Generating random numbers in parallel

Mark Hoemmen <mhoemmen AT cs DOT berkeley DOT edu>

26. Aug. 2007

1 What is a random number generator?

Homework 1 involves a Monte Carlo (MC) simulation. For that application, you need a lot of random numbers. To make the MC simulation parallel, you'll need to generate the random numbers in parallel too. We won't make you implement all of this yourself in HW 1, but we do think it's important that you understand what's going on behind the scenes. It's likely you'll have to use random numbers in some of your parallel applications in the future, and you may very well have to parallelize random number generation yourself some time.

We'll begin by explaining what we mean by "random" numbers. Then we'll talk about modeling random numbers as streams, and how this helps with correct and efficient parallelization.

1.1 Random numbers

A truly *random* process is completely unpredictable. It produces a sequence of outcomes that follow a particular probability distribution, but future outcomes cannot be determined by mathematics or science, no matter how much information you collect about the process. When each outcome is a number, we have a sequence of *random numbers*.

1.2 Probability distribution

The *probability distribution* associated with a random process governs the likelihood of any particular outcome. The most familiar distribution is the *uniform distribution*, which means that any number is as likely to appear as any other. This is what we want from coin tosses or die[1] rolls – unless, of course, we want to cheat!

You might have seen other distributions that are very useful in practice. These include the *Gaussian distribution* or "bell curve," which governs things like heights and weights in a population, and the *exponential distribution*, which might have been discussed in your systems class in the context of queueing theory. If you have uniformly distributed random numbers, you can usually transform them into random numbers with another distribution. Hence, we'll assume

[1] "Die" is the singular of "dice."

from now on in this document that “random numbers” refers to uniformly distributed random numbers.

1.3 Pseudorandom, not random

It’s hard to say whether any physical process can be truly random. One can even build a mechanical coin-flipping device that makes the coins always come up heads.[2] However, many events, such as radioactive decay, can be accurately modeled as random.[3] Any computer algorithm is a non-random process by definition: it is a deterministic function of its inputs, with an ordered and finite mathematical description. If you know the algorithm and the inputs, you can predict the output by following the rules of the algorithm.

In order to do accurate Monte Carlo simulations, good cryptography, and the like, we need a simple algorithm that gives us numbers with statistical properties of randomness. Note that I didn’t write “random numbers,” but rather, “numbers with statistical properties of randomness.” These are called *pseudorandom numbers*. This distinction is a good thing for us, because there are actually simple algorithms with these properties. We call these algorithms *pseudorandom number generators*, or PRNG’s for short.

1.4 Pseudorandom number generators

You might have used PRNG libraries for your previous homeworks and projects. An example is rand() in the C standard library. Usually, you call some “function,” and it gives you a “random number.” We put “function” in quotes because it’s definitely *not* a function in the mathematical sense: it gives you a different answer each time you call it! What’s going on behind the scenes?

A clue comes from how PRNG libraries are initialized. Usually you have to call some initialization function, like srand() in the C standard library, and give it some information. This is called a *seed*. Usually, if you give a program the same seed twice, it produces the same sequence of random numbers.

It’s pretty clear from this that a PRNG is a kind of state machine: each time you ask for a random number, the state changes, so that the random number is different next time. In fact, since the PRNG runs on a computer, which is finite, it must be a *finite state machine*. Formally, a PRNG is defined by three things:

- An *initial state* s_0;
- A *transition function* S on states, such that $s_{k+1} = S(s_k)$ for all $k \geq 0$;

[2]See, for example, `http://www.news.cornell.edu/chronicle/97/10.23.97/statistics.html`.

[3]This has been exploited in physical devices, such as the one described in this patent: `http://www.freepatentsonline.com/6745217.html`.

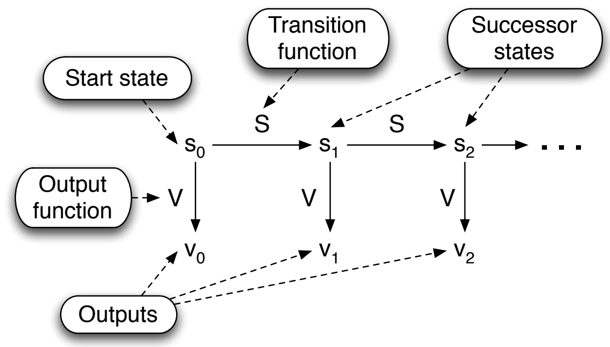

Figure 1: The state model of a pseudorandom number generator.

- An *output function* V on states, such that for all $k \geq 0$, $V(s_k)$ is the pseudorandom number at invocation k of the PRNG.[4]

Figure 1 illustrates these three parts of a PRNG. The sequence of pseudorandom numbers produced by the PRNG is $V(s_0)$, $V(s_1)$, $\ldots$.

The transition function of most PRNG's is a recurrence relation based on modular integer arithmetic. Try this yourself: take some simple function of integers with integer constants, like $f(x) = ax + b \mod M$, and iterate on it. For the right values of a, b, and M, you'll get a sequence that skips around a lot and looks random. Try, for example, $M = 24$, $b = 13$, and $a = 7$. Start with $x = 0$, and you should get the following list of numbers: 0 7 2 9 4 11 6 13 8 15 10 17 12 19 14 21 16 23 18 1 20 3 22 5, after which it repeats with 0. People actually use this method (with different constants); it's called a *linear congruential generator*.[5] It's easy to code and efficient, but its randomness properties aren't so good. Improvements usually involve using a different transition function and/or keeping more state. Sometimes, changing the output function can help too; this is called *tempering*.

Each PRNG defines a single sequence of pseudorandom numbers. Since modular integer arithmetic allows only a finite number of numbers, eventually the sequence must repeat. Once the numbers start repeating, the numbers aren't pseudorandom anymore; they are *correlated*. That is, one can use previously seen numbers to predict future numbers in the sequence. This can hurt the accuracy of Monte Carlo simulations. Correlations are also bad for cryptography algorithms, because they mean that an adversary might see patterns in encrypted text and exploit them to break the code. Statistical tests can be used to determine whether a particular PRNG produces a sequence with sufficient statistical randomness, but sometimes a PRNG that passes these tests

[4]V stands for "value." We didn't use capital-O because that looks too much like zero or "big-Oh" notation. Fun fact: many mathematicians call big-Oh notation "Landau notation," after the German mathematician who popularized it.

[5]Look this up in Wikipedia for some values of a, b, and M used in practice.

may break on a certain sensitive application. It may be a good idea to test a MC simulation with multiple PRNG's, to check if it's sensitive to the particular PRNG that you use.

2 Parallel PRNG's

2.1 Choosing the model

How do we write a parallel pseudorandom number generator? and how do we make it run fast? One part of making parallel code run fast is *choosing the right models*. We'll start with an example: the C standard library's PRNG. It's designed for sequential use only, but its interface does present a particular model of the pseudorandom number generation algorithm. You use it as follows:

- First call the srand() subroutine with an unsigned 32-bit integer "seed," which is used to generate the initial state of the sequence. Typically, to make an unpredictable sequence, one uses the current time.
- Then call rand(), which for each call, returns 32-bit signed integer random numbers in the sequence.

This simple interface hides something, namely, the *shared state* which is read and written by rand(). Each time you call rand(), it reads this "previous state," and uses the PRNG recurrence to produce the "current state." This state is used to calculate the current random number in the sequence, which the subroutine returns.

The C library has many functions like this, that modify shared state. Such functions are called "not thread-safe" or (more generally) *nonreentrant*. If you don't read the man pages carefully, you may not realize this. This actually happened to our research group colleague who was developing the solution code for Homework 1. Clearly, the C library was not designed with concurrency in mind!

2.2 The shared state model

How can we adapt the C library's PRNG to make it thread-safe? The colleague who developed Homework 1 discovered by trial and error that rand() was not thread-safe. He corrected the problem by the following steps:

- Ensure that srand() is called first by only one thread, before any thread can invoke rand(). He did this by starting the program with only one thread, calling srand() in that one thread, and only then creating the other threads.
- Protect every call to rand() with a mutex. This ensures that the shared state used by rand() can only be updated by one thread at a time.

This is an example of what one might call the *shared state* model of thread synchronization. Our colleague saw the PRNG as something that can read or write the state of an (invisible) shared variable at any time. The natural way to protect such a shared variable is to wrap it with some sort of mutual exclusion. This results in a correct solution, but one that does not perform well in practice, in this application. Every worker thread calls rand() quite often, near or perhaps even below the scale of overhead for a contended lock. As a result, the mutex protecting rand() tends to "linearize" the threads. Performance looks more like a sequential code – or may be even worse. That's pretty embarrassing to explain to the boss, when she comes around and asks how your your week spent parallelizing her application went!

2.3 Shared state model too general

In the shared state model, the problem seems insurmountable. Many people recommend using a hardware solution, like the "compare and swap" (CAS) instruction supported by many modern CPU's. However, CAS can only protect a very small amount of shared state (32 or 64 bits, usually). Many PRNG's use significantly more shared state that this. For example, the Mersenne Twister, which is designed for Monte Carlo simulations, uses 624 32-bit integers [3]. Furthermore, any sort of synchronization, whether in hardware or in software, involves a performance penalty, which probably won't scale to very large numbers of threads.

The problem is not so much with the implementation, but with the model. The shared state model is too general. It can model all sorts of things – a shared hash table or cache, or even shared memory itself! If you had the chance to simulate a shared cache in your computer hardware classes, you know how many details you have to keep track of in order to make it work correctly – let alone efficiently. Often, a more restrictive model can enable valuable optimizations.

2.4 The stream model

Recall my first description of PRNG's as a recurrence relation: an initial state, and a (pure, i.e., no side effects) function that transforms the previous state into the current state. This pattern appears in many different places, under many different names and forms. You might think of it as a *lazy list*, because you have the current state (the CAR or head of the list) and a function to get to the next state (the CDR function). If you have ever programmed in Haskell or similar languages, you might know it as a special kind of *monad*. We'll call it a *stream* here.

Why should you care about models? Well, recall that the PRNG recurrence is a pure function of its input, which is the current state. That is, the recurrence function doesn't modify the current state. This means we can define the interface to the PRNG differently than the C library does. Rather than destructively setting the initial state with srand() and destructively modifying it with rand(), we can define an initial state and loop nondestructively over all successive states.

Here's how you might write this in Scheme.[6] Suppose that given a state, the NEXT−STATE function returns the next state. Then we can map over some number NUM−ITERATIONS of the states:

```
(define map-states
  (lambda (current-state the-function num-iterations)
    (if (<= num-iterations 0)
        nil
        (cons (the-function current-state)
              (map-states (next-state current-state)
                          the-function
                          (- num-iterations 1))))))
```

The great thing about this kind of interface is that it makes the shared state disappear! There's only a "stream" of pseudorandom numbers.[7] As long as you can get the different threads to use different, statistically uncorrelated streams, the threads no longer need to communicate.

2.5 Multiple streams

How do we get multiple streams of random numbers? There are two approaches. First, we could split a single stream into multiple, uncorrelated streams. For this, we have to make sure that no two threads see the same point in the sequence, until the threads exhaust all the numbers in the sequence. Second, we could generate multiple streams by using different PRNG's, one for each thread. Each PRNG's sequence must be uncorrelated with any of the other sequences.

2.5.1 Sequential reproducibility

Often, you may want that the parallel version produces exactly the same output as the sequential version. This is very, very handy for debugging and testing. If you only have one stream of pseudorandom numbers, no matter how many threads you use, then you can write your code to guarantee that you'll always get the same answer for the same initial state. In contrast, if you use a different stream for each thread, you may get different answers, depending on the number of threads. This may actually be a good thing sometimes, and it's not often a bad thing, but you should be aware of it.

2.5.2 Running out of numbers?

Splitting a single stream is generally easier, because all you have to do is ensure that different threads never see the same part of the stream. This ensures that

[6] I tend to code in Common Lisp rather than Scheme, so you might find a syntax bug here.

[7] Many extensions to the C library offer a similar interface, which is usually called rand_r(). Presumably the "r" stands for "reentrant."

each thread's pseudorandom numbers are not correlated. However, splitting the stream means that there are fewer pseudorandom numbers available per thread. Let's say the sequence is N numbers long, and there are p threads. Let's also assume that p divides N evenly. Then each thread only has N/p pseudorandom numbers. For some large-scale (p is large), long-running scientific MC simulations, this may not be enough. You should always estimate the number of pseudorandom numbers that you need for your application, to be sure that sure that N/p is enough for you. If you're writing a video game, you might not care, but if you're doing a "serious" MC simulation, this is important.

2.5.3 Splitting a single stream

There are two ways to split a stream. Let's say that the threads are numbered from 0 to $p-1$, and the elements in the sequence are numbered from 0 to $N-1$. Then we can either

1. start thread t at spot $t * N/p$ in the sequence, and let each thread step through its length N/p subsequence; or
2. start thread t at spot t in the subsequence, and let each thread skip p elements at a time in the subsequence. This is sometimes called *leapfrogging*.

Method 1 requires that the PRNG algorithm support efficiently starting at an arbitrary point at the sequence, given only the starting condition. There are PRNG's with this property. One way to implement Method 2 is to invoke the recurrence p times each time a thread needs the next element in its subsequence. However, this can be costly, especially for large p. An alternative is to use a PRNG that supports efficient "leapfrogging" (skipping ahead p steps in the sequence). PRNGlib is a library that supports both approaches [2].

2.5.4 Generating multiple streams

Another approach is to generate multiple pseudorandom streams. This often involves using a PRNG with parameters; each thread picks different parameters, so that it gets a unique sequence. It's necessary to prove that the different choices of parameters result in uncorrelated sequences.

A handy feature of the multiple-streams technique is that it's no longer necessary to transform from the shared state interface to the stream interface, because there is no shared state! This is the approach taken by the parallel version of the Mersenne Twister PRNG, as well as SPRNG Version 4.0 [3, 1].

3 Summary

- Pseudorandom number generators (PRNG's) are algorithms that produce a sequence of numbers that appears statistically random.
- The C library provides a PRNG whose interface uses the shared state model. This model is too general and can hurt parallel performance.

- The stream model is more natural for PRNG's, and it can help factor out communication between threads.
- Parallel PRNG's work by either splitting a single stream of numbers into substreams, or by generating multiple independent streams.
- Splitting a single stream makes the results independent of the number of threads, but limits how many pseudorandom numbers are available per thread.

4 Suggested exercises

These are not required, but they might be interesting to try. Some are practical (code something up) and some are for thinking.

- Find a system which supports the rand_r() extension to the C standard library, and figure out how to use it for multithreaded parallel pseudorandom number generation. How might you pick the different threads' seeds so that they are not correlated? Be as paranoid about avoiding correlations as possible.
- When might converting a PRNG from a shared state model to a stream model hurt single-processor performance? (Hint: what if there's a lot of shared state? How would we ordinarily make sure that state s_k and s_{k+1} are distinct for all values of k?)

References

[1] M. Mascagni and A. Srinivasan, *Algorithm 806: Sprng: A scalable library for pseudorandom number generation*, ACM Transactions on Mathematical Software, 26 (2000), pp. 436–461.

[2] N. Masuda and F. Zimmermann, *PRNGlib: A Parallel Random Number Generators library*, Tech. Rep. CSCS-TR-96-08, Swiss National Supercomputing Centre, CH-6928 Manno, Switzerland, May 1996.

[3] M. Matsumoto and T. Nishimura, *Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator*, ACM Transactions on Modeling and Computer Simulation, 8 (1998), pp. 3–30.