GAZİ UNIVERSITY
ENGINEERING FACULTY
COMPUTER ENGINEERING

BM479E PARALLEL COMPUTER ARCHITECTURES AND PROGRAMMING

Estimating the Value of PI by Monte Carlo Method Using Message Passing Interface in C Programming Language

Abstract

In this assignment, value of pi has been calculated using Monte Carlo method. MPI has been used for parallelizing the code. A little benchmarking has also been presented for performance measurement issues.

141180001
Abdullah Akalın
abdullah.akalin@gazi.edu.tr

31.10.2017

Contents

1 Introduction

Monte Carlo method states that, given a square located at the origin –assuming lower left corner is placed at (0,0) coordinates– which has an arc in it (one-forth of a whole circle) as in the Figure 1, the proportion of dots inside the arc to total dots randomly put in this square area is equal to pi/4. [1]

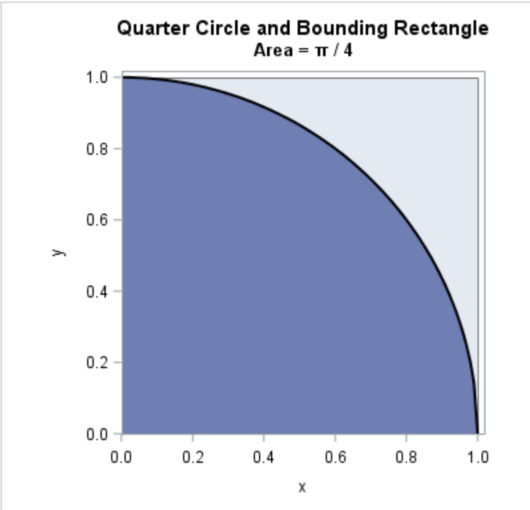

Figure 1: A unit square placed at the origin. [2]

1.1 Parallelization

In this problem, putting randomly generated dots inside the square area is an independent job. Hence, it can be done simultaneously by any number of processors. After putting all the dots defined by the user, the value of pi can be readily estimated.

2 Application

2.1 Program

In this program, I have used C as programming language and MPI for parallelization. The program begins with necessary pre-processor commands as listed below:

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <time.h>
```

```
#include <math.h>

```

After that, in the `main` function, I have defined necessary variables.

- pid: Holds the current processor number.
- nop: Holds the total number of processors.
- TOTAL_NUMBER_OF_POINTS: Holds the total number of dots that will be plotted in the square area.
- inside_points: Holds points that are placed inside the arc.
- current_total: Holds the number of points placed so far.
- gl_inside_points and gl_current_total: Hold cumulative values of the upper ones without "gl" prefix.
- pi: Holds estimated pi value.
- time_measurement: Holds the elapsed time.

```
int pid;
int nop;
const int TOTAL_NUMBER_OF_POINTS = atoi(argv[1]);
int inside_points = 0;
int current_total = 0;
int gl_inside_points;
int gl_current_total;
double pi;
double time_measurement;

```

Below, MPI initialization procedures are being done and necessary information is being gathered such as rank and size. Also a barrier has been set for synchronizing in time measurement.

```
MPI_Init(&argc, &argv);
MPI_Barrier(MPI_COMM_WORLD);
time_measurement = - MPI_Wtime();
MPI_Comm_rank(MPI_COMM_WORLD, &pid);
MPI_Comm_size(MPI_COMM_WORLD, &nop);

```

The parallel part of the program. After seeding the rand function, every processor runs their own portion of the loop and puts dots. If the magnitude of the vector (x,y) is less than 1, then the dot is inside the arc. In this case the inside point counter is incremented.

```
        srand(pid);

        for(int i = pid; i < TOTAL_NUMBER_OF_POINTS; i += nop)
        {
            double random_x = (double)rand() / (double)RAND_MAX;
            double random_y = (double)rand() / (double)RAND_MAX;
            double magnitude = random_x * random_x + random_y *
    random_y;

            if(magnitude <= 1)
                inside_points++;
            current_total++;
        }

```

Each processor adds its own number of dots to the global variables which will be readible for processor 0. Also, it is now okey to calculate the estimated time and finalize MPI.

```
        MPI_Reduce(&inside_points, &gl_inside_points,
            1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
        MPI_Reduce(&current_total, &gl_current_total,
            1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

        time_measurement += MPI_Wtime();
        MPI_Finalize();

```

If no more dots are being waited, then calculate pi and print necessary information to `stdout`.

```
        if(gl_current_total == TOTAL_NUMBER_OF_POINTS)
        {
            pi = 4 * (double)gl_inside_points / (double)
    gl_current_total;
            printf("%d,%d,%f,%f\n", nop, TOTAL_NUMBER_OF_POINTS,
    time_measurement, pi);
        }

```

2.2 Scripts

These scripts are written for analyzing the performance of the program. The outputs are simply forwarded to csv files to plot the graphs.

This script runs the program 100000000 times starting with 1 dot to 1.000.000.000 dots, using 4 processors.

```
#!/bin/bash

echo "ProcessorCount, PointCount, ElapsedTime, Pi"
for (( i=1; i<=1000000000; i*=10 ));
do
    mpirun -np 4 ./pi $i
done
```

This script runs the program 8 times starting with 1 processor to 8 processors, 1.000.000.000 dots for each.

```
#!/bin/bash

echo "ProcessorCount, PointCount, ElapsedTime, Pi"
for i in `seq 1 8`;
do
    mpirun -np $i ./pi $1
done
```

3 Analysis and Benchmarking

The program has been run at full capacity. Can be seen in Figure 2.

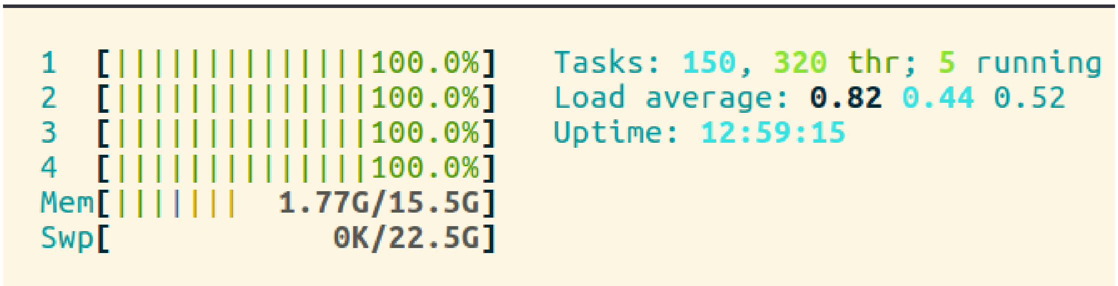

Figure 2: CPU usage.

Output of processes can be seen in Figure 3.
Estimation of pi by number of points is shown in Figure 4.
Performance measurement can be seen in Figure 5.

```
Every 2,0s: ps aux | grep ./pi | grep -v grep                Tue Oct 31 08:15:56 2017

zamma     10082  0.0  0.0  16232   972 pts/2    S+   08:15   0:00 mpirun -np 4 ./pi
zamma     10084 94.5  0.0  42808  7628 ?        Rs   08:15   0:03 ./pi 1000000000
zamma     10085 90.5  0.0  42808  7616 ?        Rs   08:15   0:03 ./pi 1000000000
zamma     10086 92.7  0.0  42808  7652 ?        Rs   08:15   0:03 ./pi 1000000000
zamma     10087 95.2  0.0  42808  7616 ?        Rs   08:15   0:03 ./pi 1000000000
```

Figure 3: Running 4 instances.

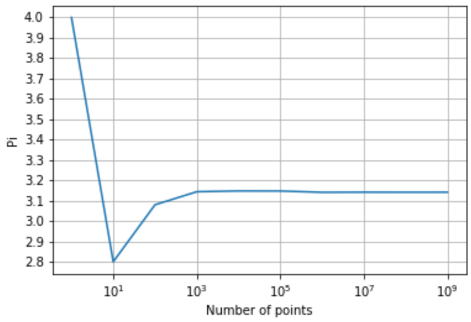

Figure 4: Pi convergence with respect to number of dots placed.

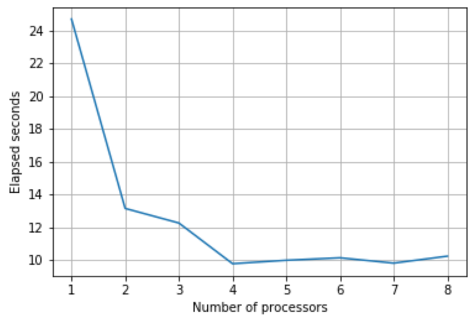

Figure 5: Performance measurement by time.

A Source Code

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <time.h>
```

```
#include <math.h>

int main(int argc, char** argv)
{
    int pid;
    int nop;
    const int TOTAL_NUMBER_OF_POINTS = atoi(argv[1]);
    int inside_points = 0;
    int current_total = 0;
    int gl_inside_points;
    int gl_current_total;
    double pi;
    double time_measurement;

    MPI_Init(&argc, &argv);
    MPI_Barrier(MPI_COMM_WORLD);
    time_measurement = - MPI_Wtime();
    MPI_Comm_rank(MPI_COMM_WORLD, &pid);
    MPI_Comm_size(MPI_COMM_WORLD, &nop);

    srand(pid);

    for(int i = pid; i < TOTAL_NUMBER_OF_POINTS; i += nop)
    {
        double random_x = (double)rand() / (double)RAND_MAX;
        double random_y = (double)rand() / (double)RAND_MAX;
        double magnitude = random_x * random_x + random_y * random_y;

        if(magnitude <= 1)
            inside_points++;
        current_total++;
    }

    MPI_Reduce(&inside_points, &gl_inside_points, 1, MPI_INT, MPI_SUM,
      0, MPI_COMM_WORLD);
    MPI_Reduce(&current_total, &gl_current_total, 1, MPI_INT, MPI_SUM,
      0, MPI_COMM_WORLD);

    time_measurement += MPI_Wtime();
    MPI_Finalize();

    if(gl_current_total == TOTAL_NUMBER_OF_POINTS)
    {
        pi = 4 * (double)gl_inside_points / (double)gl_current_total;
        printf("%d,%d,%f,%f\n", nop, TOTAL_NUMBER_OF_POINTS,
    time_measurement, pi);
    }

    return 0;
```

```
}
```

B Scripts

```
#!/bin/bash

echo "ProcessorCount, PointCount, ElapsedTime, Pi"
for (( i=1; i<=1000000000; i*=10 ));
do
    mpirun -np 4 ./pi $i
done
```

```
#!/bin/bash

echo "ProcessorCount, PointCount, ElapsedTime, Pi"
for i in `seq 1 8`;
do
    mpirun -np $i ./pi $1
done
```

C Script Outputs

```
ProcessorCount, PointCount, ElapsedTime, Pi
4,1,0.000092,4.000000
4,10,0.000054,2.800000
4,100,0.000057,3.080000
4,1000,0.000064,3.144000
4,10000,0.000221,3.148000
4,100000,0.000984,3.147600
4,1000000,0.010109,3.141164
4,10000000,0.093073,3.141704
4,100000000,0.953610,3.141455
4,1000000000,9.553220,3.141574
```

```
ProcessorCount, PointCount, ElapsedTime, Pi
1,1000000000,24.716689,3.141604
2,1000000000,13.166669,3.141539
3,1000000000,12.276997,3.141592
4,1000000000,9.795310,3.141574
5,1000000000,10.004150,3.141580
6,1000000000,10.155986,3.141686
7,1000000000,9.829380,3.141711
8,1000000000,10.254730,3.141695
```

References

[1] “Introduction to parallel computing.” https://computing.llnl.gov/tutorials/parallel_comp/#ExamplesPI. Erişim tarihi: 31.10.2017.

[2] “Monte carlo estimates of pi and an important statistical lesson - the do loop.” https://blogs.sas.com/content/iml/2016/03/14/monte-carlo-estimates-of-pi.html. Erişim tarihi: 31.10.2017.