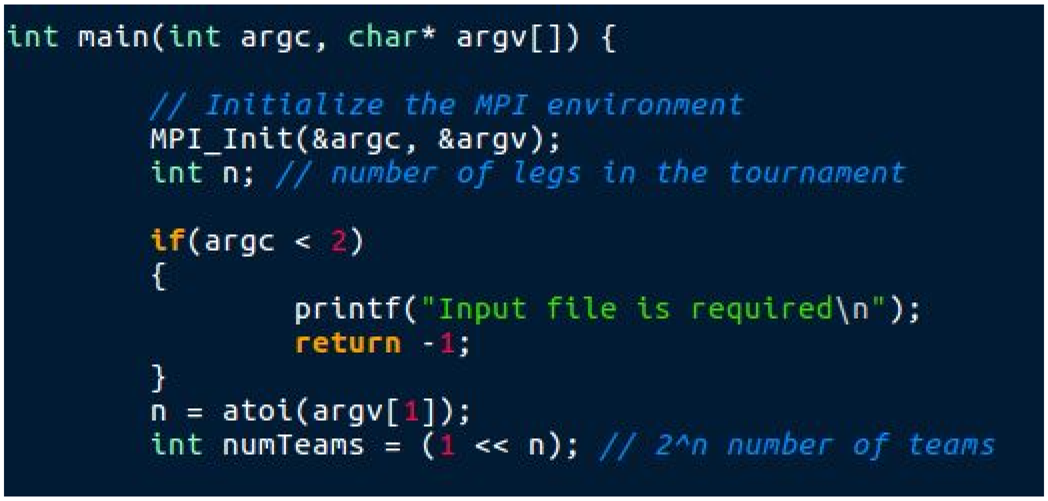
Introduction

Within the framework of this assignment, a parallel algorithm is developed for tournament simulation, the utilization of MPI library for such type of application is deployed and the total time of the tournament with respect to the degree of parallelism is investigated.

Algorithm Description

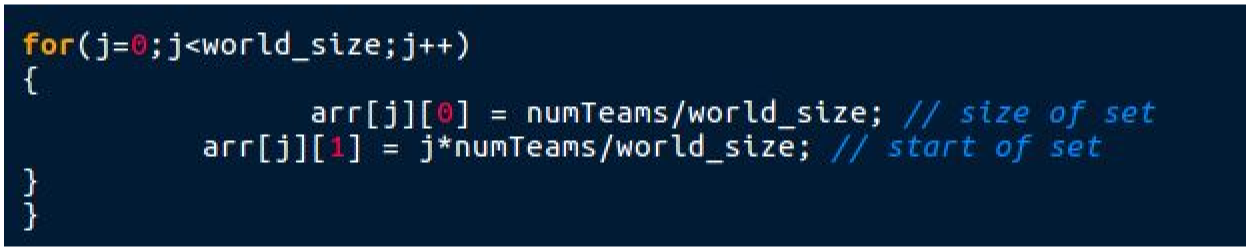
The algorithm consists of two stages, the first stage of the algorithm is performed in parallel where a number of processes are working in parallel in order to simulate the tournament for a number of teams, the second stage of the algorithm is done using only a single process, at this stage, the tournament will be simulated with the remaining teams (that are qualified after first stage) and a single winner will be determined.

- **Initialization**



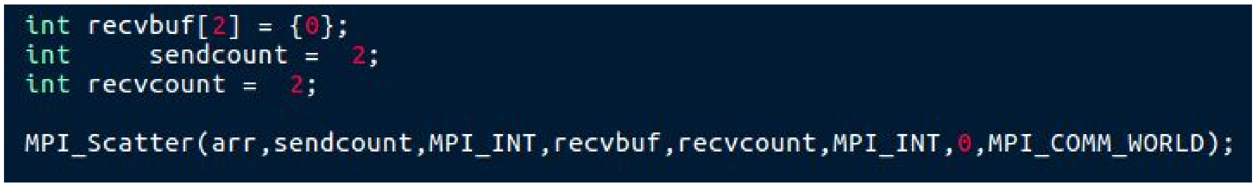
```c
int main(int argc, char* argv[]) {

        // Initialize the MPI environment
        MPI_Init(&argc, &argv);
        int n; // number of legs in the tournament

        if(argc < 2)
        {
                printf("Input file is required\n");
                return -1;
        }
        n = atoi(argv[1]);
        int numTeams = (1 << n); // 2^n number of teams
```

The initialization step involves determining the number of teams, the application accepts argument "n" which is the total legs of the tournament, number of teams is 2^(n). After determining the number of teams, process 0 will prepare an array having a size of 64 elements (maximum number of processes that can simulate the tournament), each element of the array will consist of two components, namely, size and offset.



```c
for(j=0;j<world_size;j++)
{
                arr[j][0] = numTeams/world_size; // size of set
          arr[j][1] = j*numTeams/world_size; // start of set
}
}
```

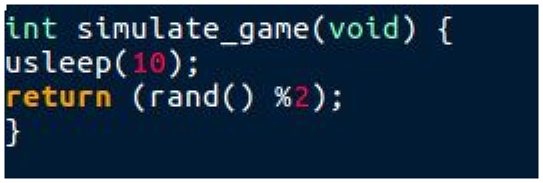
The size represent the size of the set (number of teams) that a single process will work on and the offset represent the index of the first team on that set, for example, if there are 32 teams and 4 processes, then process 0 will work on a set of size 8 teams starting from team 0, process 1, will work on a set of same size but starting with team 8, and so on. Once the setup of that array is done by process 0, it will distribute the elements of that array on the available processes, the size of the message is 2*sizeof(int), (offset and size each of type int), each process will receive exactly a single element. The operation of distributing the work among processes is done using the function **MPI_Scatter** as shown below:



```c
int recvbuf[2] = {0};
int     sendcount =  2;
int recvcount =  2;

MPI_Scatter(arr,sendcount,MPI_INT,recvbuf,recvcount,MPI_INT,0,MPI_COMM_WORLD);
```

- **First Stage**

At this point, all available processes have received the data set that each will simulate. Each process has received exactly (Total_number_teams / Total_number_processes) number of teams, each team with index (2i + offset) will play against team with index (2i + 1 + offset), where "offset = process_rank*(Total_number_teams / Total_number_processes)", and "i" is an integer equal to 0,1,2,3,4….((Total_number_teams / Total_number_processes)/2 - 1). At the end of the first stage simulation each process will yield a single winner team. The simulation of each match is handled using the following function:



```
int simulate_game(void) {
usleep(10);
return (rand() %2);
}
```

The above function will return either 0 or 1, 0 means the winner is the team with index (2i + offset) and 1 means the winner is the team with index (2i + 1 + offset). The following pseudocode summarises the process of the first stage:

```
Rem = (Total_number_teams / Total_number_processes)
While ( Rem != 1)
{
Index = 0;
For i = 0:2:Rem-1
{
Teams_array[index] = Teams_array[i + simulate_game()];
Index++;
}
Rem = (Rem >> 1) %shift to right one bit
}
%Teams_array : array that holds the set of teams each process is responsible for.
```

The complexity of the first stage is

$$\frac{n}{2\times p} \times \frac{1-1/2^{Log(n/p)}}{1-1/2}$$

This number represent the total number of matches to be played in first stage, where
n: total number of teams
p: total number of process

At the end of this stage all processes send their winners to a single process using the function **MPI_Gather.**

- **Second Stage**

The second stage of the algorithm is a straight forward sequential simulation of the tournament using only a single process, the following pseudocode summarises the process of the second stage:

```
Rem = total_number_processes;
Index = 0;
while(Rem != 1)
{
For i = 0:2:Rem-1
Teams_array[index] = Teams_array[i + simulate_game()];
Index++;
}
Rem = (Rem >> 1) %shift to right one bit
}
%Teams_array : array that holds the winner teams of first stage
```

The complexity of this stage is

$$\frac{p}{2} \times \frac{1-1/2^{Log(p)}}{1-1/2}$$

This is the number of matches to be played at this stage, where
p: total number of processes.

Total complexity of the algorithm is

$$\frac{n}{2\times p} \times \frac{1-1/2^{Log(n/p)}}{1-1/2} + \frac{p}{2} \times \frac{1-1/2^{Log(p)}}{1-1/2}$$

n: total number of teams
p: total number of process

Results:

4 cases are tested, case number of processes is equal to 1,4,8 and 16. The case when the number of processes is equal to 1 will be taken as the sequential case. Two metrics are used for comparison, execution time and speed improvement.

Speed improvement = execution sequential time / execution time in parallel

Within this test number of legs is chosen as 16, so the number of teams is 2^(16).

Number Of Processes	Execution Time (seconds)	Speed Up
1	4.38785	1
4	1.09839	3.9948014822
8	0.56348	7.7870554412
16	0.32622	13.4505854945

Table 1: Numerical Results

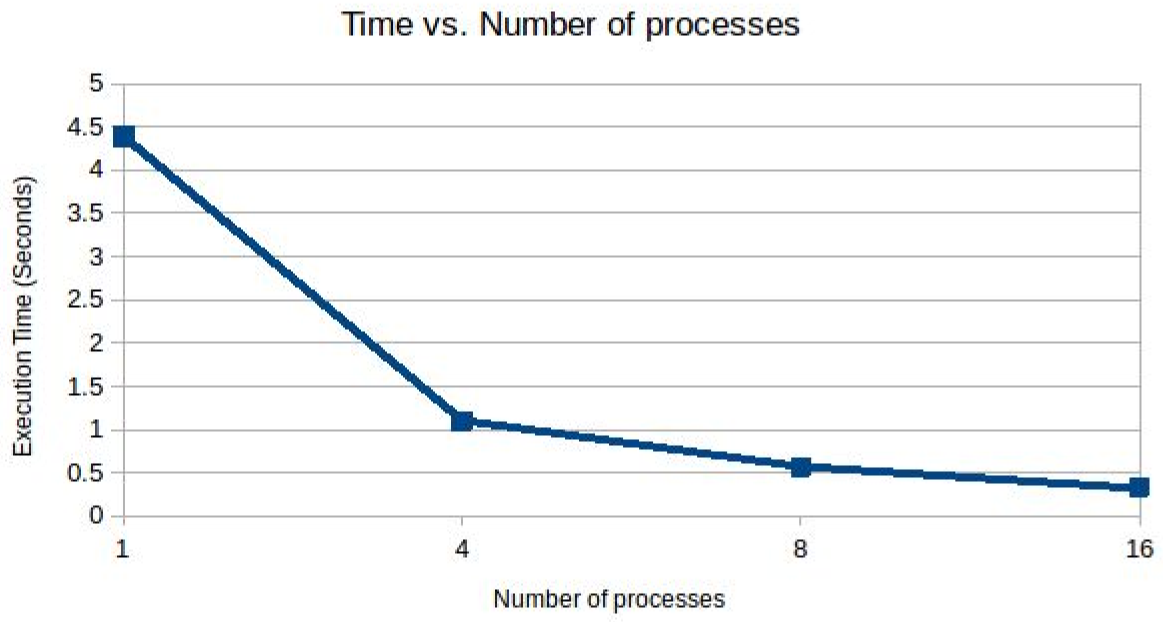


Figure 1: Execution time vs. number of processes

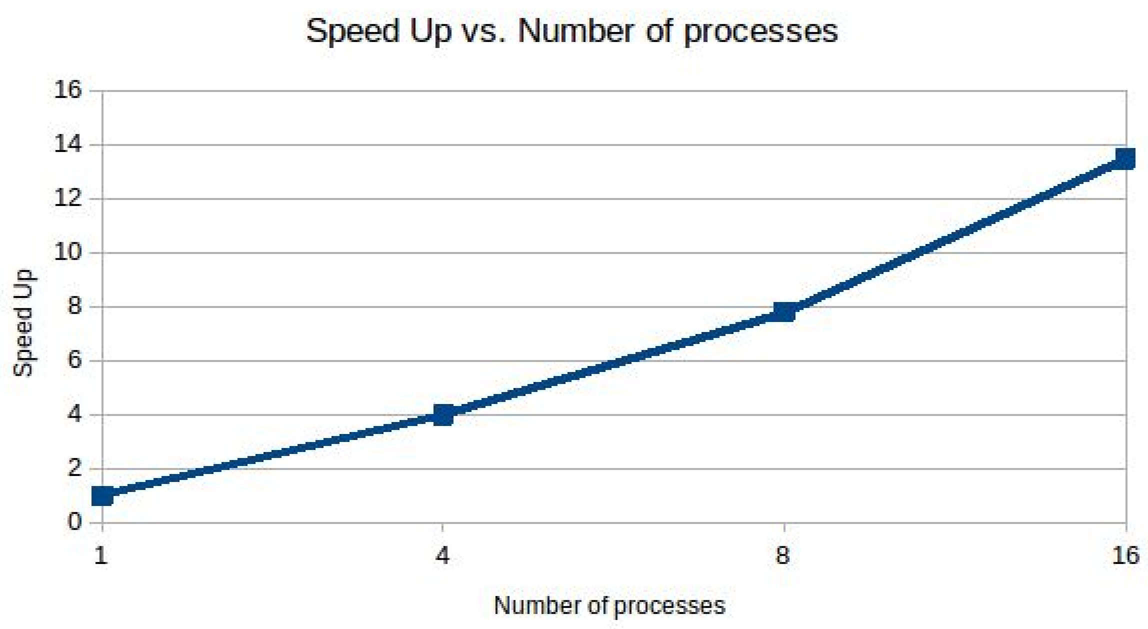


Figure 2: Speed improvement vs. number of processes

Obtained results show that the execution time decreases with increasing number of processes and the speed improvement increases as the number of processes increases, this is expected since as the number of processes increases more parts of the tournament will be simulated in parallel. There is also an observation that as the number of processes increases the slope of the speed improvement graph decreases, this is expected since as more processes are involved the overhead communication starts to be significant which may reduce the speed improvement, moreover, as the number of processes increases further by the definition of the algorithm there is an expectation for decrease in speed up and increase in execution time, because when the number of processes is large, then number of matches for a single process in the first stage of the algorithm decreases and in the second stage (that is done in sequential) large number of matches have to be simulated .

The Algorithm can be improved by deploying parallelism in the second stage of the algorithm, meaning the matches of the second stage can be simulated in parallel but with less number of processes than the first stage, and further stages can be added for parallel execution until there is only two teams where a single process can handle their match simulation.

Compile and Run:

To build the output file you can simply type “make” on terminal and the output file “hw2” will be created. To run the code type “sbatch test.sh” , note, to specify number of processes modify #SBATCH --nodes= and -n arguments on test.sh file.