
# CS 475/575 -- Spring Quarter 2023

# Project #1

### OpenMP: Monte Carlo Simulation

### 100 Points

### Due: April 18

---

*This page was last updated: March 29, 2023*

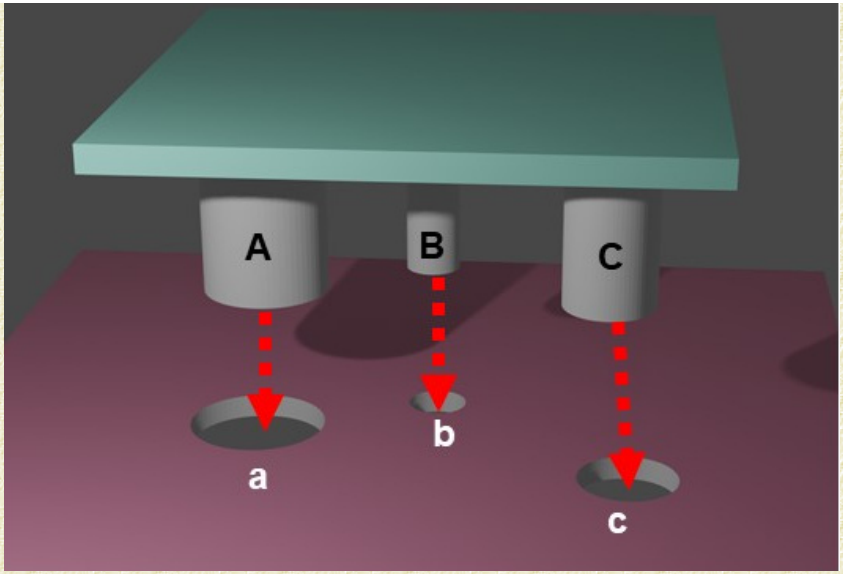
---

## Introduction

Monte Carlo simulation is used to determine the range of outcomes for a series of parameters, each of which has a probability distribution showing how likely each option is to happen. In this project, you will take a scenario and develop a Monte Carlo simulation of it, determining how likely a particular output is to happen.

Clearly, this is very parallelizable -- it is the same computation being run on many permutations of the input parameters. You will run this with OpenMP, testing it on different numbers of threads (at least 1, 2, 4, 6, and 8).

## The Scenario




A top plate has 3 pins in it which must fit into a bottom plate with 3 holes in it. The pin locations are defined as :

| Pin | X | Y | Radius |
|---|---|---|---|
| A | 3.0 | 4.0 | 2.0 |
| B | 4.0 | 5.0 | 2.0 |
| C | 5.0 | 4.0 | 2.0 |

The holes were drilled simultaneously and so the drills rattled and vibrated. Thus, the location and sizes of the holes are approximate and are estimated to be:

| Hole | X | Y | Radius |
|---|---|---|---|
| a | 2.9 ± 0.2 | 4.1 ± 0.2 | 2.2 ± 0.2 |
| b | 4.1 ± 0.1 | 4.9 ± 0.1 | 2.2 ± 0.2 |
| c | 5.0 ± 0.1 | 4.0 ± 0.05 | 2.2 ± 0.2 |

Given all this uncertainty, what is the probability that the three pins will actually fit into the three holes?

## Requirements:

Run this for some combinations of trials and threads. Do timing for each combination. Like we talked about in the **Project Notes**, run each experiment some number of tries, NUMTIMES, and record just the peak performance.

Do a table and two graphs. The two graphs need to be:

1. Performance versus the number of Monte Carlo trials, with the colored lines being the number of OpenMP threads.
2. Performance versus the number OpenMP threads, with the colored lines being the number of Monte Carlo trials..

(See the **Project Notes, Scripting, Graphing,** and **Pivot Tables** to see an example of this and how to get Excel to do most of the work for you.)

Chosing one of the runs (one of the ones with the maximum number of trials would be good), tell me what you think the actual probability is.

Compute Fp, the Parallel Fraction, for this computation.

## Does a Pin Fit?

**d** is the distance from the (x,y) pin center to the (x,y) hole center:
float d = Length( pinx-holex, piny-holey );
A pin fits into its hole if: **d+pinRadius ≤ holeRadius**

All three pins must fit into their respective holes for this trial to be deemed a success. If even one pin does not fit, this trial is a failure.

## The Program Flow

The code below is printed in the handout to make it easy to look at and discuss.
**Note: if you are on Windows, take the ", stderr" out of the #pragma line!**

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

// print debugging messages?
#ifndef DEBUG
#define DEBUG   false
#endif

// setting the number of threads:
#ifndef NUMT
#define NUMT               2
#endif

// setting the number of trials in the monte carlo simulation:
#ifndef NUMTRIALS
#define NUMTRIALS       50000
#endif

// how many tries to discover the maximum performance:
#define NUMTIMES        20

//#define CSV

// the pins; numbers are constants:
const float PinAx =     3.0f;
const float PinAy =     4.0f;
const float PinAr =     2.0f;

const float PinBx =     4.0f;
const float PinBy =     5.0f;
const float PinBr =     2.0f;

const float PinCx =     5.0f;
const float PinCy =     4.0f;
const float PinCr =     2.0f;


// ranges for the random numbers:
```

```
const float HoleAx   =  2.90f;
const float HoleAy   =  4.10f;
const float HoleAr   =  2.20f;
const float HoleAxPM =  0.20f;
const float HoleAyPM =  0.20f;
const float HoleArPM =  0.20f;

const float HoleBx   =  4.10f;
const float HoleBy   =  4.90f;
const float HoleBr   =  2.20f;
const float HoleBxPM =  0.10f;
const float HoleByPM =  0.10f;
const float HoleBrPM =  0.20f;

const float HoleCx   =  5.00f;
const float HoleCy   =  4.00f;
const float HoleCr   =  2.20f;
const float HoleCxPM =  0.10f;
const float HoleCyPM =  0.05f;
const float HoleCrPM =  0.20f;


// return a random number within a certain range:
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 );
}

// call this at the top of your main( ) if you want to force your program to use
// a different random number sequence every time you run it:
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 );
}


// square a number:
float
Sqr( float x )
{
        return x*x;
}


// square root of the sum of the squares:
float
Length( float dx, float dy )
```

```
{
	return  sqrt( Sqr(dx) + Sqr(dy) );
}



int
main( int argc, char *argv[ ] )
{
#ifdef _OPENMP
	//fprintf( stderr, "OpenMP is supported -- version = %d\n", _OPENMP );
#else
	fprintf( stderr, "No OpenMP support!\n" );
	return 1;
#endif

	TimeOfDaySeed( );		// seed the random number generator

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

	// better to define these here so that the rand() calls don't get into the thread timing:
	float *holeaxs  = new float [NUMTRIALS];
	float *holeays  = new float [NUMTRIALS];
	float *holears  = new float [NUMTRIALS];

	float *holebxs  = new float [NUMTRIALS];
	float *holebys  = new float [NUMTRIALS];
	float *holebrs  = new float [NUMTRIALS];

	float *holecxs  = new float [NUMTRIALS];
	float *holecys  = new float [NUMTRIALS];
	float *holecrs  = new float [NUMTRIALS];

	// fill the random-value arrays:
	for( int n = 0; n < NUMTRIALS; n++ )
	{
		holeaxs[n]  = Ranf(  HoleAx-HoleAxPM,  HoleAx+HoleAxPM );
		holeays[n]  = Ranf(  HoleAy-HoleAyPM,  HoleAy+HoleAyPM );
		holears[n]  = Ranf(  HoleAr-HoleArPM,  HoleAr+HoleArPM );

		holebxs[n]  = Ranf(  HoleBx-HoleBxPM,  HoleBx+HoleBxPM );
		holebys[n]  = Ranf(  HoleBy-HoleByPM,  HoleBy+HoleByPM );
		holebrs[n]  = Ranf(  HoleBr-HoleBrPM,  HoleBr+HoleBrPM );


		holecxs[n]  = Ranf(  HoleCx-HoleCxPM,  HoleCx+HoleCxPM );
		holecys[n]  = Ranf(  HoleCy-HoleCyPM,  HoleCy+HoleCyPM );
		holecrs[n]  = Ranf(  HoleCr-HoleCrPM,  HoleCr+HoleCrPM );
	}

	// get ready to record the maximum performance and the probability:
	double  maxPerformance = 0.;	// must be declared outside the NUMTIMES loop
	int     numSuccesses;		// must be declared outside the NUMTIMES loop


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

		numSuccesses = 0;
```

```
		// note: the Pin numbers don't need to be declared shared( ) because they are const variables!
		// note note: if you are windows, take the ", stderr" out of this list!
		#pragma omp parallel for default(none) shared(holeaxs,holeays,holears, holebxs,holebys,holebrs, holecxs,holecys,holecrs, stderr) reduction(+:numSuccesses)
		for( int n = 0; n < NUMTRIALS; n++ )
		{
			// randomize everything:
			float holeax = holeaxs[n];
			float holeay = holeays[n];
			float holear = holears[n];

			float holebx = holebxs[n];
			float holeby = holebys[n];
			float holebr = holebrs[n];

			float holecx = holecxs[n];
			float holecy = holecys[n];
			float holecr = holecrs[n];

			float da = Length( ????? );
			if( ????? )
			{
				float db = Length( ????? );
				if( ????? )
				{
					float dc = Length( ????? );
					if( ????? )
						numSuccesses++;
				}
			}

		} // for( # of  monte carlo trials )

		double time1 = omp_get_wtime( );
		double megaTrialsPerSecond = (double)NUMTRIALS / ( time1 - time0 ) / 1000000.;
		if( megaTrialsPerSecond > maxPerformance )
			maxPerformance = megaTrialsPerSecond;

	} // for ( # of timing tries )

	float probability = (float)numSuccesses/(float)( NUMTRIALS );		// just get for last NUMTIMES run

#ifdef CSV
	fprintf(stderr, ????? );
#else
	fprintf(stderr, "%2d threads : %8d trials ; probability = %6.2f ; megatrials/sec = %6.2lf\n",
		NUMT, NUMTRIALS, 100.*probability, maxPerformance);
#endif

}
```

Print out: (1) the number of threads, (2) the number of trials, (3) the probability of all three pins fitting into the holes and (4) the MegaTrialsPerSecond. Printing this as a single line with commas between the numbers but no text is nice so that you can import these lines right into Excel as a CSV file.

## Function for Getting Random Numbers Within a Range:

To choose a random number between two floats, use:

```
#include <stdlib.h>

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 );
}

// call this if you want to force your program to use
// a different random number sequence every time you run it:
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 );
}
```

## Setting Up To Compile This From a Makefile on Flip

Put the following lines into a file called **Makefile**:

```
proj01:         proj01.cpp
                g++     proj01.cpp   -o proj01  -lm  -fopenmp
```

Run it as:

```
make  proj01
./proj01
```

## Setting Up To Compile and Run This From a Script on Flip

You can save yourself a *ton* of time by setting this up to run from a script. Check the **Project Notes** to see how to do that in bash, C-shell, or Python. If you've never done this before, learn it now! You will be surprised how much time this saves you throughout this class. Here it is as a bash script:

```
#!/bin/bash
for t in 1 2 4 8 12 16 20 24 32
do
  for n in 1 10 100 1000 10000 100000 500000 1000000
  do
     g++   proj01.cpp  -DNUMT=$t -DNUMTRIALS=$n  -o proj01  -lm  -fopenmp
    ./proj01
  done
done
```

Run it as:

```
bash  loop.bash  >&  proj01.csv
```

Diverting it to a CSV file sets you up to import it right into Excel.

## The Turn-In Process:

Your turnin will be done at http://teach.engr.oregonstate.edu and will consist of:

1. All source files (.cpp).
2. A PDF report with
    - A title, your name, and your email address.
    - A rectangular data table of the performance numbers as a function of threads and NUMTRIALS.
    - The **2** performance graphs. The two graphs need to be:
        1. Performance versus the number of Monte Carlo trials, with the colored lines being the number of OpenMP threads.
        2. Performance versus the number OpenMP threads, with the colored lines being the number of Monte Carlo trials.

      The graphs need to have labels on the axes and on the legend. See the **Project Notes, Scripting, Graphing,** and **Pivot Tables** to see an example of this and how to get Excel to do most of the work for you.
    - Your estimate of the Probability.
    - Your estimate of the Parallel Fraction (*show your work!*).
    - Your commentary: why do the graphs look the way they do? What are they telling you?
3. ***Do not put your PDF into a zip file.*** Leave it out separately so my collect-all-the-PDFs script can find it.

## Grading:

| Feature | Points |
|---|---|
| Proper rectangular data table | 10 |
| Good graph of performance vs. number of trials | 20 |
| Good graph of performance vs. number of threads | 20 |
| Good estimate of the probability | 15 |
| Good estimate of Fp, the Parallel Fraction (*show your work*) | 15 |
| Commentary | 20 |
| **Potential Total** | **100** |