The RNG Random Number Library*

Bret R. Beck and *Eugene D. Brooks III*

University of California
Lawrence Livermore National Laboratory
Livermore, California 94550
UCRL-MA-141673

December 8, 2000

Abstract

We describe a strategy for random number generation that efficiently supports per-particle-state for Monte Carlo transport applications. Unlike prior random number generation strategies, where the design goal of a countable number of independent full period sequences matches a per-cpu random number state well, we provide an uncountable number of statistically independent short sequences that match the typical lifetime of a Monte Carlo particle well. The short sequences are selected from a single random number stream of suitable quality and period, using a cryptographic hash function to select random starting positions. The random number generators described herein are *stateless*. The random number state is carried along with the particle. The stateless design removes the read-modify-write hazard associated with traditional random number generators in shared memory parallel environments, and helps support deterministic behavior when a particle is moved from one processor to another in a distributed memory application. Our library is implemented in the C programming language. A Fortran interface to the library is provided.

*Work performed under the auspices of the U. S. Department of Energy by the Lawrence Livermore National Laboratory under contract No. W-7405-ENG-48.

1 Introduction

Monte Carlo algorithms have been heavily used in transport applications since the earliest days of electronic computing. In spite of this very long history, little has changed in how random number generators are used in these applications. Random numbers have typically been generated with a linear congruential recurrence formula of the form

$$x[n] = a\,x[n-1] + b \pmod m \quad , \tag{1}$$

where x is the hidden random number state stored in statically allocated memory. The constants a and b are chosen to produce a maximal period and good randomness properties. The modulus of the arithmetic, m, is typically chosen to be a power-of-two in order to minimize the cost of performing modular arithmetic using binary operations on a computer. The code developer is usually provided with routines that allow one to save and restore the internal state x of the generator. These routines are used for job check pointing purposes and can also be used to set the starting state for a statistically independent instance of the program run.

The increasing popularity of shared memory parallel computing has exposed a hazard in the traditional approach when the internal state, x, is shared among the threads or processors executing the application. The internal state can be protected with a lock, removing the read/modify/write hazard, but such an arrangement still results in non-deterministic (but correct) behavior because the processors can access the random number generator in any order influenced by machine time sharing effects. One must localize the internal random number state, on a per-thread or per-processor basis, in order to remove this hazard. The coding required to localize memory in this manner is specific to a given parallel programming environment, leading to a multiplicity of random number library implementations (possibly on a given shared memory multiprocessor) and the attendant risk of loading against the wrong one.

Further complications arise in a distributed memory programming environment where the number of processors employed can be large. The straight forward extension of the per-thread or per-processor approach has led to the development of random number generators with parameterized constants that provide independent full period sequences [1], assuring the independence of random numbers generated in each processor. Although this solution provides independent generators for each processor, it can lead to problems with determinism when a particle must be moved between processors for load balancing purposes. When the particle is moved, it encounters a different random number stream and its history is changed as a result. More significantly, its presence affects the history of all of the particles being processed by the processor that it is moved to, leading to undesired avalanche effects in application behavior.

In the RNG library described in this paper, we take an alternative approach of efficiently supporting a per-particle random number state. The random number generator library is stateless, leading to a single implementation for serial, shared memory parallel, and distributed memory applications. The user may use the library to implement a single random number state, a per-processor or per-thread random number state for parallel applications, or a per-particle random number state in the limit of fine granularity. A pointer to the desired random number state is supplied as an argument to the random number routines provided in the library.

2 Design Requirements

In order to support per-particle state in Monte Carlo transport calculations the size of the state must be relatively small in comparison to the amount of storage normally associated with a particle. A Monte Carlo particle might have an energy or velocity, three position coordinates, and three direction coordinates associated with it, in addition to a few parameters which indicate the type of particle that is being represented. To prevent application storage from being dominated by random number state, the size of the state must be small when compared to perhaps a dozen words of memory. This precludes using lagged-Fibonacci [2] generators that require many tens to hundreds of words of memory even though this class of random number generators can produce sequences of very high quality.

Although the random number strategy we have implemented can be extended to any generator that is suitably bit efficient, all of the generators that we have implemented are linear congruential generators for which the state fits in a single 64-bit word. This meets the requirement of being small compared to the amount of storage normally associated with a Monte Carlo particle. We offer carefully chosen prime moduli to resolve patterns that appear in the linear congruential generators that are commonly used.

If we are to provide for an uncountable number of statistically independent random number states using a single random number stream of demonstrated quality, we need a mechanism to efficiently select random starting positions on the random number stream. A different linear congruential generator can be used for this purpose [3], but this provides insufficient de-correlation between the resulting random number seed states for individual particles. In the RNG library we use a cryptographic hash function to generate statistically independent random number states. The hash function can be used to construct a sequence of uncorrelated random number states by passing an ordinal sequence through it. The hash function may also be used to spawn a new random number state from a parent, with the position, in the random number sequence, of the child's state being uncorrelated to the position of the parent's state.

3 Random Number Generators

The RNG library provides a family of linear congruential generators of the form

$$x[n] = a\,x[n-1] + b \pmod{m} \quad , \tag{2}$$

wherein the random number state fits in a 64-bit unsigned word. The multiplier, a, the addend, b, and the modulus, m, are chosen to provide a variety of high quality generators for the code developer to employ. Two power-of-two modulo generators, a Mersenne prime modulo generator, and several other prime modulo generators are provided. Generators using a modulus that is a power of two are the fastest, but suffer from a hierarchy of patterns [4] in the low order bits that may pose a problem for some applications. Generators using a prime modulus do not suffer from the hierarchy of patterns noted for a power of two modulus, with the choice of Mersenne primes [4] being particularly efficient. Patterns of a different form have been discovered for Mersenne primes [5], however. Primes of a slightly more complicated construction remove this defect while not being significantly more expensive for modular arithmetic.

The prime modulus generators provided in the RNG library have the useful feature that the value zero does not occur as a valid random number state. By carefully zeroing a random number state when it is copied or retired, the random number generator routines will trap on the invalid value should the retired random number state be recycled. This has proven to be very useful when debugging applications that implement per-particle random number state.

The interface routines for the random number generators in the RNG library take the general form:

```
Rng_Type RngXSeed( Rng_UInt32 i, Rng_UInt32 j );
Rng_Type RngXSpawn( Rng_Type *x );
int iRngX( Rng_Type *x );
double dRngX( Rng_Type *x );
float fRngX( Rng_Type *x );
```

where **X** is replaced with a token that indicates the type of linear congruential recurrence relation being used for the random number generator.

The `RngXSeed()` function creates a randomly selected state by passing the two integer arguments through the cryptographic hash function. The resulting random number state is returned. `Rng_Type`, `Rng_UInt32` and the function declarations are defined in the header file rng.h. The routine `RngXSpawn()` takes an existing RNG state, called the parent, and advances it. A copy of the advanced parent state is then hashed to produce an uncorrelated child RNG state that is returned. As an example of the use of `RngXSpawn()`, consider the case where a "parent particle", `parent`, spawns three "child particles", `child[3]`. The particles are represented by a `struct` that has a Rng_Type member named `rng`. Then, the following lines of code demonstrate the seeding of the children's RNG states using the parent's RNG state.

```
for( i = 0; i < 3; i++ ) {
    child[i].rng = RngXSpawn( &(parent.rng) );
}
```

Note that `parent.rng` is advanced three times in this example. Finally, note that both `RngXSeed()` and `RngXSpawn()` take 64-bits values, `RngXSeed()` in the form of two 32-bit unsigned integers, and hash them through the same hashing algorithm described in Section 4.

The last three functions step the random number state and return uniformly distributed numbers in integer and floating point formats. The function `iRngX()` returns a 31-bit integer in the range $[0,\ 2^{31})$. The function `dRngX()` returns a double precision floating point value in the range [0,1). The function `fRngX()` returns a single precision floating point value in the range [0,1). If one casts the value returned by `dRngX()` to a `float`, it is possible for the value 1.0 to occur due to rounding.

3.1 64-bit Linear Congruential Generator

This algorithm is the fastest, especially on 64-bit architectures, but does not produce as good a pseudo random number sequence as the prime modulus generators. The next value for the random number state, $x[n]$, is determined from the current value, $x[n-1]$, by

$$x[n] = a\,x[n-1] + b \pmod{2^{64}} \quad . \tag{3}$$

The values of the multiplier, $a = 2862933555777941757$, and the addend, $b = 3037000493$, are hardwired into the implementation and are known to produce a good random number list. The period of this random number generator is $2^{64} \approx 1.8 \times 10^{19}$. This generator is the same as the default one-stream SPRNG [1] lcg64 generator and has been checked to insure that it satisfies the requirements for maximal period[1].

The interface routines for this generator are:

```
Rng_Type Rng64Seed( Rng_UInt32 i, Rng_UInt32 j );
Rng_Type Rng64Spawn( Rng_Type *x );
int iRng64( Rng_Type *x );
double dRng64( Rng_Type *x );
float fRng64( Rng_Type *x );
```

This generator is the fastest, particularly on 64 bit computer platforms. It is useful for performance sensitive applications that are insensitive to the pattern content present in the low order bits. As zero is an allowed random number state, this generator is not useful for debugging correct state retirement in a per-particle state implementation.

3.2 48-bit Linear Congruential Generator

This generator is the same as the default one-stream SPRNG 48-bit lcg. This generator suffers from the same patterns in the low order bits as the 64-bit generator, but these patterns become more significant because of the smaller width of the random number state. Like the 64-bit generator, this generator is fast. The next value for the random number state, $x[n]$, is determined from the current value, $x[n-1]$, by

$$x[n] = a\,x[n-1] + b \pmod{2^{48}} \quad . \tag{4}$$

The values of the multiplier, $a = 44485709377909$, and the addend, $b = 11863279$, are hardwired into the implementation and are known to produce a good random number list. The period of this random number generator is $2^{48} \approx 2.8 \times 10^{14}$.

The interface routines for this generator are:

```
Rng_Type Rng48Seed( Rng_UInt32 i, Rng_UInt32 j );
Rng_Type Rng48Spawn( Rng_Type *x );
int iRng48( Rng_Type *x );
double dRng48( Rng_Type *x );
float fRng48( Rng_Type *x );
```

Our advice is to only use this generator to check application sensitivity to the quality of the random number stream.

3.3 48-bit CRI Linear Congruential Generator

This algorithm is provided by the operating system on computers manufactured by Cray Research Incorporated (CRI), and as such it has been heavily used at computer centers employing CRI hardware. We provide it in order to support comparison with the other random number generators provided in the RNG library. This generator suffers from the same patterns in the low order bits as the 64-bit generator, but these patterns become more significant because of the smaller width of the random number state. The next value for the random number state, $x[n]$, is determined from the current value, $x[n-1]$, by

$$x[n] = a\,x[n-1] \pmod{2^{48}} \quad . \tag{5}$$

[1]D.E. Knuth, **The Art of Computer Programming**, vol. 2, page 16-21, (1998).

The value of the multiplier, $a = 44485709377909$, is hardwired into the implementation and is known to produce a good random number list (note that lcg48 and lcgCRI have the same multiplier). Both the multiplier, and the random number state are odd. The period of this random number generator is $2^{46} \approx 7.0 \times 10^{13}$ per stream. The generator actually creates two independent streams based on the value of the "twos" bit in the random number state that is preserved by the recurrence relation. The hash functions associated with this generator set the "twos" bit randomly, taking full advantage of the pair of random number streams produced by the recurrence relation. The value zero does not occur for this random number generator.

The interface routines for this generator are:

```
Rng_Type RngCRISeed( Rng_UInt32 i, Rng_UInt32 j );
Rng_Type RngCRISpawn( Rng_Type *x );
int iRngCRI( Rng_Type *x );
double dRngCRI( Rng_Type *x );
float fRngCRI( Rng_Type *x );
```

The pattern content of this generator is frightening, making it somewhat amazing that it has been successfully used for so many years. Because this generator duplicates the arithmetic of a commonly used random number generator, and a zeroed state is trapped, it can be useful during the debugging stage of program transformation to a per-particle state approach. Past its utility for debugging purposes, our advice is to only use this generator to check application sensitivity to the quality of the random number stream.

3.4 61-bit Prime Modulus Linear Congruential Generator

This algorithm is slower than the generators using a power of two modulus, but produces a random number sequence free of short period patterns in the low order bits. A pattern does occur for this generator, however, and is described in [5]. The next value for the random state $x[n]$ is determined from the current value $x[n-1]$ by:

$$x[n] = a\,x[n-1] \pmod{p} \tag{6}$$

Where the multiplier, $a = 437799614237992725$, is hardwired into the algorithm and p is a Mersenne prime; that is, a prime that is of the form $p = 2^m - 1$ where m is also a prime. For this generator $m = 61$, which is the largest value of m less than 64 for which $2^m - 1$ and m are primes. The multiplier was again chosen to obtain maximal period of the generator, $2^{61} - 2 \approx 2.3 \times 10^{18}$, for this algorithm. This generator is the same as the default one-stream SPRNG pmlcg generator and was checked to insure that it satisfies the requirements for maximal period.

The interface routines for this generator are declared as:

```
Rng_Type RngP61Seed( Rng_UInt32 i, Rng_UInt32 j );
Rng_Type RngP61Spawn( Rng_Type *x );
int iRngP61( Rng_Type *x );
double dRngP61( Rng_Type *x );
float fRngP61( Rng_Type *x );
```

3.5 $2^{64} - 2^{10} + 1$ Modulus Linear Congruential Generator

Due to the discovery of a pattern for random number generators based on Mersenne primes, a family of prime modulus generators has been explored that removes this pattern while still maintaining good efficiency for modular arithmetic. In addition to removing the offending pattern, these generators also offer a slightly longer period because they use more of the available 64-bits in the random number state. As discussed in [5], the key to removing the pattern resulting from the Mersenne prime modular arithmetic is using special primes with a "+1" instead of "−1" as the last term in the expression composed of powers of two. The first of these is a prime modulus linear congruential generator provided by the recurrence relation:

$$x[n] = a\,x[n-1] \pmod p \tag{7}$$

Where the multiplier, $a = 3355703948966806693$, is hardwired into the algorithm and p is the prime, $2^{64} - 2^{10} + 1$. The multiplier was again chosen to obtain maximal period of the generator, $2^{64} - 2^{10} \approx 1.8 \times 10^{19}$.

The interface routines for this generator are declared as:

```
Rng_Type RngP64_10Seed( Rng_UInt32 i, Rng_UInt32 j );
Rng_Type RngP64_10Spawn( Rng_Type *x );
int iRngP64_10( Rng_Type *x );
double dRngP64_10( Rng_Type *x );
float fRngP64_10( Rng_Type *x );
```

This random number generator suffers from no patterns that we are aware of. We are currently evaluating its quality using statistical tests. The fact that this generator has a prime modulus and its period is close to 2^{64} makes it very attractive. This generator also traps a zeroed state, making it equally useful for debugging.

3.6 $2^{62} - 2^{16} + 1$ Modulus Linear Congruential Generator

A second prime modulus generator that addresses the pattern appearing for Mersenne primes is provided by the prime modulus $2^{62} - 2^{16} + 1$. This is provided by the recurrence relation:

$$x[n] = a\,x[n-1] \pmod p \tag{8}$$

Where the multiplier, $a = 3355703948966806692$, is hardwired into the algorithm and p, the prime modulus, is $2^{62} - 2^{16} + 1$. The multiplier was again chosen to obtain maximal period of the generator, $2^{62} - 2^{16} \approx 4.6 \times 10^{18}$.

The interface routines for this generator are declared as:

```
Rng_Type RngP62_16Seed( Rng_UInt32 i, Rng_UInt32 j );
Rng_Type RngP62_16Spawn( Rng_Type *x );
int iRngP62_16( Rng_Type *x );
double dRngP62_16( Rng_Type *x );
float fRngP62_16( Rng_Type *x );
```

This random number generator suffers from no patterns that we are aware of and we are currently evaluating its quality using statistical tests.

4 The Cryptographic Hash Function

We use a cryptographic hash function to construct an initial random number state from a pair of integers, or from a parent random number state for a child particle. Block encryption functions are perfectly good for this, themselves providing random numbers of good quality that are uncorrelated with linear congruential recurrence relations. The encryption key provides a means for the user to easily create a different random instance of the problem run, affecting the starting values for all of the particles in the application. The current crop of block encryption functions operating on 64 bits match our choice of a 64-bit unsigned integer for random number state quite nicely.

The fact that an encryption algorithm is reversible is not essential, but does assure that the hash function provides a random, one-to-one, shuffle of the random number states produced by the generator. This property is guaranteed in the case of the 64-bit linear congruential generator, where the random number state is a full 64-bit value. Output collisions can occur in the other generators that do not fully use all possible 64-bit values, but the probability of such collisions is low.

Any block encryption algorithm of sufficient quality will suffice as our random hash function for random number state generation. We have chosen the encryption algorithm IDEA [6] [7] because of its efficient portable implementation on general purpose computers. The IDEA encryption algorithm is comprised of 8 rounds of a Feistel network, mixing the operations of three 16-bit wide algebraic groups to scramble the result. We have found that two rounds of IDEA produces a suitably random output from its input, and that four rounds would offer an adequate safety margin against correlations being detectable between parent and child random number states produced by the hash function. The full 8 rounds of IDEA are available for the truly paranoid Monte Carlo practitioner. The user can adjust the number of rounds of IDEA used, in response to performance needs, in the event that the costs of random number state generation become significant. The number of rounds of IDEA used can be adjusted from 1, to the full 8 rounds, with 4 rounds being the default.

The IDEA encryption algorithm employs a key of 128 bits. A default key is provided. This key was randomly chosen using dice. The subroutine that sets the IDEA key in our library takes two 32-bit integers provided as arguments by the user and constructs the 128-bit IDEA key by exclusive or'ing this 64 bits with the upper and lower 64 bits of the default key. It is very difficult, if not impossible, for the user to generate a bad choice for the encryption key. The default encryption key is the same as the one generated if the user passes zeros to `RngSetKey`.

The functions to control the properties of the hash function are:

```
void RngSetKey( Rng_UInt32 k1, Rng_UInt32 k2 )
void RngSetKeyRounds( int n )
```

Here, Rng_UInt32 is a 32-bit unsigned integer and is defined in the header file `rng.h`. The `RngSetKey()` and `RngSetKeyRounds()` functions are used for all of the random number generators provided. The `RngSetKeyRounds()` function sets the number of rounds of IDEA used in the hash function, silently enforcing the range 1 through 8.

`RngSetKeyRounds()` and `RngSetKey()` may be called in any order, but must, if used, be called before the other RNG routines are called. `RngSetKeyRounds()` does not have to be called, in which case the default value for the number of rounds is 4. The full IDEA encryption algorithm uses 8 rounds, but our tests have shown that changing a single bit avalanches with good randomness to all of the output bits in just two rounds. Since this was an empirical test, we added two more rounds to the default for a safety margin. Setting the number of rounds higher than 4 will slow down the process of seed generation, but should result in less correlation between the input and output. Setting the number of rounds to 1 produces clear evidence of correlations, but may be sufficiently random for applications where the speed of seed generation is paramount. Generally, the code developer should not expose control of the number of rounds to the user, who might easily set it too low without carefully checking statistical properties of application results.

5 Packing and Unpacking RNG State

When saving and restoring a RNG state to a data file it may be important for the application to maintain portability in a binary file format. The byte ordering of the Rng_Type depends upon the details of the host architecture, so we provide routines to pack and unpack the Rng_Type in a given portable[2] byte order. This character array of 8 unsigned bytes can be used to implement machine independent random number state in binary files, assuming that other tools are used to provide machine independent binary formats for the other C types used in the application. The routines to pack and unpack a Rng_Type to and from an unsigned char array of length 8 are, RngPack() and RngUnpack(), respectively:

```
void RngPack( Rng_Type r, unsigned char *a );
Rng_Type RngUnpack( unsigned char *a );
```

The routine RngPack(r, a) takes a Rng_Type as its first argument, `r`, storing the 64-bit value in the unsigned char array (of length 8 bytes) referred to by its second argument, `a`. The routine RngUnpack(a) takes the 64-bit value stored in the unsigned char array (of length 8 bytes) referred to by the pointer, `a`, and returns the corresponding Rng_Type.

6 A Note on Mixing Generators

All of the random number generators currently implemented in the RNG library use the type Rng_Type, a 64-bit unsigned object, to implement the random number state. Each generator, however, spans its own range of values governed by the properties of its recurrence relation and any constraints on the starting value used.

The 64-bit lcg described in Section 3.1 spans all possible values of the 64-bit Rng_Type, zero included. The 48-bit lcg of Section 3.2 uses only 48 bits. The state, in this case, spans the range $[0,2^{48})$. The 48-bit CRI compatible LCG of Section 3.3 uses only 48 bits and the state, in this case, must be odd. The prime modulus generators, with modulus P, produce state, x, which spans the range $1 \leq x < P$.

It is important that one does not mix generators for a given random number stream, for example initializing a Rng_Type with the 64-bit generator, and using one of the other generators to produce random values using this random number state. Results of such mixing between generators may be unpredictable.

[2]The byte order used is little endian, that is, least significant byte first. This is a property of the implementation, and not the RNG library specification.

7 Thread Safety and the RNG Package

There are two static values in the RNG package that one must be aware of when writing thread safe code. These values are both associated with the cryptographic hash function. The first is the number of rounds which can be modified using `RngSetKeyRounds()`. The second is the encryption key which can be modified using `RngSetKey()`. Both of these functions should be called (if desired) at the startup of a parallel program, before threads are created. If this philosophy is adhered to the RNG package is thread safe.

8 make Interface

The RNG library is distributed as a UNIX **tar** image containing all of the the source code needed to build the library and documentation. In the top level directory, **rng**, make has the following useful targets:

make	Creates Lib/rng.h and Lib/librng.a.
make test	Runs the regression and performance tests.
make clean	Cleans up object files.
make clobber	Deletes everything built with make.
make tar	Creates a tar file name ../rng.tar from the rng directory.
make doc	Creates a dvi version of this document in the Doc directory.
make ps	Creates a postscript version of this document.

The document building commands are included in order to make the current version of this document easily available to anyone receiving a copy of the library. One can copy the library, header, and processed document files to an appropriate system directory as required.

9 Including and Linking to RNG

The user must include the header file `rng.h` and link in the archive file `librng.a` to use the RNG library. The "-I" compiler flag can be used to tell the compiler where to locate the header file, for instance -I/usr/apps/include on Livermore Computing systems. Similarly, the "-L" flag can be used to tell the loader where to find the library in response the -lrng flag, for instance -L/usr/apps/lib on Livermore Computing systems. The Examples directory contains several programs that demonstrate the use of the RNG library. A README and a Makefile contain instructions for building the example programs.

10 Fortran Interface

There are numerous Monte Carlo transport codes written in Fortran that are sorely in need of an upgrade with respect to random number generation. This becomes a critical issue as the limits of parallel computing are extended for Monte Carlo applications. To satisfy this need, we have supplied a Fortran interface to the RNG library.

The key obstacle to providing a Fortran interface is the problem of supporting a 64 bit integral type to store the random number state. Typical Fortran compilers only support a 32 bit integer. We resolve this by defining the random number state to be an array of two 32 bit integers in the Fortran interface.

The subroutines that set the number of rounds and the encryption key are defined as:

```
SUBROUTINE RNGSETKEY( INTEGER K1, INTEGER K2 )
SUBROUTINE RNGSETKEYROUNDS( INTEGER N )
```

The functions and subroutines that support a particular generator are defined as:

```
SUBROUTINE RNGXSEED_I32( INTEGER ISR(2), INTEGER I, INTEGER J )
SUBROUTINE RNGXSPAWN_I32( INTEGER ISR(2), INTEGER ISA(2) )
INTEGER FUNCTION IRNGX_I32( INTEGER ISA(2) )
DOUBLE PRECISION FUNCTION DRNGX_I32( INTEGER ISA(2) )
REAL FUNCTION FRNGX_I32( INTEGER ISA(2) )
```

The token `X` is one of, **64**, **P61**, **P64_10** and **P62_16** as is the case for the C interface. The argument ISR is for the *result*, passed by reference as is customary for Fortran. The argument ISA is for the *argument*, passed by reference and for which the random number subroutines and functions produce a side effect.

11 Testing of the RNG Library

In this section we discuss some of the testing done, both theoretical and experimental, on the RNG library routines. First, number theory can be used to demonstrate that the algorithms in the RNG library have maximal period (see discussion regarding each algorithm) and to limit the choice for the multipliers (e.g., the potency test as described in [8]). Further checks of the "goodness" of a pseudo random number generator must be done experimentally. Extensive testing has been performed on the 64-bit linear congruential generator of section 3.1, on the 61-bit prime modulus linear congruential generator of section 3.4, and on the 48-bit modulus linear congruential generator of section 3.2 by the developers of SPRNG [1] library. We have duplicated the constants used in the SPRNG library for these generators in order to leverage their testing efforts. To insure that we duplicated the SPRNG arithmetic, we seeded the RNG routines and the equivalent SPRNG routines with the same values [3], and compared their results for a large number of advances on several different architectures (see Table 1).

The RNG library was also compared to Mathematica calculations for a large number of advances in order to check the accuracy of our implementation of modular arithmetic. Furthermore, on the IRIX64 machine listed in Table 1 the RNG library was tested with 10^{12} advances to insure that the values returned by the `iRng()`, `dRng()` and `fRng()` routines are within the specified limits. A more satisfying test of the returned value was done by seeding a RNG state with the value that when advanced once produces the maximum (minimum) value allowed by the generator. The `iRng()`, `dRng()` and `fRng()` routines also passed this last test on the platforms listed in Table 1.

[3]Since both the RNG and the SPRNG libraries hash the seed, the seeding was actually done by circumventing the normal seeding routines.

OS	OSF1	IRIX64	SunOS	AIX	HP-UX	Linux
Processor	DEC alpha (EV56)	SGI R10000	sparc SUNW	IBM PPC 604e	PA-RISC	AMD K-6 II
Clock Speed (MHz)	440	185	250	332	180	333
iRng64 (MRS)	33.15	8.43	3.99	20.62	3.03	9.77
dRng64 (MRS)	18.25	4.86	3.09	10.99	0.64	2.56
iRngP61 (MRS)	7.03	0.99	1.02	2.60	1.10	1.80
dRngP61 (MRS)	7.68	1.01	1.04	3.26	0.48	1.60
nrand48 (MRS)	2.33	5.43	0.69	4.50	0.76	1.51
erand48 (MRS)	2.45	4.29	0.63	3.05	0.64	1.30
SPRNG/ilcg64 (MRS)	10.89	10.53	1.33	1.45	1.85	0.82
SPRNG/dlcg64 (MRS)	7.58	7.19	1.51	1.35	2.13	1.03
SPRNG/ipmlcg61 (MRS)	5.59	2.37	0.36	0.37	0.61	1.36
SPRNG/dpmlcg61 (MRS)	4.41	2.22	0.35	0.36	0.56	1.07

Table 1: List of operating systems (OS) and processors that the RNG library was tested on and some timing information. The rate is quoted as Millions of Randoms per Second (MRS). The rows labeled by nrand48 and erand48 are the measured performance of the system provided generators, for comparison purposes. The rows labeled by SPRNG are the measured performance of the named generators from the SPRNG library for comparison purposes.

It is important that one run these tests on any new platform port in order to defend against any problems that crop up due to compiler differences. One can run the command **make test** in the build directory in order to accomplish this. This command will run the random number generators in the library to insure that they produce the correct state as compared against built-in tables that record past behavior. After this check, the approximate timings of the `iRng()`, `dRng()` and `fRng()` routines are reported in Million Randoms per Second (MRS). Finally, the generators are seeded with the values that produce, on the next call, the minimum and maximum values for the `iRng()`, `dRng()` and `fRng()` routines. These routines are called and the returned values are printed to standard output. The maximum `double` and `float` values should be the same as those printed to standard output by the program one-eps.c in the **Test** sub-directory. To compile and run this program enter the command **make one-eps** in the **Test** sub-directory. Note, in most cases the maximum double value should match that produced by one-eps.c; however, for some generators the value is slightly less due to the fact that there is no double value which when multiplied by the maximum value of that generator produces the nearest-to-one value that is less than one.

We are currently engaged in extensive statistical testing of the new prime modulus generators we have introduced in the RNG library, comparing them to those derived from the SPRNG package, and will report on the results of this testing in a separate paper.

References

[1] M. Mascagni, (1999) “SPRNG: A Scalable Library for Pseudorandom Number Generation,” to appear in Proceedings of the Third International Conference on Monte Carlo and Quasi Monte Carlo Methods in Scientific Computing, J. Spanier et al., editors, Springer Verlag, Berlin.
`http://sprng.cs.fsu.edu/RNG/`

[2] M. Mascagni, S. A. Cuccaro, D. V. Pryor and M. L. Robinson, (1995) “A Fast, High Quality, and Reproducible Parallel Lagged-Fibonacci Pseudorandom Number Generator”, Journal of Computational Physics, 119: 211-219.

[3] P. Fredrickson, et. al., “Pseudo-random trees in Monte Carlo,” Parallel Computing 1 (1984), 175-180.

[4] M. Mascagni, (1998) “Parallel Linear Congruential Generators with Prime Moduli,” Parallel Computing, 24: 923-936.

[5] B. Beck, “Linear Congruential Random Number Generators with Prime Moduli”, to be published.

[6] X. Lai and J. Massey, (1990) “A Proposal for a New Block Encryption Standard”, **EUROCRYPT**: 389.

[7] B. Schneier (1996), **Applied Cryptography**, pp. 319-325.

[8] D. E. Knuth, (1998) **The Art of Computer Programming**, vol. 2.