

Interfaces and Implementations of Random Number Generators for Java Grande Applications

P.D. Coddington, J.A. Mathew and K.A. Hawick

Advanced Computational Systems Cooperative Research Centre
Department of Computer Science, University of Adelaide
Adelaide, SA 5005, Australia
{*paulc,jm,khawick*}@cs.adelaide.edu.au

15 February 1999

Abstract

The Java Grande Forum aims to drive improvements to the Java language and its standard libraries in order that Java may be efficiently used for large-scale scientific applications, particularly on high-performance computers. Random number generators are one of the most commonly used numerical library functions in applications of this kind. For the current random number generator provided within Java, neither the implementation nor the interfaces are adequate to meet the needs of some Java Grande applications, such as Monte Carlo simulations. We present a preliminary proposal for an API for accessing a random number generator within a Java scientific software library for supporting Java Grande applications. A reference implementation of the proposed API is described, and we discuss some implementation and performance issues. Mechanisms for efficiently handling concurrency are also discussed.

1 Introduction

Java has the potential to be an excellent language for developing large science and engineering applications, particularly on distributed and high-performance computers. The Java Grande Forum [9] aims to drive improvements to the Java language and its standard libraries in order that Java may be efficiently used for such “Grande Applications”. Random number generators are commonly used in these types of applications, so it is important to provide access to an efficiently implemented, high-quality generator through a standardized Java application programming interface (API).

In this paper, we investigate the requirements for interfaces and implementations of random number generators for Java Grande applications. We explain why the existing random number generators available in Java are inadequate for use in some scientific applications, and propose a new API that might be used as part of a Java numerical library, or scientific software library (SSL).

Random number generators use iterative deterministic algorithms for producing a sequence X_i of pseudo-random numbers that approximate a truly random sequence. The main algorithms used to implement random number generators in numerical libraries are:

Linear congruential generators (LCGs), with $X_i = (A * X_{i-1} + B) \bmod M$, which we denote by $L(A, B, M)$;

Lagged Fibonacci generators (LFGs), with $X_i = X_{i-P} \odot X_{i-Q}$, which we denote by $F(P, Q, \odot)$, where P and Q are the lags, and \odot is any binary arithmetic operation, such as addition or multiplication modulo M .

Shift register generators can usually be defined in terms of LFGs using the bitwise exclusive OR function XOR, however these are of lower quality than equivalent LFGs using addition or multiplication;

Combined generators that combine (usually by addition modulo M) the results of two or more generators, usually two LCGs, or an LFG combined with an LCG or some other algorithm.

For more information on these algorithms, see one of the many review articles on random number generators [5, 8, 11, 12, 18].

A generator provided in a numerical library for general use should be both fast and of high quality, i.e. have been rigorously tested and passed a variety of standard statistical and empirical tests of randomness. However, determining the quality of a random number generator is a notoriously difficult problem, and there have been many instances of poor quality random number generators being provided in numerical libraries [5, 8, 11, 12, 18].

For many applications, the quality of the pseudo-random numbers is not that important, and any generator that is reasonably random will do the job. However in many of the Grande Applications for which random number generators are most heavily used, such as Monte Carlo simulations [1], the quality of the random number generator is crucial, since an inadequate random number generator can produce incorrect results. New applications, or more powerful computers that can produce longer sequences, may also show up flaws in generators that were previously thought to be first rate [4, 5]. So providing an adequate random number generator for Java Grande applications is a challenging problem.

2 Random Number Generators in Java

Java has a number of advantages for implementing random number generators. Most algorithms use integer arithmetic requiring specific precision, either 32 or 64 bits. Java implementations of these algorithms are portable since `int` and `long` integer types have well-defined precision, unlike the corresponding types in C. Some algorithms can be implemented more efficiently if the handling of overflow in integer arithmetic is well defined, as it is in Java. Random number generators require initialization, which needs to be done before they are called. In procedural languages it is up to the programmer to remember to call the initialization routine, however in an object-oriented language like Java, the initialization can be incorporated into the constructor of the generator class.

Another advantage of Java is that it is designed to use standard class libraries with well-defined APIs, and a huge variety of such libraries are available, including random number

generators. Fortran 90 and ANSI C both offer standard interfaces to random number generators in the form of intrinsic functions or standard libraries, however the interfaces are somewhat restrictive, in terms of both the functionality and the algorithms available. Java offers the potential for providing a full-featured, high-quality, portable class library for random number generation.

There are currently three different classes providing access to random number generators in Java [19]:

- `java.security.SecureRandom` provides an interface to cryptographically strong random number generators, which are special-purpose generators that use hardware devices and/or non-linear algorithms to produce sequences that are not predictable (unlike the simple linear generators commonly used in numerical libraries). Cryptographic generators are not used in scientific simulations requiring large quantities of random numbers, since they are much too slow, or in the case of hardware devices, are not reproducible.
- `java.util.Random` provides quite a well-structured random number generator interface. There is a fundamental method called `next` which returns up to 32 random bits, and the other methods use this to produce random numbers of different types – `nextBoolean`, `nextInt`, `nextLong`, `nextFloat` and `nextDouble`.
- `java.Math` provides a `Random` method, which provides a simpler interface to the `nextDouble` method of `java.util.Random`, for applications that just need to access a stream of pseudo-random floating point numbers.

These APIs all have shortcomings that make them inadequate for supporting Grande scientific applications. The generators used in the `SecureRandom` class are targeted at cryptography applications, which have different requirements than scientific simulations, and consequently this class has an API that is unsuitable for a Java SSL. The other APIs are aimed at general applications, but do not have some special features or provide the quality and choice of algorithms required for scientific applications such as Monte Carlo simulation. The pros and cons of the existing APIs are discussed in more detail in the next section, which outlines some requirements for a random number generator API for a scientific software library to be used by Java Grande applications.

3 API Requirements for Java Grande Applications

There has been considerable work recently by the Java Grande Forum on developing proposed APIs and reference implementations for object-oriented numerical libraries within Java [10, 2], to provide what is effectively a Java scientific software library.

The random number generator API used in `java.util.Random` is adequate for most purposes, and provides a reasonable basis for a more comprehensive API that could be used within a Java SSL. We have therefore used it as the basis for our initial proposal, although alternate approaches are possible and may prove to be more practical as more work is done on requirements and implementation details.

3.1 A choice of algorithms

The main problem with random number generators is that they do not produce truly random sequences, so no generator can be guaranteed to work adequately for a given application. It is therefore essential, particularly for Grande applications which tend to be more sensitive to the quality of the generator, that a choice of algorithms is available, so users can run their application using at least two different generators, and check that the results are the same within statistical errors.

The documentation for the `java.util.Random` class points out that the API is structured so that a programmer can implement an alternate random number generator algorithm by creating a sub-class that overrides the basic `next` method, but inherits all the others. However no other algorithms are provided. A Java SSL should provide a choice of algorithms for the random number generator, and the API should be general enough to support any generator algorithm. Ideally, it should also allow different generators to be called without having to modify the application code.

The `SecureRandom` class in the `java.security` package provides this capability, allowing the user to select from a set of standard algorithms [19]. The recommended mechanism for doing this is to provide a static method `getInstance` which can be used to obtain an instance of a random number generator class. The Java classname for the particular algorithm can be optionally supplied to the method to specify which algorithm should be used. The method will attempt to instantiate a class with the name supplied and throw an exception if it is not found. If no parameter is supplied the default generator is returned. This mechanism has been incorporated into our proposed API.

As with `java.util.Random`, users can also provide their own random number generator by simply sub-classing the Java SSL random number generator class, and providing an implementation of the methods that are algorithm-dependent, such as `next`. All the other methods can be inherited. In some cases it may be more efficient to override other methods, such as `nextFloat` or `nextDouble`.

3.2 An adequate period

Random number generators are different to standard mathematical functions in that they must maintain a state between calls to the generator, since they use iterative functions. The state of the generator is finite, so the sequence of numbers produced by the generator must repeat after a certain period. Generators provided as part of a Java SSL should have periods that are much larger than the amount of pseudo-random numbers that might be produced in a Java Grande application. How large is this likely to be?

The 32-bit linear congruential generators commonly used in the past have a period of 2^{32} , which corresponds to roughly a Megaflop-hour of operations. This is a trivial amount of computation for today's desktop machines, so these generators are no longer adequate for scientific applications. The period of the default generator used in `java.util.Random` is 2^{48} , corresponding to a few Gigaflop-days, which is commonly exceeded by current Monte Carlo simulations in computational physics, and within 20 years will seem as trivial as a Megaflop-hour seems today. 64-bit or combined 32-bit LCGs have a period of around 2^{64} , corresponding to about a Teraflop-year, which is a huge amount of computation, but within the grasp of grand challenge applications on current supercomputers, and in 20 years will be commonplace for large-scale scientific applications.

However, it is not difficult to implement generators with periods that are so large that they will be adequate for any Java Grande application in the foreseeable future. Lagged Fibonacci generators can have arbitrarily long periods, depending on the size of the lag that is used. Combining two 64-bit LCGs, or four 32-bit LCGs, gives a period of around 2^{128} , which should be adequate since it corresponds to roughly a Petaflop-age-of-the-universe computation!

3.3 A better default algorithm

The API for `java.util.Random` mandates the use of a specific algorithm for the `next` method, so that it will give the same results across any Java Virtual Machine (JVM). The algorithm chosen as the default is the 48-bit LCG used in the C library function `drand48`. This is quite a good algorithm, however it has some significant problems that make it inadequate for use as the default generator for a Java SSL. The first is that rather than using a prime modulus, which gives better quality random numbers, it uses a modulus that is a power of 2, which is faster to implement. A more serious problem is that the period of this generator is too small for many Java Grande applications.

A Java SSL should provide a higher quality, longer period default random number generator. But should we mandate a particular algorithm for the sake of portability, as is done in `java.util.Random`, or opt for flexibility and allow any algorithm to be used, as is done in `SecureRandom`? If we do not mandate a default algorithm, then we sacrifice portability, and some implementations of the Java SSL might use a substandard generator. However if we do mandate an algorithm, it may prove to be inferior as new algorithms, applications and statistical tests are developed. A better option would be to mandate a specific default algorithm for a particular version of the Java SSL, but allow the possibility that the default will change in future versions.

The Java SSL should provide a suite of different algorithms covering a range of speed and quality trade-offs to suit different applications. The default algorithm should provide a general-purpose happy medium, with good performance but high quality. Good candidate algorithms include a multiplicative lagged Fibonacci generator with a long lag, or a combined generator using either two 64-bit or four 32-bit LCGs to provide a large enough period.

3.4 Initializing and checkpointing the generator

In `java.util.Random`, initializing (or seeding) the generator is done by passing a single `long` integer (the seed) to the constructor, or to a `setSeed` method. If no seed is provided, the generator is initialized using a value taken from the clock time. This is fine for initializing the state of the `drand48` generator, which is just a 48-bit integer value, but other algorithms such as combined LCGs have multiple integers for their state variables, while LFGs require the initialization of arrays of hundreds or thousands of integer or floating point values used to keep their state.

It is unwise to leave the initialization of the state variables of these generators up to the user, since in some cases (particularly LFGs) this is a subtle process, and a naive initialization may result in a correlated (non-random) sequence. A better approach is to follow the existing API, with the user providing just a single `long` integer, and require the generator implementation to provide a sound mechanism for initializing its state based on

that single seed. This approach also has the advantage that the interface to `setSeed` is independent of the generator used.

Large-scale scientific applications, particularly Monte Carlo simulations, often require multiple runs of the program to produce a final result. To do this, the user will checkpoint the state of the simulation so that the run can be restarted later on. This includes checkpointing the state of the random number generator, by writing it to a file. The API should therefore provide a `getState` method that returns the current state of the generator. This should return a general `Object`, since different algorithms have different state variables. A corresponding `setState` method would allow such an object to be used to initialize the generator after checkpointing. These are algorithm-dependent methods that must be implemented for each generator sub-class.

Since in most cases the user will want to checkpoint the generator's state directly to a file, it is helpful to also provide a `writeState` method that writes the state to a specified file after obtaining the object using `getState`, and a `readState` method that reads in the state from a specified file and returns a general `Object` that can be used by the `setState` method. These methods can be implemented in the base class in a generator-independent way. Java's object serialization mechanism can be used to provide portable data files.

3.5 Generating arrays of random numbers

Since generating a random number only takes a few floating point operations, the overhead of the method call can be relatively high. If the application requires an array of random numbers, it may be more efficient to call a method that fills up the whole array at once, rather than making multiple method calls. Some numerical libraries provide routines to support this.

Advanced compilers should be able to inline the method calls to avoid this overhead, which would appear to obviate the need for providing such a method. However Fortran 90 provides such an interface not just to avoid the overhead of the function call, but primarily to allow for the possibility that the generation of the array of random numbers may be vectorized or parallelized, so a method of this kind may still be useful to support data parallel implementations of Java, such as proposed by the HPJava Project [7].

Within Java, we can offer a simple interface to such a method by overloading the method call for generating a single random value if it is made with an array as an argument. So a call to `nextInt()` will return a single random integer, whereas a call to `nextInt(int rand[])` will fill the array `rand[]` with random integers.

3.6 Concurrency and synchronization

The reference implementation for `java.util.Random` suggests that the `next` method be synchronized, so that it is thread-safe. Unfortunately, synchronization in Java can have a substantial performance overhead, which may reduce the speed by up to a factor of 3. It would be possible to run different instances of the random number generator in different threads, thus avoiding the synchronization overhead. However we need to avoid having overlapping or correlated pseudo-random sequences in different threads.

The problem is similar to implementing a random number generator on a parallel computer. To avoid all the processors accessing a single random number stream from one process, which has a costly communications overhead, parallel random number generators

are designed to provide different random number streams for each processor (see ref. [5] for a review of parallel random number generators). A similar approach could be adopted for the random number generator in a Java SSL, to handle concurrency from using multiple threads or a parallel Java program (using Java with MPI [3], for example). For each concurrent process, the programmer could instantiate a new random number generator, which would synchronize only in the initialization procedure. After that, all calls to the generator could be unsynchronized.

In some cases the programmer may not wish to explicitly manage the creation of a new instance of the unsynchronized random number generator for each thread in the program. The API could provide synchronized generators as default, but allow the user to call unsynchronized versions if required.

3.7 Some other issues

We have not addressed alternate probability distributions in our reference implementation. For random number generators in SSLs, the default is always to provide uniformly distributed random numbers, however some applications require other distributions, such as Gaussian, or Poisson. A `nextGaussian` method is provided by `java.util.Random`, and it would be straightforward to also provide a `nextPoisson` method, as well as other distributions. It may be possible to make this more general, by enabling the user to specify or implement the required probability distribution.

The `java.util.Random` API provides a useful method for returning an integer between 0 and `n`, by overloading `nextInt()` to allow `nextInt(int n)`. Curiously, the documentation does not list a corresponding `nextLong(long n)` method, which should be added.

3.8 The proposed API

It includes the methods available in `java.util.Random`, as well as the additions outlined in this section (which are marked with an asterisk). This should be viewed as a first pass at an API for a Java SSL random number generator, which will hopefully be subject to much discussion and improvement. A more detailed version of the API (using javadoc) is available on the Web [6].

```
public class Random extends java.util.Random {

    Random();
    Random(long seed);

    * public static Random getInstance(String type) throws RandomException;
    * public static Random getInstance();

    public void setSeed(long seed);
    * public Object getState();
    * public void setState(Object seeds);
    * public Object readState(String filename);
    * public void writeState(String filename);
```

```

protected int next(int bits);
public void nextBytes(byte[] bytes);
public boolean nextBoolean();
public float nextFloat();
public double nextDouble();
public int nextInt();
public int nextInt(int n);
public long nextLong();
* public long nextLong(long n);

* public void nextInt(int[] random_ints);
* public void nextLong(long[] random longs);
* public void nextFloat(float[] random_floats);
* public void nextDouble(double[] random_doubles);

public double nextGaussian();

}

```

4 Implementation and Performance Issues

We have developed a reference implementation of the proposed random number generator API outlined above, which is available on the Web [6]. We have implemented five different algorithms, all of which are believed to be excellent generators for scientific applications.

- **LCG64** – a 64-bit LCG with a prime modulus, $L(2307085864, 0, 2^{63} - 25)$, recommended by L’Ecuyer [14].
- **Ranmar** – a variation of Marsaglia’s commonly-used RANMAR generator combining a simple Weyl generator with an additive LFG [17, 8], however we have increased the lags and used $F(4423, 1393, +)$ to improve the quality.
- **MultLFG** – a multiplicative lagged Fibonacci generator, $F(1279, 418, *)$.
- **Ranecu** – the popular combined 32-bit LCG of L’Ecuyer [13, 8].
- **CLCG4** – a combination of four (rather than two) 32-bit LCGs, to give a longer period [15].

Table 1 shows timings for the random number generator implementations in both C and Java (JDK 1.2) on two different platforms — a 300 MHz Sun UltraSPARC under Solaris and a 300 MHz Pentium II under NT. The algorithms we have implemented are similar in performance to the default generator for `java.util.Random`, but are of higher quality.

The synchronization overhead is only about 50% for Solaris on the SPARC, but as much as a factor of 3 on Pentium under NT. However, the performance of the synchronized methods is about the same for each JDK, since NT on the Pentium gives correspondingly better performance than Solaris for SPARC on the basic unsynchronized methods.

Table 1: Results of timings of Java (JDK 1.2) and C implementations of different random number generators, given as millions of pseudo-random numbers produced per second (so higher values are better). Java results are given for both synchronized and unsynchronized method calls. The Sun results are for a 300 MHz Sun UltraSPARC under Solaris and the NT results are for a 300 MHz Pentium II under Windows NT. `drand48` refers to the implementation in the C library and in `java.util.Random`. Some algorithms have no C timings since they require 64-bit longs.

		drand48	LCG64	Ranmar	MultLFG	Ranecu	CLCG4
Sun C	float	0.75	—	3.46	—	1.32	0.62
Sun Java unsynch	int	—	0.69	0.98	2.53	0.56	0.20
	float	—	1.11	2.44	2.09	1.26	0.41
	double	—	1.09	0.63	1.79	0.55	0.21
Sun Java synch	int	2.03	0.56	0.70	1.63	0.44	0.18
	float	1.85	0.89	1.50	1.20	0.97	0.38
	double	0.86	0.84	0.49	1.30	0.49	0.20
NT Java unsynch	int	—	1.81	1.20	6.14	1.27	0.48
	float	—	3.04	3.88	5.08	4.11	1.14
	double	—	3.04	0.78	5.24	1.51	0.48
NT Java synch	int	7.97	1.11	0.83	1.93	0.69	0.36
	float	6.17	1.44	1.70	1.88	1.80	0.81
	double	3.17	1.49	0.62	1.89	0.97	0.42

The use of improved just-in-time (JIT) compilers in JDK 1.2 has greatly improved the performance over JDK 1.1. Results obtained using older compilers (JDK 1.1.3) showed the Java implementations to be around 10 times slower than C code, however the latest JDK gives results that are only about 50% slower than C. The `drand48` routine has somehow been optimized so that it is actually faster in Java than in C.

Many of the routines in scientific software libraries, such as linear algebra solvers, involve substantial amounts of Fortran code, so in developing scientific software libraries for Java, it is much easier to just provide a Java interface to these existing routines and to call them as native methods [2]. This usually provides better performance than a pure Java implementation, however it sacrifices the portability of the implementation, which is one of the advantages of using Java. Fortunately, random number generator algorithms only require a small amount of coding, and are easy to convert to pure Java. It would also be easy to provide an implementation using Java interfaces to native methods, however since the pure Java version is not much slower than C, the additional overhead of a native method call may outweigh the small performance advantage of the native code.

5 Conclusions

Java currently provides a random number generator in `java.util.Random` that is adequate for many applications, although both the interface and the implementation lack many of the qualities required for the type of large-scale scientific applications being addressed by the Java Grande Forum.

Some of the interface issues include mechanisms for checkpointing the state of the random number generator; generating arrays of random numbers; handling concurrency; and providing a convenient way to choose between a variety of different algorithms. We have presented a proposal for an improved API, and provided a reference implementation, for a random number generator package that could be used as part of a Java scientific software library.

Implementation issues include choosing a good general-purpose default algorithm that provides an adequate period and a reasonable trade-off between speed and quality. The default algorithm in `java.util.Random` does not have a large enough period of repetition for some scientific applications such as large-scale Monte Carlo simulations, and it is unfortunate that a 48-bit linear congruential generator was selected as the default, rather than a superior 64-bit generator. For a random number generator within a Java SSL, a better default algorithm would be required. Good candidate algorithms include a multiplicative lagged Fibonacci generator or a combined linear congruential generator.

Both the proposed interface and the reference implementation are still in a preliminary stage, and further user input, implementation tests, experimentation and discussion will be required to improve them. However, we have highlighted a number of the issues involved, and suggested some possible solutions.

Acknowledgements

This work was carried out under the Distributed High-Performance Computing Infrastructure (DHPC-I) project of the Research Data Networks (RDN) and Advanced Computational Systems (ACSys) Cooperative Research Centers (CRC). RDN and ACSys are established under the Australian Government's CRC Program.

References

- [1] K. Binder ed., *Monte Carlo Methods in Statistical Physics*, Springer-Verlag, Berlin, 1986.
- [2] R.F. Boisert *et al.*, Developing numerical libraries in Java, *Proc. of ACM Workshop in Java for High-Performance Network Computing*, Stanford, February 1998, <http://www.cs.ucsb.edu/conferences/java98/program.html>.
- [3] Bryan Carpenter *et al.*, MPI for Java - Position Document and Draft API Specification, Java Grande Forum Technical Report JGF-TR-03, November 1998, <http://www.npac.syr.edu/projects/pcrc/reports/MPIposition/position.ps>.
- [4] P.D. Coddington, Analysis of Random Number Generators Using Monte Carlo Simulation, *Int. J. Mod. Phys. C* **5**, 547 (1994).

- [5] Paul D. Coddington, Random Number Generators for Parallel Computers, *The NHSE Review*, <http://nhse.cs.rice.edu/NHSEreview/>, 1996 Volume, Second Issue.
- [6] P.D. Coddington, J.A. Mathew and K.A. Hawick, Random number generators for Java Grande, <http://acsys.adelaide.edu.au/projects/javagrande/random/>.
- [7] HPJava Project, <http://www.npac.syr.edu/projects/pcrc/HPJava/>.
- [8] F. James, A review of pseudorandom number generators, *Comp. Phys. Comm.* **60**, 329 (1990).
- [9] The Java Grande Forum, <http://www.javagrande.org/>.
- [10] Java Grande Forum, Making Java Work for High-End Computing, Java Grande Forum technical report JGF-TR-1, <http://www.javagrande.org/reports.htm>.
- [11] D.E. Knuth, *The Art of Computer Programming Vol. 2: Seminumerical Methods*, Addison-Wesley, Reading, Mass., 1981.
- [12] P. L'Ecuyer, Random numbers for simulation, *Comm. ACM* **33:10**, 85 (1990).
- [13] P. L'Ecuyer, Efficient and portable combined random number generators, *Comm. ACM* **31:6**, 742 (1988).
- [14] P. L'Ecuyer, F. Blouin, and R. Couture, A Search for Good Multiple Recursive Generators, *ACM Trans. on Modeling and Computer Simulation* **3**, 87 (1993).
- [15] P. L'Ecuyer and T.H. Andres, A Random Number Generator Based on the Combination of Four LCGs, *Mathematics and Computers in Simulation* **44**, 99 (1997).
- [16] G.A. Marsaglia, A current view of random number generators, in *Computational Science and Statistics: The Interface*, ed. L. Balliard, Elsevier, Amsterdam, 1985.
- [17] G.A. Marsaglia, A. Zaman and W.-W. Tsang, Toward a universal random number generator, *Stat. Prob. Lett.* **9**, 35 (1990).
- [18] S.K. Park and K.W. Miller, Random number generators: Good ones are hard to find, *Comm. ACM* **31:10**, 1192 (1988).
- [19] Sun Microsystems Inc., Java Platform 1.2 API Specification, <http://java.sun.com/products/jdk/1.2/docs/api/index.html>.