*Distributed random number generation**

F. WARREN BURTON[1] AND REX L. PAGE[2]

[1] *School of Computing Science, Simon Fraser University, Burnaby, British Columbia, Canada V5A 1S6*
burton@cs.sfu.ca

[2] *Amoco Production Company, Research Center, 4502 East 41st Street, Post Office Box 3385, Tulsa, Oklahoma 74102, USA*
rpage@trc.amoco.com

Abstract

In a functional program, a simple random number generator may generate a lazy list of random numbers. This is fine when the random numbers are consumed sequentially at a single point in the program. However, things are more complicated in a program where random numbers are used at many locations, such as in a large simulation. The programmer should not need to worry about providing separate generators with a unique seed at each point where random numbers are used. At the same time, the programmer should not need to coordinate the use of a single stream of random numbers in many parts of the program, which can be particularly difficult with lazy evaluation or parallel processing.

We discuss several techniques for distributing random numbers to various parts of a program, and some methods of allowing different program components to evaluate random numbers locally. We then propose a new approach in which a random number sequence can be split at a random point to produce a pair of random number sequences that can be used independently at different points in the computation.

The approach can also be used in distributed procedural programs, where it is desirable to avoid dealing with a single source of random numbers. The approach has the added advantage of producing repeatable results, as might be needed in debugging, for example.

Capsule review

Functional programmers often face 'plumbing problems' – distributing pieces of a global data structure to the places where they are needed. A typical example is the distribution of random numbers from a central generator throughout the rest of the program. The most obvious methods lead to pragmatic difficulties, including awkward plumbing, space leaks, and undesirable statistical dependence of random sequences. After surveying several solutions, the authors offer a new one which uses both nondeterminism and some special properties of random number generators.

1 Introduction

Problems arise in writing functional programs for applications requiring random numbers because randomness is related to nondeterminism, which can easily conflict with referential transparency, a basic requirement of functional programming

* This work was supported by the Amoco Production Company.

systems (Clinger, 1982; Hughes and O'Donnell, 1990; Søndergaard and Sestoft, 1990). Problems of a different nature arise in supplying random numbers to an application running on a computing system with many processing elements. This calls for the provision of an indefinite number of streams of statistically independent random numbers, which conflicts with the standard practice of generating random numbers at a central source. The problems that arise in these two arenas turn out to be related.

Consider the task of writing a functional program to describe a computation that requires random numbers. Suppose that a procedural random number generator is available, and that repeated invocations of this generator produce a sequence of independent random numbers. If the computation uses random numbers in only place, then a functional program could contain one reference to the entire sequence (represented by a lazy list), and the computation would consume numbers from the sequence as needed. On the other hand, if the computation requires random numbers at several points, the program could not use the one available sequence at all of those points, because that would lead to statistical dependence among supposedly independent parts of the computation.

To make the various required sequences statistically independent, the given sequence must be parcelled out somehow to all those parts of the program that require random numbers. A similar program for a parallel computing system would face the problem of distributing random numbers from a single sequence to multiple processes. Thus, functional programs and parallel computations both encounter the same difficulty with random number generators: that of sharing a single resource among multiple consumers.

To address this problem, in section 2 we start by considering an approach to random number generation that will be used in examples in the remainder of the paper. In sections 3–5 we consider several approaches to the problem of distributing random numbers from such a generator to multiple consumers. In section 6 we propose an approach to the distribution problem based on splitting a random sequence at a random point, and discuss its advantages for parallel computation.

2 Sequential random number generation

With many random number generators, each random number in a sequence may be quickly generated from the previous one. This is one of two properties that we will require for our approach. The other property is that the nth random number of the sequence, for arbitrary n, can be computed reasonably quickly.

Our required properties are satisfied by most random number generators in common use (L'Ecuyer, 1988; Park and Miller, 1988). For example purposes, we will use the 'minimal standard' random number generator proposed by Park and Miller (1988). This random number generator assumes that integers within at least the range $[-2^{31}+1 \,..\, 2^{31}-1]$ are available for arithmetic operations, and it delivers integers in the range $[1 \,..\, 2^{31}-2]$.

Let $x_0, x_1, \ldots,$ be a random number sequence, and let *next* be the function defined by $x_{i+1} = next(x_i)$. We will call the initial value in the sequence, x_0, the seed.

```
module Random (Random, seed, generate) where
data Random = Seed Int

modulus = 2147483647   -- = 2^31-1

seed:: Int -> Random
seed s = Seed (normalize s)
normalize s = (s`mod`(modulus-2))+1
-- Avoid zero as a seed value.

next:: Int -> Int
-- next x = (x * 7^5) `mod` modulus
next x = if (result >= 0) then (result) else (result+modulus)
        where
        result = (x `mod` q) * a - (x `div` q) * r
        a = 16807       -- = 7^5
        q = 127773      -- = modulus div a
        r = 2836        -- = modulus mod a
generate:: Random -> [Float]
generate (Seed x) = map scale (iterate next x)
        where
        scale y = fromIntegral y/fromIntegral modulus
```

Fig. 1. Minimal standard random number generator in Haskell.

The programmer provides a seed, and the *next* function for the minimal standard random number generator is as defined in the following equation:

$$next(x) = (x \times a) \bmod m$$

where

$$a = 16807 = 7^5$$

$$m = 2147483647 = 2^{31} - 1.$$

An implementation of *next* in Haskell is given in Fig. 1. The implementation introduces an abstract type, ***Random***, to simplify subsequent discussions. The implementation avoids overflow problems, assuming the system supports integers in the range $[-2^{31}+1\,..\,2^{31}-1]$. The Haskell standard (Hudak *et al.*, 1990) requires only that a the range $[-2^{39}+1, 2^{29}-1]$ be supported for type ***Int***. In some Haskell implementations, it may be necessary to use type ***Integer*** instead of type ***Int***, which is likely to significantly degrade performance. This cannot be avoided, since the minimal standard random number generator delivers integers in the range $[1\,..\,2^{31}-2]$.

3 Plumbing sequences

A straightforward approach to supplying random numbers to a computation is to feed a single sequence into the first point of need, and to design that portion of the program to deliver, as part of its results, the unused portion of the sequence for use elsewhere. In a functional program, for example, a function needing a sequence of random numbers would take the sequence as one of its inputs, and would include the unused portion of the sequence in its output. The remainder of the sequence could then be used in the same way in other parts of the computation.

One problem with this approach is that functions are burdened with having to produce an extra result component, and the compositions of these functions that are required to describe the overall computation are much more complex than would otherwise be necessary. That is, the extra 'plumbing' needed to channel the random numbers through the computation can complicate programs significantly.

Another problem is that of locating the first point of use of the random sequence. Often, a programmer cannot easily predict where a processor evaluating a functional program will begin its computation. Even if the point of first usage and the sequence of subsequent usages are immediately apparent, the program might be running on a computing system with many processors, in which case the strict sequentiality in this solution of the parcelling problem will not be acceptable. That is, strictly sequential access to the random number sequence can greatly reduce the amount of parallel computation available in the program. A sequential program using lazy evaluation will encounter a similar problem: many of the advantages of lazy evaluation may be lost if each program component must consume all of the random numbers it will need before passing the remainder of the random numbers to another part of the program.

4 Balanced splitting of sequences

A second approach, suggested by L'Ecuyer (1988), is to split the random number sequence into two or more subsequences whenever a program component contains more than one subcomponent requiring random numbers. It is necessary to be able to extract random values from either subsequence, but this is not difficult to arrange.

One way to split a sequence is to place all elements with odd indices in one subsequence, and the rest of the elements, those with even indices, in the other. Applying a function such as ***alternate***, defined as follows, would split the sequence as required

```
alternate [] = ([],[])
alternate [x] = ([x],[])
alternative (x:y:zs) = (x:xs,y:ys)
    where
    (xs,ys) = alternate zs.
```

However, this formulation would cause terrible space leaks if the two subsequences were not consumed at similar rates.

We can avoid space leaks by generating the odd and even terms independently. With the minimal standard random number generator, since

$$(x_{i+1}) = (x_i \times a) \bmod m$$

we have

$$(x_{i+2}) = (x_i \times a^2) \bmod m.$$

Thus, to split a sequence we merely square the multiplier and use that square as the basis for two random number generators. One of them uses the original seed x_0, and the other uses x_1 as its seed.

To implement this idea, we need some additional functions for performing arithmetic modulus $2^{31}-1$. These are given in Fig. 2. Addition is straight forward,

```
modulus = 2147483647      -- = 2^31-1

add:: Int -> Int -> Int
x `add` y = if (result >= 0) then (result) else (result+modulus)
    where
    result = (x-modulus)+y

mult:: Int -> Int -> Int
-- If (y < sqrt modulus) then (x `mult` y = (x*y) mod modulus)
x `mult` y = if (result >= 0) then (result) else (result+modulus)
    where
    result = (x `mod` q)*y-(x `div` q)*r
    q = modulus `div` y
    r = modulus `mod` y

times:: Int -> Int -> Int
-- if (x = a*s+b) and (y = c*s+d)
-- then x*y = a*c*s^2+(a*d+b*c)*s+b*d
-- pick s so that (a,b,c,d < sqrt modulus)
times x y = acss `add` middle `add` bd
    where
    s = 46341       -- = floor(sqrt modulus)+1
    ss = 4634       -- = (s^2) mod modulus
    a = x `div` s
    b = x `mod` s
    c = y `div` s
    d = y `mod` s
    acss = (a*c) `mult` ss
    temp = (a*d) `add` (b*c)
    middle = (temp `mult` (s-1)) `add` temp
    bd = b `mult` d
```

Fig. 2. Functions for arithmetic modulus $2^{31}-1$.

and is performed by ***add***. The ***mult*** function is similar to the original ***next*** function, and performs multiplication provided the second argument is less than $\sqrt{}$*modulus*. This is used in ***times***, which performs multiplication mod ***modulus*** for full sized arguments.

Fig. 3 shows Haskell code for a revised implementation of ***Random***, which includes an operation ***split*** to partition a random number sequence into two subsequences. In this code, random number sequences are represented by a (*seed*, *multiplier*) pair.

One problem with this solution is that ***times*** is significantly more expensive to compute than ***next***. If many elements must be generated for most subsequences, then it is worth spending more time splitting a sequence so elements can be generated more quickly.

A solution is to represent a subsequence as a pair containing a starting index and subsequence length (rather than a starting value), and to split the interval in half when splitting is required. For example, ***Seed i n*** would represent the subsequence $x_i, x_{i+1},$

```
data Random = Seed Int Int

seed:: Int → Random
seed s = Seed (normalize s) 16807

seed:: Random → (Random, Random)
split (Seed s a) =
        (Seed s aa, Seed (s ‘times‘ a) aa)
        where
        aa = a ‘times‘ a

generate:: Random → [Float]
generate (Seed x a) = map scale (iterate (times a) x)
        where
        scale y = fromIntegral y/fromIntegral modulus
```

Fig. 3. Splitting a sequence into odd and even elements.

$\ldots, x_{i+n-1}$. This sequence would split into subsequences represented by ***Seed i halfn*** and ***Seed*** (***i***+***halfn***) (***n***−***halfn***), where ***halfn*** is ***n*** ***‘div’ 2***.

To implement this, we need a function ***rand*** such that ***randi*** $= x_i$. With

$$x_0 = 1$$

$$x_{i+1} = x_i \times a$$

we can compute

$$\mathbf{\mathit{rand\,i}} = a^i.$$

The code in Fig. 4 computes ***randi*** in O(log *i*) time.

```
square x = x ‘times‘ x

rand:: Int → Int
-- rand i = (a^i) mod modulus
-- Note: (rand i = generate (seed 1) !! i)
rand i =
        if i = = 0 then 1
        else if even i then square (rand (i‘div‘2))
        else {-odd i-} next (square (rand (i‘div‘2)))
```

Fig. 4. A function to compute ***rand i*** $= x_i = a^i$.

We can use ***rand*** to implement ***Random***, as shown in Fig. 5. A programmer supplied seed is still allowed. Each random number is multiplied by this seed mod *modulus*. Distinct random numbers remain distinct under this multiplication. The iterated function is again *next*, which can be computed quickly.

The approach of splitting a random number sequence into two subsequences of equal length, either by taking alternative elements or taking consecutive elements, solves the problem of parallel computation. Nothing prevents different processors from using different subsequences at the same time. This approach also alleviates the plumbing problem: it reduces a complex structure of input/output argument bindings to a hierarchy of input-only argument bindings.

On the negative side, the tree approach delivers sequences that are too short at deep levels in the tree. In practice, any pseudo-random number generator will produce only

```
data Random = Seed Int Int Int
-- Seed seed index length

seed :: Int -> Random
seed s = Seed (normalize s) 0 2147483646

split :: Random -> (Random, Random)
split (Seed s i n) =
    (Seed s i halfn, Seed s (i+halfn) (n-halfn))
    where
    halfn = n 'div' 2

generate :: Random -> [Float]
generate (Seed s i n) = map scale (iterate next (rand i 'times' s))
    where
    scale y = fromIntegral y/fromIntegral modulus
```

Fig. 5. Splitting a sequence into equal length subsequences.

a finite sequence of values. Repeatedly halving that sequence at successive levels of recursion leads to shorter and shorter sequences at deeper nodes in the tree. Even if the original sequence is very long, a computation that happens to proceed down the tree, rather than branching out across it, will quickly run into unacceptably short sequences. For example, if the original sequence has a few billion numbers in it, a recursion in the program that proceeds down the tree will encounter sequences as short as a few hundred numbers after only a couple dozen levels of recursion. This would be unacceptable in many simulations.

5 Nondeterministic random numbers

A completely different approach to random numbers is to regard them as nondeterministic values. Unlike the other approaches that we consider in this paper, all of which can be implemented without extending ordinary functional languages, this requires linguistic support for nondeterminism. However, language extensions for nondeterminism are worth considering because they have applications not only in the context of random number generation, but also in other computations, such as operating systems, where functional languages have been difficult to apply (Burton, 1988; Hughes and O'Donnell, 1990).

Variations on this theme of nondeterminism are possible. A program could take an infinite lazy tree of random values as an additional argument. Each random value in the tree would be supplied by the system when it is first used. The values in the tree need not be predetermined. Instead, given a sequential random number generator, each time a new value is used for the first time, the next value in the sequence will be assigned to it. This approach is based on a method for supporting nondeterminism in a functional language proposed by Burton (1988). The approach maintains referential transparency, since a random value remains unchanged once it is set. Furthermore, it permits parallel computation.

On the other hand, the order in which a distributed computing system uses random values may not be deterministic, which means results will fail to be exactly reproducible, even though they will remain statistically equivalent (assuming the random number generator is a good one). Finally, there is a possible bottleneck in a parallel system if all random numbers originate from a single location.

Hughes and O'Donnell (1990, 1991) have proposed an alternate approach to dealing with nondeterminism in functional programs. Their approach treats nondeterministic functions in a special framework that preserves referential transparency by viewing nondeterministic values as sets of possibilities, while allowing an implementation to represent such sets by single elements.

Following the Hughes–O'Donnell model, we could view a random number generator as a nondeterministic function (of no arguments) that, on repeated invocation, delivers a sequence of random numbers one at a time. Their model would prevent an interpretation of this function that would violate referential transparency, but at the same time would permit multiple invocations of the function at different points in the program to deliver different numbers, thus parcelling out random numbers as needed.

This approach is attractive because it avoids the plumbing problem entirely, and uses random number generators in the traditional way. That is, the system could use a library function that, via successive invocations, would generate a sequence of random numbers.

This single source of random numbers, invocable from anywhere, avoids the problem of routing subsequences to different parts of the program. However, it has the same disadvantages as the lazy tree approach: implementations on parallel computing systems may lead to bottlenecks and reproducibility problems.

A method that facilitates parallel computation, provides repeatability (even in a parallel computing environment), and distributes short sequences more or less uniformly around the tree (thus avoiding a bias against deep recursions) is the subject of the next section.

6 Random splitting of sequences

Let us assume that we have a random number generator where each element of the random number sequence can be generated quickly from the previous one and the *i*th can be computed in a reasonable amount of time (e.g. an amount of time that is an acceptable cost whenever a new subsequence is required). The minimal standard random number generator has these properties, but the results in this section apply to other random number generators as well.

Let $x_0, x_1, \ldots$ be a random number sequence, and *next* be the function defined by $x_{i+1} = next(x_i)$. Define a function *rand* by

$$\begin{aligned} rand(0) &= x_0 \\ rand(i+1) &= next(rand(i)). \end{aligned}$$

That is, $rand(i) = x_i$, and the sequence of random numbers is

$$\begin{aligned} &x_0, next(x_0), next(next(x_0)), next(next(next(x_0))), \ldots \\ &= rand(0), rand(1), rand(2), rand(3), \ldots. \end{aligned}$$

```
data Random = Seed Int

seed:: Int→Random
seed s = Seed (normalize s)

split:: Random→(Random, Random)
split ((Seed s)) = (Seed (next s), Seed (rand s))

generate:: Random→[Float]
generate (Seed s) = map scale (iterate next s)
    where
    scale y = fromIntegral y/fromIntegral modulus
```

Fig. 6. Splitting a sequence randomly.

We have already seen how these functions are constructed for the minimal standard random number generator.

The function *rand* maps the very unrandom sequence $0, 1, 2, \ldots,$ into the original random sequence. If the random number generator is any good at all, we would expect *rand*(i) to be a independent of i for all practical purposes.

The strategy we propose is as follows: whenever we need a new subsequence of random numbers, we split the sequence at a random point, that is, from a seed x, we generate the pair of seeds *next*(x) and *rand*(x) (see Fig. 6). If eventually we need many subsequences, each will have a random starting point.

This approach has the same advantages as the method of section 4: it does not impede parallel computation, and it does not require unmanageable plumbing. In addition, it avoids the primary disadvantage of the old method, that is the bias against deep probes in the tree.

For this method to work effectively, there must be only a small chance of encountering, in practice, duplicate subsequences (and, therefore, statistical dependence among random number sequences that should behave as if they were independent). In the following we present conditions under which it is likely that this will be the case.

Given m objects drawn independently, with replacement, from a pool of p objects, there are $m(m-1)/2$ distinct pairs of objects. The probability that any given pair consists of duplicate objects is $1/p$. Hence, we would expect about $m(m-1)/(2p)$ of the pairs to contain duplicates. Therefore, duplicate objects will be unlikely if m^2 is much less than p.

Suppose a program requires m independent random number subsequences, and that the program will obtain these subsequences by partitioning a sequence of n pseudo-random numbers. Assume for the moment that the program will consume about the same number of random numbers, say k of them, from each of the subsequences.[1] The probability of overlap between two subsequences of length k with independently chosen starting values is $(2k-1)/n$, because overlap occurs only if the starting value of the second sequence is the same as the starting value of the first, or within $k-1$ positions on either side of it. If we select at random m seeds for the subsequences, then we have $m(m-1)/2$ different pairs of subsequences. The expected

[1] More generally, let k be the number of random numbers consumed by the most heavily used of the subsequences.

number of these pair consisting of overlapping sequences is $m(m-1)/2 \times (2k-1)/n$. (The probability that there exists a pair of overlapping subsequences is less than this expected number of overlaps.) Therefore, the required subsequences will be likely to behave independently if $m^2 \ll n/k$.

We have not carried out extensive statistical testing of the collections of random numbers that are generated using this method. However, we did generate a tree of depth 14, with value 2 at the root and with the children of the node with value *i* having values *next i* and *rand i*. There were no duplicated values among the 32767 random numbers in the tree.[2]

7 Conclusion

It is easy to define a random number generator in a functional language. The generator simply returns a lazy infinite list of random numbers. What is more difficult is sharing the sequence among many different processes that require random numbers, as for example in a complex simulation program.

We have looked at some alternative methods for partitioning a random number sequence. Some of the simpler methods are sufficient for simple problems. However, if many random number subsequences may be required, and the structure of the computation is not known in advance, then partitioning the random number sequence in a random fashion appears to be the most promising solution to the problem.

References

Burton, F. W. 1988. Nondeterminism with referential transparency in functional programming languages. *Computer J.*, **31** (3): 243–247 (Mar.).

Clinger, W. 1982. Nondeterministic call by name is neither lazy nor by name. In *Conf. record 1982 ACM Symposium on LISP and Functional Programming*, pp. 226–234 (Aug.).

Hudak, P., Wadler, P., Arvind, B. B., Fairbairn, J., Fasel, J., Hammond, K., Hughes, J., Johnsson, T., Kieburtz, D., Nikhil, R., Peyton-Jones, S., Reeve, M., Wise, D. and Young, J. 1990. Report on the programming language Haskell: A non-strict purely functional language (Version 1.0). Technical Report YALEU/DCS/RR-777, Department of Computer Science, Yale University.

Hughes, R. J. M. and O'Donnell, J. 1990. Expressing and reasoning about non-deterministic functional programs. In *Proc. 1989 Glasgow Workshop*, pp. 308–328, *Workshops in Computing*, Springer-Verlag.

Hughes, R. J. M. and O'Donnell, J. 1991. Nondeterministic functional programming with sets. In *Proc. IV Higher Order Workshop*, pp. 11–31, *Workshops in Computing*, Springer-Verlag.

L'Ecuyer, P. 1988. Efficient and portable combined random number generators. *Comm. ACM*, **31** (6): 742–749, 774 (Jun.).

Søndergaard, H. and Sestoft, P. 1990. Referential transparency, definiteness and unfoldability. *Acta Informatica*, **27**: 505–517.

Park, S. K. and Miller, K. W. 1988. Random number generators: Good ones are hard to find. *Comm. ACM*, **31** (10): 1192–1201 (Oct.).

[2] When we initially tried a root value of 1, we quickly discovered that *next* 1 and *rand* 1 were the same, so the entire right subtree was identical to the left subtree. That 1 is not a good seed comes as no great surprise.