Estimating π Using a Monte Carlo Method

November 14, 2008

In this exercise, you will become familiar with Chapel's basic syntax, including some of the parallel tasking features, while estimating the value of π using a Monte Carlo method.

1. As with any Monte Carlo method, you will need a pseudorandom number generator. A simple one, taken from [**?**], is a multiplicative linear congruential generator defined by the following formula:

$$x_{n+1} = ax_n \mathrm{mod} m, \tag{1}$$

where the multiplier $a = 16807$ and the modulus $m = 2^{31} - 1 = 2147483647$. To get started with this exercise, you will implement this random number generator as a function that takes an argument (corresponding to x_n in the formula above) and use it to compute and return the next random number in the sequence. (If you are terrified of starting with a blank screen, you can use RandomNumber0.chpl in the solutions to the exercises as a starting point.) Be careful with how you do the multiplication: the product ax_n can be as large as $16807 \times 2147483646 \approx 1.03 \times 2^{45}$.

2. To test your random number generator, you can write a main program that invokes your random number generator 10000 times, with 1 as the argument when you invoke it the very first time. Have your program write out the last return value; if it writes out 1043618065, you have implemented your random number generator correctly.

3. You can use your random number generator to simulate throwing darts at random at a unit circle. Each dart will fall somewhere on a 1×1 grid. Therefore, you will need to modify your random number generator to generate real numbers between 0 and 1 inclusive. One way to do this would be to create another function that takes no arguments, calls the one you just wrote, subtracts 1 from the return value (so the value is now in the range of 0 and 2147483645 inclusive), and then divides that by 2147483645 (using floating-point arithmetic). Your new function can pass in a global variable to the one you wrote before.

4. Since you follow good software engineering practices, you will want to encapsulate your code in a module. First, replace your random number generator's argument (often referred to as the seed) with a (global) config var that is initialized to 1. This

will have the advantage of allowing the initial value to be specified on the command line when you run your program by using the "–<*config-var-name*>=<*value*>" flag (without the double quotes), where *config-var-name* is the name of your config var, and *value* is the initial value you want to use (instead of 1). Next, to make things more convenient later on, put your main program's code into a function that does not take any arguments. Create a separate source file that uses the module you just wrote and invokes what used to be the main program and is now a function. You will need to use the "`--main-module` <*module-name*>" flag when you compile your source files together to specify which module contains the main program. (Unless you specify otherwise in your source code, the name of a module is taken from its file name.)

5. In order to determine if a particular simulated dart hit inside the unit circle, you can generate a random point (x, y) in a 1×1 grid, and then calculate its distance from the origin using the formula $d = \sqrt{x^2 + y^2}$. If this distance $d \leq 1$, then the dart fell inside the unit circle. You can then calculate the proportion of how many simulated darts fell inside the unit circle as compared to the total number of simulated darts. This proportion corresponds to the area within a 1×1 grid that is within the unit circle. Now, the area of a unit circle is given by the formula $A = \pi r^2$, where r is the radius of the circle. Since in this case, the radius $r = 1$, the area is simply π. However, a 1×1 grid covers only one of four quadrants in which the unit circle lies, so the proportion of simulated darts that fall inside the unit circle within this grid will have to be multiplied by 4. I.e.,

$$\pi \approx 4 \times \frac{D_{\text{in}}}{D_{\text{total}}},$$

where D_{in} is the number of simulated darts that fall inside the unit circle within a 1×1 grid, and D_{total} is the total number of simulated darts thrown.

In the separate source file you created above, write a program that generates a random point (by invoking your random number generator twice), and determines if it is inside the unit circle. Now, put a loop around this, and keep a count of the total number of points generated, as well as how many of those points fell inside the unit circle. Then, print out the proportion of these counts multiplied by 4. Run your program several times, each time with a different number of generated points. (You may find it convenient to use a config const to control the number of generated points from the command line you use to run your program.) The accuracy of your estimate of π should improve as you increase the number of simulated darts.

Optional: Use the reduce operator with a square-bracketed forall expression as its operand to count how many points are inside the unit circle. Note that you can use a boolean expression as part of the forall expression; the value of a boolean expression is 1 if it evaluates to true, and 0 otherwise.

6. It would be great to improve the performance of your program by parallelizing it. One way to do that would be to break up the counting of the points among several tasks by using the coforall statement. This will require a thread-safe random number generator. You can test if your random number generator is thread safe by replacing the for statement in what used to be the main program of your random number generator module with a coforall statement. (Ignore the warning message about running out of threads to run your program.) Since the computation for each iteration is so short, it is unlikely you will run into any race conditions. However, if you use a standard module called Time, and insert `sleep(1);` in the function that computes the next random number to delay its computation by about a second, you will be much more likely to run into a race condition, and the output of your program will very likely be different.

7. In order to make your random number generator thread safe, you can use a sync variable so that only one iteration of the coforall statement can read and then modify the global variable your random number generator uses: once a sync variable is read, no task (including the one that just read it) can read it again until it has been written to. Now that your random number generator is thread safe, replace the for statement in your program to estimate π with a coforall statement. You may need to use a sync variable to ensure you get an accurate count!

8. Unfortunately, this is not a very satisfactory way to parallelize this program because the random number generator serializes execution. There are several ways around this. One way is to have each iteration use a different random number generator to create several points. For the purposes of this exercise, you can create a new random number generator by using different values for the multiplier a in equation **??** above, such as 397204094 or 950706376. (See [**?**].) For each iteration, the count of points that lie inside the unit circle can be accumulated in a separate variable, and these counts can be added together at the end.

9. As an exercise for the reader, you can distribute the computation to multiple locales, and have each locale execute one iteration of the coforall statement. In order to avoid having each locale access global arrays or variables, which would reside in locale 0, you can encapsulate all the data that a locale will need into a class, and then have each locale instantiate that class.