

CS 140 Program 1: Monte Carlo integration and random number generators

Assigned January 22, 2007

Due by midnight Wednesday, January 31

The goals of this assignment are to write a simple Monte Carlo integration routine, and to explore some of the pitfalls of parallel random number generation. Read Section 3.2.2 of the textbook for background.

1 Approximating π by Monte Carlo integration

Your assignment is to write and run a parallel program that computes an approximation to pi using two-dimensional Monte Carlo integration.

In this case, 2D Monte Carlo is a fancy way of saying “approximate the area of a round board by throwing darts at a square board and counting what fraction of them land inside the round board.” A dart is a pair of (x, y) of random numbers between 0 and 1. The square board is the unit square, $0 \leq x \leq 1$ and $0 \leq y \leq 1$. The round board is actually only the upper right hand quadrant of the unit circle, consisting of those (x, y) with $x^2 + y^2 \leq 1$. Formally speaking, you will use Monte Carlo to approximate the integral

$$4 \int_0^1 \int_0^1 f(x, y) dx dy,$$

where

$$f(x, y) = \begin{cases} 1 & \text{if } x^2 + y^2 \leq 1, \\ 0 & \text{otherwise.} \end{cases}$$

The value of this integral is, of course, $\pi \approx 3.1415926535897932$.

Monte Carlo is a common way to compute integrals in many dimensions. There are actually lots faster ways to compute pi, but this makes a good test of a Monte Carlo routine because we know exactly what the answer is supposed to be.

Write your program as a master-slave embarrassingly parallel program, in which the master sets the parameters and then collects the final answer when the slaves are all finished.

You would expect your program to compute better and better approximations as the number n of darts grows. Try it out for various values of n . I suggest $10^2, 10^4, 10^6, 10^8, 10^{10}, \dots$; but don't do any runs that take more than about a minute. For each test, use the exact value of pi to compute the error in your approximation.

2 Random number generators

The trickiest part of this computation is generating the random numbers. In particular, you want each processor to have “its own supply” of random numbers, independent of the others.

One way to do this is to generate all the random numbers on the master processor and distribute them to the slaves with sends and receives. This doesn't work well at all, though, for two reasons: First, the master does about as much work as all the other processors combined just to generate the random numbers; second, the volume of communication is $O(n)$, the same as the amount of computation, so the program would be hopelessly communication-bound.

A better approach is to let each processor generate its own random numbers. But then how do we make sure they're actually independent of each other?

For this problem, you should experiment with several different random number generators, as follows. (Make sure you scale your random numbers to be between 0 and 1.)

1. The C library routine `rand`, using the default seed. (Question: what's wrong with this approach?)
2. The C library routine `rand`, using `srand` to set the seed to a different value in each processor. (Question: what's wrong with this approach?)
3. The "good" multiplicative congruential generator described on page 97 of the book, with $a = 16807$, $m = 2^{31} - 1$, and $c = 0$. Use the "jump constant" method to get a different sequence in each processor. (Question: what's wrong with this approach?)
4. For extra credit, locate the `SPRNG` parallel random number generator library (it's installed somewhere on DataStar, and the library is also available online) and use one of its methods.

3 What to run and turn in

For each random number generator you use, run several values of n and p , roughly up to what you can run in a minute of time for each p . I suggest trying $p = 1, 2, 4, 8, 16, \dots$

For each generator, make tables of the error in your estimate as a function of n and p , and also of the running time as a function of n and p . For each generator, do you get the same results for fixed n with different values of p ? For a large value of n , is your speedup close to p ? What peculiar behaviour and/or disadvantages do you observe for the different generators? What other conclusions can you draw?

You should turn in your source code, plus some representative console output (you don't have to turn in the output of all your runs), plus a README file that contains the tables above and a report at least a couple of paragraphs long interpreting the results and giving your conclusions.

You can run this problem on either DataStar or the cluster. I recommend trying DataStar just because the tools are better — for example, you can use VAMPIR et al. to understand your speedup results — but you may use the cluster if you prefer.

You may do this assignment alone or in groups of two.