JAPARA – A Java Parallel Random Number Generator Library for High-Performance Computing

P.D. Coddington and A.J. Newell
School of Computer Science, University of Adelaide
Adelaide, SA 5005, Australia
paulc@cs.adelaide.edu.au

Abstract

Random number generators are one of the most common numerical library functions used in scientific applications. The standard random number generator provided within Java is fine for most purposes, however it does not adequately meet the needs of large-scale scientific applications, such as Monte Carlo simulations. Previous work has addressed some of these problems by extending the standard Random API in Java and providing an implementation that includes a choice of several different generator algorithms. One issue that was not addressed in this work was concurrency. Implementations of the standard Java random number generator use synchronized methods to support the use of the generator across multiple Java threads, however this is a sequential bottleneck for parallel applications. Here we present a proposal for further extending the standard API to support parallel generation of random number streams, which we have implemented in JAPARA, a Java Parallel Random Number Generator Library for high-performance computing.

1. Introduction

Java has the potential to be an excellent language for developing large-scale science and engineering applications [5]. Random number generators are commonly used in these types of "Java Grande" applications, so it is important to provide access to an efficiently implemented, high-quality random number generator library through a standardized Java application programming interface (API).

Existing random number generators available in the standard Java libraries are inadequate for use in some applications such as large-scale Monte Carlo simulations. Previous work by Coddington *et al.* [2] addressed many of the deficiencies of the standard `java.util.Random` class [14], by extending this class to provide:

- a better default algorithm with a longer period;
- a choice of high-quality, long-period generator algorithms;
- methods for checkpointing and restarting the generator;
- methods for generating arrays of random numbers, to avoid method call overhead.

However one issue that was not addressed in this work was support for concurrency. Scientific applications that make heavy use of random number generators, such as Monte Carlo simulation, are often implemented so as to take advantage of parallel processing, since they typically require large amounts of computation and are generally straightforward to parallelize.

Java is well-suited to the development of parallel applications due to its inbuilt support for threads. However the reference implementation of the standard Java random number generator `java.util.Random` suggests that the method for random number generation be synchronized, so it is thread-safe. For parallel programs such as a parallel Monte Carlo simulation, having a random number generator that is synchronized is a sequential bottleneck that could greatly limit the speedup and scalability of the parallel program. Even for sequential programs, synchronized methods in Java can have a significant performance overhead [2].

In this work, we have adapted the random number generator library of Coddington *et al.* [2] into a library called JAPARA (JAva PArallel RAndom number generator library), which provides a selection of random number generator algorithms that allow the independent generation of random number streams on different threads, so there is no need for synchronized methods for generating the random numbers. JAPARA was designed to be easy to use, with a familiar interface to Java programmers. It is based on the previous sequential library developed by Coddington *et* *al.* [2], which provided some simple extensions of the standard `java.util.Random` API. JAPARA uses the minimal additional extensions to this API that are required to enable support for parallel random number generation, while still providing the same interface and performance for sequential applications.

2. Random Number Generators for Java

There are three different classes providing access to random number generators in the standard Java libraries [14]. They all have shortcomings that make them inadequate for supporting large-scale scientific applications.

The `java.security.SecureRandom` class 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 reproduceable. The `SecureRandom` class is targeted at cryptography applications, which have different requirements than scientific simulations, and consequently this class has an API that is unsuitable for this purpose.

The `java.util.Random` class provides 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`. There is also a `nextGaussian` method to produce random numbers with a Gaussian (or normal) probability distribution, rather than a uniform distribution.

Finally, `java.Math` provides a `random` method, which is just a simpler interface to the `nextDouble` method of `java.util.Random`.

The `java.util.Random` class provides a good basis for a random number generator for large-scale scientific applications, however it has several limitations for these types of applications:

1. The main problem is that there is no choice of different generator algorithms. Because random number generator algorithms are deterministic (i.e. they produce pseudo-random rather than truly random numbers), no generator is perfect, so there is always the possibility that it will adversely affect results for a particular application. For this reason simulations should always be done using at least two different generators to check that the choice of generator does not affect the results.

2. The generator algorithm used is quite good, and the period (the number of random numbers that can be generated before the sequence repeats itself) is adequate for current hardware, however it would not adequate for the kinds of large-scale simulations that are currently possible on large supercomputers, and will be common even on commodity systems in 10-20 years time. A default generator with larger period would be better for large-scale simulations.

3. Scientific applications such as Monte Carlo simulations typically require such long execution times that it is essential to checkpoint the state of the simulation, which includes the state of the random number generator. How the state of a generator is stored is different for different generators – it may be anything from a single integer value to a large array of floating point numbers. Methods to save and restore the state of the generator to an object or a file are useful to hide details of the generator's state.

4. Many large-scale scientific applications require generating large arrays of random numbers. Since generating a random number is typically very fast (of the order of a microsecond), method call overhead can be significant. It may be more efficient to provide a method that fills a specified array with random numbers, rather than using a loop that calls a standard method to generate a single random number.

5. Standard implementations of the API use a synchronized method for generating the random numbers, so that it is thread-safe. For parallel applications, having a random number generator that is synchronized is a sequential bottleneck that could greatly limit the speedup and scalability of the parallel program.

JAPARA addresses the final point concerning concurrency, which will be discussed in Section 4 of this paper. JAPARA is an extension of previous work by Coddington *et al.* [2], which addressed the first four points in the above list, by making minimal extensions to the standard `java.util.Random` class. This class is designed so that a programmer can utilize a different generator algorithm by extending the class and just overriding the constructor and the `next()` method, which is used by all the other methods for generating different types of random numbers. All these other methods can just be inherited, or overridden if it would be more efficient to do so.

Rather than leaving it to the programmer to extend the standard `java.util.Random` class with their own generator implementation, Coddington *et al.* provided a library consisting of several classes implementing different generator algorithms. The programmer can then select from any of these classes. The recommended mechanism for doing this, which is used by the `SecureRandom` class in the `java.security` package [14], is to provide a static method `getInstance` which can be used to obtain an instance of a specified random number generator class. The Java classname for the particular algorithm can be optionally supplied to the `getInstance` method to specify which algorithm should be used. The method will attempt to instantiate an object of that class, or throw an exception if it is not found. If no parameter is supplied a default generator is returned. This mechanism allows different generators to be called without having to modify the application program, by just specifying the generator name as an input string to the program. It is also trivially extensible to provide implementations of new generators.

The extensions to the standard `java.util.Random` API that are used by JAPARA to address the limitations listed above are shown in Figure 1.

Several other random number generator libraries for Java have been developed, mainly to provide Java implementations of different random number generator algorithms. These include RngPack [3] and randomX [15], both of which provide a good choice of high-quality, long-period generators. However these libraries are not designed to extend the standard `java.util.Random` library, which we believe is a positive feature of our work. Also they do not address all of the other problems listed above, in particular the issue of concurrency.

3. Parallel Random Number Generators

In principle it is straightforward to develop a parallel random number generator in Java by just running different instances of a standard sequential random number generator in different threads, thus avoiding the synchronization overhead from running a single instance of the generator that is accessed from all threads. However this is not as simple as it might seem, since we need to avoid having overlapping or correlated sequences on different threads.

For all of the commonly used random number generator algorithms, there are various techniques for ensuring that sequences can be generated independently (i.e. with no synchronization required) on different threads and that there is no overlap (and minimal correlation) of these sequences [1]. These techniques require some synchronization (or communication) during the initialization of the generator in the constructor, but after that, calls to the methods for generating a new random number can be unsynchronized, so there is no overhead for concurrent execution.

There are four main techniques used for parallel random number generation:

- **Leapfrog** – The sequence of random numbers that would be produced by the sequential random number generator algorithm is partitioned among the processors in a cyclic fashion, like a deck of cards dealt to card players, so that process (or thread) p of an application with N processes (or threads), which would typically be run in an N processor parallel machine, generates the sub-sequence $X_p, X_{p+N}, X_{p+2N}, \ldots$

- **Sequence Splitting** or **Boosting** – The sequence is partitioned among processes (or threads) in a block fashion, by splitting it into non-overlapping contiguous sections, with the size of each section being much larger than would be required for any simulation. This can be done for generator algorithms that allow efficient *boosting* to an arbitrary element in the sequence.

- **Independent Sequences** - The initial seeds are chosen in such a way as to produce a long period independent sequence on each processor (this is only feasible for certain generators such as lagged Fibonacci generators).

- **Parameterization of the Generator** - The generator has a different parameterization on each processor, so there is effectively a different generator algorithm running on each processor.

Most of these approaches will only work for certain generator algorithms, however at least one of these approaches will work for all of the commonly used algorithms. Several freely available parallel random number generator libraries have been developed that use one or more of these techniques, mostly for use with message-passing MPI programs, for example PRNGlib [13] and Scalable Library for Pseudorandom Number Generation (SPRNG) [12].

To our knowledge there has been no implementation of a parallel random number generator library in Java. L'Ecuyer *et al.* have developed a Streams and Substream package [11] for for C++ and Java that could in principle be used for parallel random number generation, however it only uses a single generator algorithm. Our goal was to develop a parallel library that conformed as much as possible to the standard `java.util.Random` API, with the improvements introduced by Coddington *et al.* [2]. In particular, we wanted to provide a choice of generator algorithms, which is very important for large-scale scientific applications.

4. Design

The programming model that JAPARA is aiming to support is that a user instantiates a separate random number generator (RNG) object for each thread. These RNG objects can then generate their own sequences of random numbers completely independently, so that no synchronization is required.

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

      Random();
      Random(long seed);

      // Allow a choice of generator algorithm
   *  public static Random getInstance(String type) throws RandomException;
   *  public static Random getInstance();

      // Initialize a new independent generator
  **  public static void setSequenceSeed();

      public void setSeed(long seed);

      // Enable checkpointing of generator state
   *  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();
      // Added for completeness
   *  public long nextLong(long n);

      // Generate an array of random numbers
   *  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();

   }
```

Figure 1. The API for the JAPARA random number generator library to support large-scale scientific (Java Grande) applications. Extensions to the standard java.util.Random API proposed in previous work by Coddington et al. are marked with asterisks. Additional extensions described in this paper to support parallel execution are marked with a double asterisk. Comment lines indicate the purpose of the extensions.

The main requirements in designing and implementing JAPARA were to ensure that it provides independent non-overlapping sequences for each instance of the generator, while conforming as closely as possible to the sequential API of `java.util.Random`, and its extension by Coddington *et al.* described in the previous section. In particular, JAPARA should be able to be used for sequential as well as parallel programs, in which case the interface should be the same as that defined by Coddington *et al.*, i.e. a simple extension to `java.util.Random`.

Perhaps the simplest approach to constructing the parallel random number generator would be to set the seed once when the first generator object is instantiated (for example in the main program or a master thread), and then all subsequent generator objects that are instantiated in other threads would be automatically seeded using the techniques described in Section 3 to ensure that the sequences for each generator instance do not overlap. This is the approach used in L'Ecuyer's Streams and Substream package [11].

However it is difficult to see how to implement this approach in a simple way and still conform to the semantics of the constructor for the standard `java.util.Random` API. In the standard API, the programmer can specify a particular seed value to initialize (or seed) the generator in the constructor, e.g.

```
Random rand = new Random(mySeed);
```

or alternatively, seed the generator after the new Random object has been instantiated, by using the `setSeed` method, e.g.

```
Random rand = new Random();
rand.setSeed(mySeed);
```

We could design the parallel API so that the first Random object is instantiated and seeded in the usual way (as above), and all subsequent Random objects are automatically given an appropriate seed using the techniques for creating independent random sequences. For example, we could perhaps do this by ignoring any seed specified in the constructor and replacing it by the automatically generated seeds, although this would obviously violate the syntax of the standard API, and it may be that the programmer really does want to specify the seeds for all the generators. Alternatively we could perhaps indicate that automatic seeding should be used by not specifying a seed in the constructor, e.g.

```
Random rand = new Random();
```

However this would also violate the syntax of the standard API, for which an empty constructor means that the seed should be set from the current value of the clock.

One possibility would be to overload the constructor by passing it a dummy value of a particular type, such as a string. However this is not very elegant, and if we are going to extend the standard API by adding something like this, we may as well add something that makes it clear what is actually happening, i.e. a method that automatically generates the seeds for a new independent non-overlapping sequence of random numbers. In the current implementation of JAPARA this method is called `setSequenceSeed`, as shown in Figure 1. Another possibility that we have recently considered would be to overload the `setSeed` method so that if it is called with no parameters (which is not allowed in the standard sequential API) then the seeds are automatically generated. This is perhaps a more elegant way of indicating what is actually happening, i.e. the seeds are being set automatically with no input from the user.

In order to automatically seed the generator for each new instance, for example by using boosting or the independent sequence method, we use the concept of two different kinds of seeds:

- A *class* seed (i.e. a static variable in Java) – this is set the first time that the user creates any RNG object, and changed (e.g. boosted) to ensure non-overlapping sequences every time `setSequenceSeed` is called.

- An *instance* seed (i.e. an instance variable that is different for every RNG object) – by default this is set based on the seed specified in the constructor or `setSeed`, but is set to be the class seed every time `setSequenceSeed` is called.

Note that these seeds store the initial state of the generator for each sequence, so in general they will not be just a single integer value like the seed that is used to initialise the generator in the Random constructor or the `setSeed` method.

5. Implementation and Algorithms

Currently we have developed parallel versions of two of the five different generator algorithms implemented by Coddington *et al.* [2], and are working on parallelizing the remaining algorithms. We have also implemented a parallel version of a new combined multiple recursive generator. The generators currently availble in JAPARA are:

- `LCG64` – a 64-bit Linear Congruential Generator (LCG) with a prime modulus, recommended by L'Ecuyer [10].

- `MultLFG` – a multiplicative Lagged Fibonacci Generator (LFG) with large lags [2].

- `CMRG` – the combined multiple recursive generator developed by L'Ecuyer [8, 7] and used in his Streams and Substreams package [11].

Generator	Initialization Time (secs)
Parallel CMRG	0.041
Parallel LCG64	0.097
Parallel MultLFG	0.347

Table 1. Time required to initialize 4 independent sequences in 4 threads for different generators in JAPARA.

These generators are all parallelized in the standard way using either boosting (for the LCG and CMRG) or independent sequences (for the LFG).

Since JAPARA provides independent generator instances for each thread, the execution time for each instance of the random number generator will be the same as measured for the sequential implementation [2], as long as there is one thread per processor.

The only performance overhead in the parallel version is in the initialization of each instance of the generator. Currently `setSequenceSeed` is implemented as a synchronized method, to ensure that there are no race conditions in initializing the generators. This means that the initialization of the generators is done sequentially, which could potentially be a bottleneck when there are a large number of threads, particularly for the lagged Fibonacci generator which needs to initialize an array with the order of a thousand elements.

Some preliminary measurements of the initialization time are shown in Table 1, measured on a Sun server containing four 296 MHz UltraSPARC-II processors. Since sequence creation is sequential, the times increase roughly linearly with the number of threads, and are independent of the number of processors used. Although the initialization times are significantly larger than the generation times (which are of the order of a microsecond on similar hardware [2]), they are still quite small, however they could potentially become significant (particularly for the lagged Fibonacci generator) for very large numbers of threads.

We are investigating the possibility of initializing the sequences concurrently. In principle, setting the seeds using the methods outlined in Section 3 could be done independelty (and therefore in parallel) as long as the sequence number is known. So we expect that `setSequenceSeed` does not have to be synchronized, although it will need to call a synchronized method to update and return a static variable storing the number of sequences. However this synchronized method should be very fast and allow for significant concurrency in the initialization of the sequences.

6. Conclusions and Future Work

Java provides a random number generator in `java.util.Random` that is adequate for most applications, however both the interface and the implementation lack many of the qualities required for some large-scale scientific applications. In particular it lacks support for efficient concurrent execution.

We have designed, implemented and tested a Java random number generator library called JAPARA, which extends the standard Java API to provide additional functionality for supporting large-scale scientific applications on high-performance computers, including a choice of a number of different high-quality random number generator algorithms.

Each of the generators is implemented so that synchronization is only required for the initialization of the generators, but not for the generation of random numbers, thus enabling efficient concurrent generation of independent random number streams in multiple Java threads. JAPARA can also be used for sequential Java programs, providing the same interface, functionality and performance as our previous sequential version of this library.

JAPARA is still under development. We are currently working on implementing parallel versions of the other commonly-used random number generator algorithms that were implemented in the sequential version of this library, as well as some new algorithms.

We are plannning to modify the implementation of the generators so that the initialization of new sequences can be done in parallel. Rather than having a synchronized method to generate the new sequence, which leads to sequential initialization, the only synchronized method will be assigning a unique sequence number to each generator instance. For all the current algorithms, the independent sequences can be generated in parallel using only this value and the initial seed for the generator.

Making this change to the design of JAPARA will also make it easier to support other models of concurrent programming. Although the JAPARA library was designed with the goal of supporting parallelism using Java threads, it could very easily be modified to work with a message passing or distributed computing programming paradigm. The only feature that would need to be changed is the mechanism for assigning a unique sequence number to each generator instance, which can be done using a synchronized method in JAPARA, but could be trivially implemented in any parallel programming model.

We are also working on more rigorous tests of the generators, both sequential (individual streams for each generator instance) and parallel (correlations between streams in different instances of the generator). All of the sequential generator algorithms used in JAPARA have been subjected to a battery of tests and are known to be of high quality, so the only real issue for sequential testing is to ensure that the algorithms have been correctly implemented in Java. So far we have just used two simple tests to provide a "sanity check" of the Java implementation. Firstly, the generators were checked to see if the sequence of numbers produced by `nextDouble()`, which must be in the range $[0, 1)$, have an average of a 0.5 within statistical errors. The second test test checks for a uniform distribution of the random numbers by generating histograms of the values of `nextDouble()` over small intervals. We plan to apply some more stringent tests to all the generator algorithms before the JAPARA code is publicly released. We are also working on methods for testing the independence of the parallel random number generator streams. There has been surprisingly little research in this area.

References

[1] Paul D. Coddington. Random Number Generators for Parallel Computers. *The NHSE Review*, http://nhse.cs.rice.edu/NHSEreview/, 1996 Volume, Second Issue.

[2] P.D. Coddington, J.A. Mathew and K.A. Hawick. Interfaces and Implementations of Random Number Generators for Java Grande Applications. Proc. of High Performance Computing and Networks (HPCN) Europe '99, Amsterdam, April 1999.

[3] Paul Houle. RngPack. http://www.honeylocust.com/RngPack/.

[4] F. James. A review of pseudorandom number generators. *Comp. Phys. Comm.* **60**, 329 (1990).

[5] Java Grande Forum. Making Java Work for High-End Computing. Java Grande Forum technical report JGF-TR-1, http://www.javagrande.org/reports.htm.

[6] P. L'Ecuyer. Efficient and portable combined random number generators. *Comm. ACM* **31:6**, 742 (1988).

[7] P. L'Ecuyer. Combined Multiple Recursive Generators. *Operations Research* **44**, 816–822, 1996.

[8] P. L'Ecuyer. Good Parameter and Implementations for Combined Multiple Recursive Random Number Generators. *Operations Research* **47**, 159–164, 1999.

[9] 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).

[10] 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).

[11] P. L'Ecuyer, R. Simard, E. J. Chen and W. D. Kelton. An Object-Oriented Random-Number Package with Many Long Streams and Substreams. *Operations Research* **50**, 1073–1075, 2002.

[12] M. Mascagni, D. Ceperley, and A. Srinivasan. SPRNG, A scalable library for pseudorandom number generation. *Proc. Third Int. Conf. on Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing*, June 1998.

[13] N. Masuda and F. Zimmermann. PRNGlib: A Parallel Random Number Generators library. Technical Report CSCS-TR-96-08, CH-6928 Manno, Switzerland, 1996.

[14] Sun Microsystems Inc. Java Platform 1.2 API Specification. http://java.sun.com/products/jdk/1.2/docs/api/.

[15] John Walker. The randomX package for Java. http://www.fourmilab.ch/hotbits/source/randomX/randomX.html.