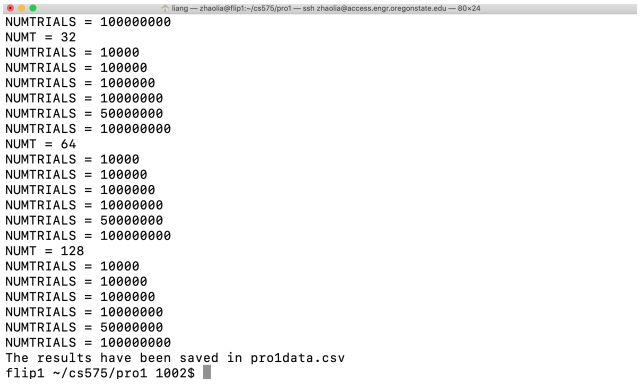
CS 475/575 -- Spring Quarter 2020

Project #1

OpenMP: Monte Carlo Simulation

100 Points

Due: April 15

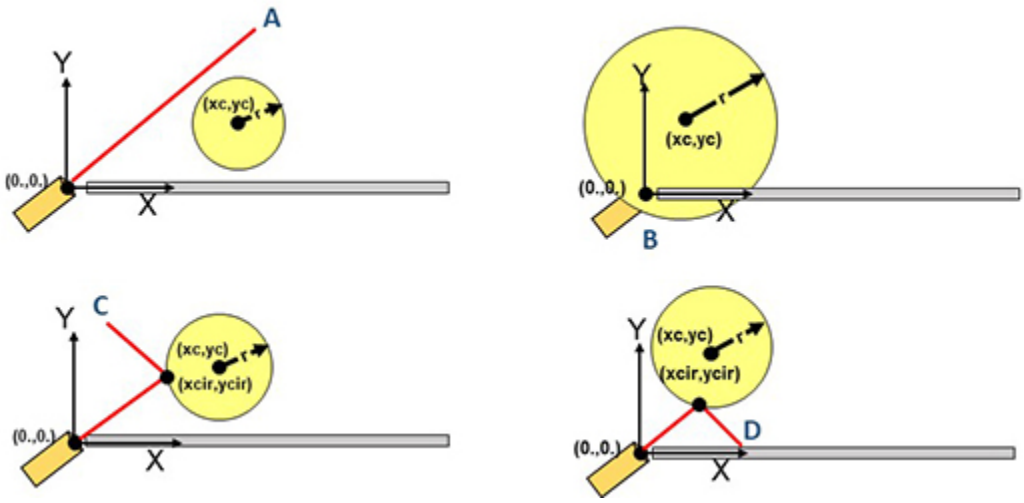


Liang Zhao

933-667-879

zhaolia@oregonstate.edu

Hi, this project is a job for multicore Monte Carlo simulation to determine whether the laser reaches the board below. The four situations that will appear in this experiment are shown in the figure below.



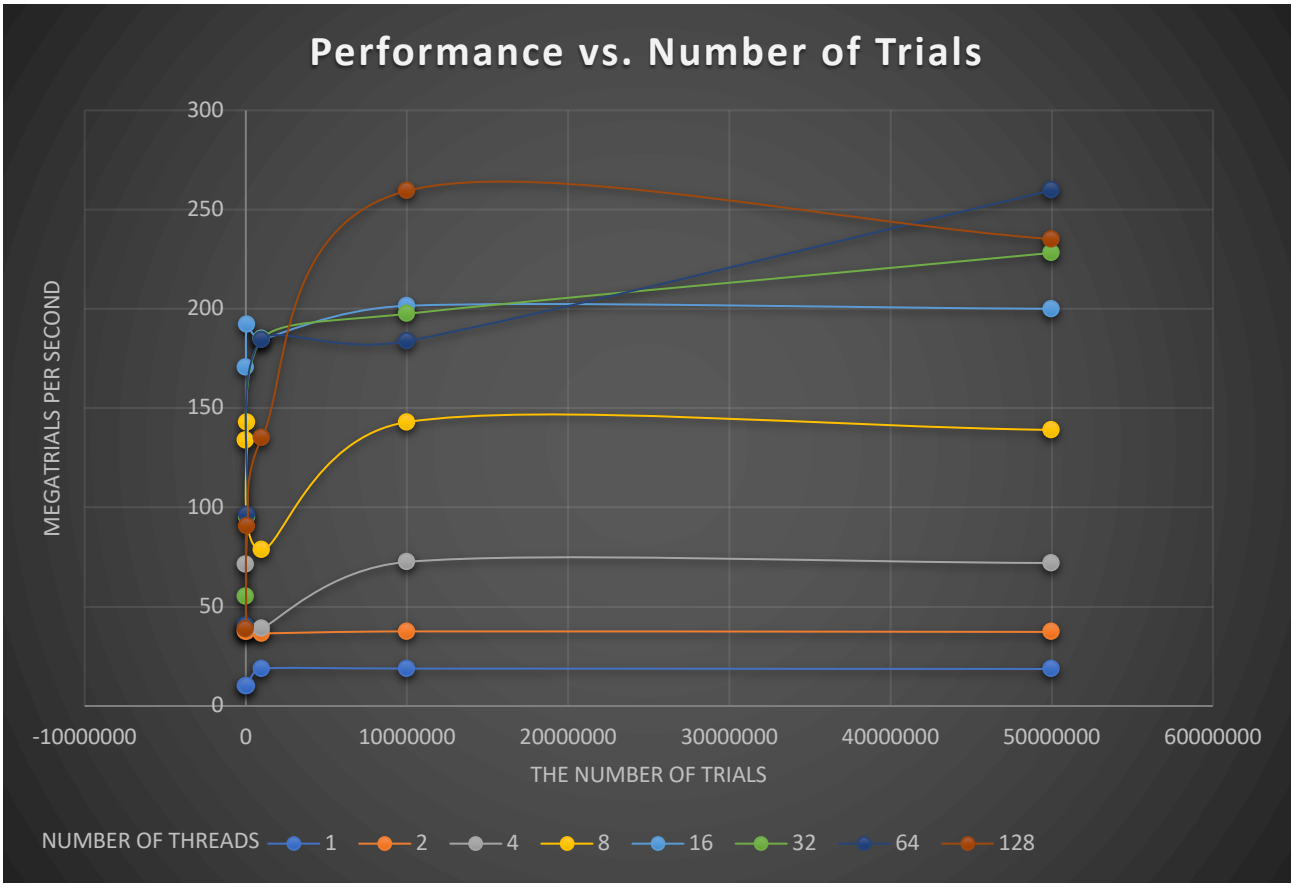
My program run on OSU Linux server: access.engr.oregonstate.edu. I used a bash script to automatically adjust the number of threads and trials, and save the results in a csv file. Let us first look at the results of the experiment.

Number of Threads	Number of Trials	Probability of Hitting the Plate	Megatrials per Second
1	10000	0.1275	10.041
1	100000	0.13093	9.951
1	1000000	0.130953	18.841
1	10000000	0.1309926	18.851
1	50000000	0.1310209	18.651
2	10000	0.1344	37.512
2	100000	0.13214	37.352
2	1000000	0.131103	36.572
2	10000000	0.1309543	37.572
2	50000000	0.1309785	37.302
4	10000	0.1305	71.424
4	100000	0.13111	39.134
4	1000000	0.131135	39.094
4	10000000	0.1311279	72.564

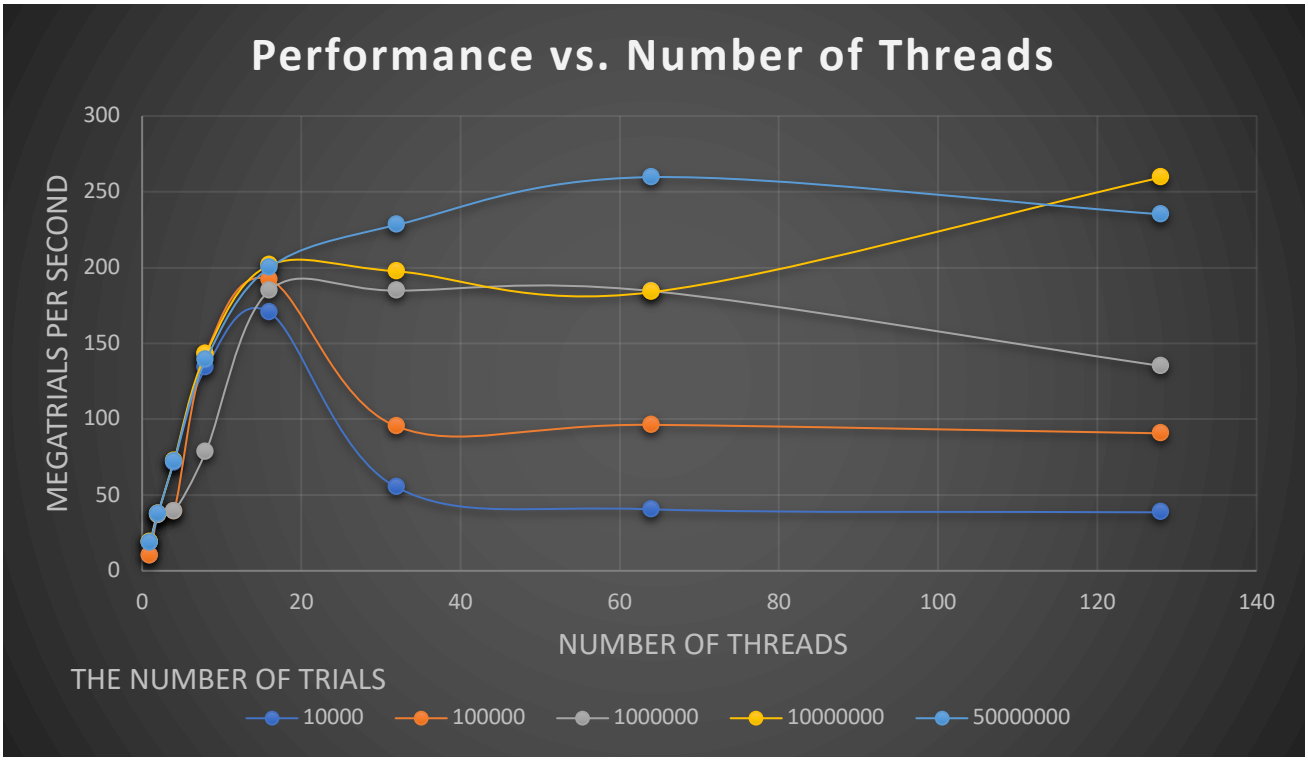
4	50000000	0.1310541	72.034
8	10000	0.1303	133.948
8	100000	0.13013	142.968
8	1000000	0.130705	78.628
8	10000000	0.1311348	142.988
8	50000000	0.1310253	139.058
16	10000	0.1348	170.4116
16	100000	0.13119	192.0416
16	1000000	0.130602	184.9216
16	10000000	0.1309378	201.5716
16	50000000	0.1310669	200.0416
32	10000	0.132	55.0032
32	100000	0.13152	95.2232
32	1000000	0.130727	184.8232
32	10000000	0.1309973	197.5432
32	50000000	0.131013	228.2132
64	10000	0.1285	40.5964
64	100000	0.13038	96.3064
64	1000000	0.130649	184.5364
64	10000000	0.1310518	183.7964
64	50000000	0.1309927	259.6964
128	10000	0.1266	38.59128
128	100000	0.13097	90.69128
128	1000000	0.130838	135.11128
128	10000000	0.1312106	259.48128
128	50000000	0.1310105	235.15128

(1): According to the actual number of random tests, the more the closer to the theoretical probability, so I think the actual probability is 13%.

(2): The graph of performance vs. number of trials



(3) The graph of performance vs. number of threads



(4)

Threads	10000 Trials	100000 Trials	1000000 Trials	10000000 Trials	50000000 Trials
1	10.041	9.951	18.841	18.851	18.651
2	37.512	37.352	36.572	37.572	37.302
4	71.424	39.134	39.094	72.564	72.034
8	133.948	142.968	78.628	142.988	139.058
16	170.4116	192.0416	184.9216	201.5716	200.0416
32	55.0032	95.2232	184.8232	197.5432	228.2132
64	40.5964	96.3064	184.5364	183.7964	259.6964
128	38.59128	90.69128	135.11128	259.48128	235.15128

I choose the commonly used 8 threads and 1 thread for speed comparison base on 50000000 Trials.

Speedup, S = (Execution time with one thread) / (Execution time with four threads) = (Performance with eight threads) / (Performance with one thread)

float Fp = (8./7.)*(1. - (1./S));

So, S = (139.058/18.651) = 7.46

Fp = (8/7)*(1-(1/7.46)) = 0.9896

```c
#define _USE_MATH_DEFINES
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>
#include <stdio.h>

// setting the number of threads:
#ifndef NUMT
#define NUMT 64
#endif

// setting the number of trials in the monte carlo simulation:
#ifndef NUMTRIALS
#define NUMTRIALS   10000
#endif

// how many tries to discover the maximum performance:
#ifndef NUMTRIES
#define NUMTRIES 10
#endif


// ranges for the random numbers:
const float XCMIN = -1.0;
const float XCMAX =  1.0;
const float YCMIN =  0.0;
const float YCMAX =  2.0;
const float RMIN  =  0.5;
const float RMAX  =  2.0;


// function prototypes:
float       Ranf( float, float );
int         Ranf( int, int );
void        TimeOfDaySeed( );



// main program:
int
main( int argc, char *argv[ ] )
{

#ifndef _OPENMP
```

```
    fprintf( stderr, "No OpenMP support!\n" );
    return 1;
#endif


    float tn = tan( (M_PI/180.)*30. );

    TimeOfDaySeed( );       // seed the random number generator

    omp_set_num_threads( NUMT );    // set the number of threads to use in the fo
r-loop:`

    // better to define these here so that the rand() calls don't get into the th
read timing:
    float *xcs = new float [NUMTRIALS];
    float *ycs = new float [NUMTRIALS];
    float * rs = new float [NUMTRIALS];

    // fill the random-value arrays:
    for( int n = 0; n < NUMTRIALS; n++ )
    {
        xcs[n] = Ranf( XCMIN, XCMAX );
        ycs[n] = Ranf( YCMIN, YCMAX );
        rs[n] = Ranf(  RMIN,  RMAX );
    }

    // get ready to record the maximum performance and the probability:
    float maxPerformance = 0.;      // must be declared outside the NUMTRIES loop
    float currentProb;              // must be declared outside the NUMTRIES loop

    //looking for the maximum performance:
    for (int t = 0; t < NUMTRIES; t++)
    {
        double time0 = omp_get_wtime();

        int numHits = 0;
        #pragma omp parallel for default(none) shared(xcs,ycs,rs,tn) reduction(+:
numHits)
        for (int n = 0; n < NUMTRIALS; n++)
        {
            // randomize the location and radius of the circle:
            float xc = xcs[n];
            float yc = ycs[n];
            float  r = rs[n];
```

```
            // solve for the intersection using the quadratic formula:
            float a = 1. + tn * tn;
            float b = -2.*(xc + yc * tn);
            float c = xc * xc + yc * yc - r * r;
            float d = b * b - 4.*a*c;

            // case A: If d is less than 0., then the circle was completely misse
d. (Case A) Continue on to the next trial in the for-loop.
            if (d < 0) {
                continue; //A
            }

            // hits the circle:
            // get the first intersection:
            d = sqrt(d);
            float t1 = (-b + d) / (2.*a);   // time to intersect the circle
            float t2 = (-b - d) / (2.*a);   // time to intersect the circle
            float tmin = t1 < t2 ? t1 : t2;     // only care about the first inte
rsection

            // case B: the circle completely engulfs the laser pointer
            if (tmin < 0) {
                continue;  //B
            }

            // where does it intersect the circle?
            float xcir = tmin;
            float ycir = tmin * tn;

            // get the unitized normal vector at the point of intersection:
            float nx = xcir - xc;
            float ny = ycir - yc;
            float nxy = sqrt(nx*nx + ny * ny);
            nx /= nxy;  // unit vector
            ny /= nxy;  // unit vector

            // get the unitized incoming vector:
            float inx = xcir - 0.;
            float iny = ycir - 0.;
            float in = sqrt(inx*inx + iny * iny);
            inx /= in;  // unit vector
            iny /= in;  // unit vector

            // get the outgoing (bounced) vector:
            float dot = inx * nx + iny * ny;
```

```
            float outx = inx - 2.*nx*dot;   // angle of reflection = angle of inc
idence`
            float outy = iny - 2.*ny*dot;   // angle of reflection = angle of inc
idence`

            // find out if it hits the infinite plate:
            float tt = (0. - ycir) / outy;

            // case C: reflected beam went up instead of down
            if (tt < 0) {
                continue;
            }

            // case D:
            numHits++;
        }

        double time1 = omp_get_wtime();
        double megaTrialsPerSecond = (double)NUMTRIALS / (time1 - time0) / 100000
0.;
        if (megaTrialsPerSecond > maxPerformance)
            maxPerformance = megaTrialsPerSecond;
        currentProb = (float)numHits / (float)NUMTRIALS;
    }


    // printf("Number of threads: %d\n", NUMT);
    // printf("Number of trials: %d\n", NUMTRIALS);
    // printf("Probability: %1.7lf\n", currentProb);
    // printf("MegaTrialsPerSecond: %8.2lf\n", maxPerformance);


    //printf("Threads: %d\t Trials: %d\t Probability:  %8.2lf\t MegaTrialsPerSeco
nd: %8.2lf\t\n", NUMT, NUMTRIALS, currentProb, maxPerformance);
    printf("%d,%d,%1.7lf,%8.2lf", NUMT, NUMTRIALS, currentProb, maxPerformance);



}
```

```
//Helper Functions
float
Ranf( float low, float high )
{
        float r = (float) rand();               // 0 - RAND_MAX
        float t = r  /  (float) RAND_MAX;       // 0. - 1.

        return   low  +  t * ( high - low );
}

int
Ranf( int ilow, int ihigh )
{
        float low = (float)ilow;
        float high = ceil( (float)ihigh );

        return (int) Ranf(low,high);
}

void
TimeOfDaySeed( )
{
    struct tm y2k = { 0 };
    y2k.tm_hour = 0;   y2k.tm_min = 0; y2k.tm_sec = 0;
    y2k.tm_year = 100; y2k.tm_mon = 0; y2k.tm_mday = 1;

    time_t  timer;
    time( &timer );
    double seconds = difftime( timer, mktime(&y2k) );
    unsigned int seed = (unsigned int)( 1000.*seconds );    // milliseconds
    srand( seed );
}
```

```
#!/bin/bash

touch pro1data.csv
rm pro1data.csv
```

```
echo "Number of Threads, Number of Trials, Probability of Hitting the Plate, Mega
trials per Second" >> pro1data.csv

# number of threads:
for t in 1 2 4 8 16 32 64 128
do
    echo NUMT = $t
    # number of trials:
    for s in 10000 100000 1000000 10000000 50000000 100000000

    do
        echo NUMTRIALS = $s
        g++ -DNUMTRIALS=$s -DNUMT=$t pro1.cpp -o pro1 -lm -fopenmp
        ./pro1 >> pro1data.csv
    done
    echo -e >> pro1data.csv
done


echo "The results have been saved in pro1data.csv"
```