Monte Carlo Methods and π

Monte Carlo methods use random number generators in simulations. For example, one might study traffic flow by using a random number generator to specify when, where, and how fast vehicles are moving. One might study the behavior of the stock market by using a random number generator to simulate when and how much of a stock an investor is buying or selling. Monte Carlo methods are especially valuable when more conventional approaches (e.g., solving differential equations) are very difficult or impossible to use. They can often be used (as in the traffic flow example) to directly simulate a physical situation.

One of the simplest applications of Monte Carlo methods is to estimate π, the ratio of the area of a circle to the square of its radius. To see how this works imagine that you have a circular dart board with a radius of 1 foot, and the dartboard is attached to a square with sides 2 feet in length so that the edge of the dartboard is inscribed in the square. Now close your eyes and start throwing darts at the square.

After (say) several thousand tosses, you open your eyes and count the number of darts that hit the dartboard and the number that hit the square. Think about the ratio

$$\frac{\text{Number that hit the square} + \text{Number that hit the dartboard}}{\text{Number that hit the dartboard}}.$$ 

Since the darts were tossed randomly at the board, we would guess that this ratio should be about the same as

$$\frac{\text{Area of the square}}{\text{Area of the circle}}.$$ 

So suppose

$$n = \text{Number that hit the square} + \text{Number that hit the dartboard}$$

$$d = \text{Number that hit the dartboard}.$$ 

Then since the area of the square is 4 square feet and the area of the circle is π square feet, we should have

$$\frac{n}{d} \approx \frac{4}{\pi}.$$ 

Equivalently, we should have

$$\pi \approx \frac{4d}{n}.$$ 

1
Furthermore, as $n$, the total number of darts hitting the square and the dartboard, increases, the law of averages says that the approximation should get better and better.

We can use this idea to estimate $\pi$ by replacing our dartboard and dart-throwing with a random number generator. Here’s the basic idea:

```c
in_circle_count = 0;
seed the random number generator;
for (i = 0; i < n; i++) {
    Generate random x-coordinate in [-1,1];
    Generate random y-coordinate in [-1,1];
    if ((x,y) is in the unit circle) in_circle_count++;
}
pi_estimate = 4.0*in_circle_count/n;
```

Random number generators are ordinarily “seeded” to control the start of the sequence of random numbers. So if a program is supposed to generate a different sequence of random numbers every time it’s run, the program could be seeded with something like the number of seconds since midnight. This isn’t important for a serial $\pi$ program, but, as we’ll see, it is important for a parallel $\pi$ program.

We can determine whether $(x, y)$ is in the unit circle by finding out how far it is from $(0,0)$:

$$\text{if}(\sqrt{x^2 + y^2} \leq 1).$$

We can save a little bit of computational work by observing that $\sqrt{x^2 + y^2} \leq 1$, if and only if $x^2 + y^2 \leq 1$. So we can simplify our test to

$$\text{if}( x^2 + y^2 \leq 1).$$

**Serial Program**

It gets the true value of $\pi$ (up to the precision of the computer) by using the arctangent or inverse tangent function: since

$$\tan\left(\frac{\pi}{4}\right) = 1,$$

we have

$$\pi = 4 \arctan(1).$$

The math library inverse tangent function is `atan`. So we define

```c
#define PI 4.0*atan(1.0)
```
Note that since the program uses the `atan()` function it will need to be linked with the library. So it should be compiled with something like

```
$ gcc -g -Wall -o monte_carlo_pi monte_carlo_pi.c -lm
```

The C header file `stdlib.h` contains several random number generators. Probably the best one is `drand48()`. To seed this you should use the `srand48()` function, which takes a long int argument. In the serial program, we just use argument 0. In the parallel program, you’ll need to use more care (see below).

The `drand48()` function generates a double between 0 and 1. In order to get it to generate a double between -1 and 1, you can simply multiply by 2 and subtract 1. So the program gets the $x$- and $y$-coordinates with the code

```
x = 2.0*drand48()-1.0;
y = 2.0*drand48()-1.0;
```

**The Parallel Algorithm**

In order to parallelize this algorithm we can simply divide the number of throws at the dartboard among the processes. So each MPI process will “throw $n/p$ darts” and count the number that hit the dartboard. Then we can form a global sum on process 0, which can then be used for estimating $\pi$.

An important detail is the seeding of the random number generator on each process with `srand48()`. If you simply give it the argument 0, every process will generate exactly the same sequence of random numbers, and there won’t be any point in using multiple processes. One way to avoid this is to give it the argument `my_rank` on each process.

**The MPI Program**

Input will be the same as the input for the serial program: $n$, the total number of points in the square (or the number of “darts”). This should be read in by process 0 and distributed among the other processes. Each process should calculate $n/p$, the number of points or “darts” it’s responsible for. Each process should then simulate throwing $n/p$ darts. When the processes are done with their simulations, the total number of darts that hit the circle across all the processes should be stored on process 0, and process 0 should print

- The number of dart throws ($n$),
- The number that hit the dartboard or circle,
- The resulting estimate of $\pi$,
- The actual value of $\pi$, and
- The error in the estimate: $\pi - \text{estimate}$.