A Fast High-Quality Pseudo Random Number Library For Java [*]

Muhammad Hydari [†] David Ceperley [‡] Ashok Srinivasan [§]
Michael Mascagni [¶]

July 23, 1998

Abstract

The need for fast high-quality pseudo random number generators arises in many scientific applications. However, implementing a pseudo random number generator that can be used in parallel without inter-stream correlation or communication is not trivial. We have previously developed such a library (SPRNG) for FORTRAN and C/C++ users. In this article, we describe a port of this library using Java Native Interface thus making available the same fast, high quality pseudo random number generators to Java programmers as are used by FORTRAN and C/C++ programmers.

1 Introduction

The Java programming language has received substantial consideration as a programming language for scientific and engineering computations, in particular those that require a large amount of computing resources. In the future, such resources may be found on a heterogeneous "grid" of processors. For such applications, it has been suggested that Java has the potential to provide a

[*]The SPRNG library was developed with support of DARPA grant number DABT 63-95-C-0123 and with the use of NCSA computational resources. Work on the Java version of SPRNG was supported by a grant from Sun Microsystems, Inc. **This has been submitted for publication to "Scientific Programming". John Wiley will be copyright owner if this is accepted for publication. It is being made available in this format for maximum dissemination of this research.**

[†]NCSA and Department of Computer Science, University of Illinois, Urbana, Illinois 61801(hydari@uiuc.edu)

[‡]NCSA, University of Illinois, Illinois, 61801(ceperley@uiuc.edu)

[§]Indian Institute of Technology, Bombay, India. This research was conducted when this author was at NCSA (ashok@math.iitb.ernet.in)

[¶]University of Southern Mississipi, Hattiesburg, Mississipi 39406 (Michael.Mascagni@usm.edu) better environment than other languages such as FORTRAN and C++ [1]. This is because Java has certain characteristics such as portability (write once, run anywhere), thorough run time checking, security, easy to use libraries for graphical user interface, and network programming constructs. The syntax is close enough to C++ that an experienced C++ programmer can become productive in Java in a few days.

Despite these advantages, a number of potential roadblocks have been recognized by the scientific programming community such as floating point efficiency, complex numbers, multi-dimensional arrays, serialization, remote method invocation, thread synchronization, thread scalability, class versioning (standarization) in which Java needs improvement before it can become an important programming language for scientific applications. For further details the reader should see [2]. We have decided to port a classical Monte Carlo simulation to Java to make performance comparison with simulations written in FORTRAN and C/C++. For this purpose, as well as for many other related applications we needed to have available a standard, parallel pseudo random number generator.

Monte Carlo simulations often require fast high quality pseudo random numbers. As individual processors are becoming faster, and as users perform larger simulations using multiple platforms, the quality of the generator becomes more important. As the simulations become larger, the statistical resolution becomes better and it becomes more important that the pseudo random number generator provide statistically uncorrelated numbers. This is no different in principle than the quality of any other mathematical function such as, for example, the square root function. In practice, however, it is much more difficult to characterize the "quality" of a pseudo random number generator, since one can not definitively determine whether a finite set of numbers is sufficiently random. We do not enter into this issue here but refer the reader to a recent review [3] and to other references on pseudo random number generation and testing [4, 5].

To run simulations in parallel it is necessary to provide a different pseudo random sequence for each process as communication of the numbers is too slow and error prone. The SPRNG library addresses this issue by parameterizing a family of generators. When new processes are created, a free sequence is passed to the new process. After the sequence is created no further communication is necessary. For correctness, it is important that the pseudo random number sequence for the various processes not have statistical correlation with the other sequences. For the generators provided in the SPRNG library, this property has been verified by very large statistical tests. In addition, SPRNG allows one to save the state of the simulation (for example at the end of a run, or when a process is moved from one processor to another) and restart it from that point on at a later time.

The Java random number library lacks several of the features mentioned above. It uses a 48 bit Linear Congruential Generator [6] where the random numbers could repeat in less than a week of simulation on today's processors. Parallelization is often accomplished by generating *ad hoc* seeds for different processes where the same sequence may be used leading to significant interstream correlation.

Thus we had three options with regards to random numbers (i) to use the Java random number (ii) to write our own random number library in Java (iii) to develop a native interface to SPRNG

SPRNG provides four different pseudo random number generators, each of them available for serial and parallel Computation: several Linear Congruential Generators including a 64 bit one. A combined generator and a Lagged Fibonacci Generator. It has been ported to several platforms including CRAY T3E, IBM SP2, HP/Convex Exemplar, SGI Power Challenge, SUN, SGI, DEC Workstations and LINUX. These generators have been subjected to some of the largest random number tests involving individual sequences of 10^{13} numbers. Their speed is also competitive with other generators. The faster generators have been found to be about five times faster than drand48 available on UNIX machines [7].

Given the above mentioned advantages of SPRNG, we decided to use the Java Native Interface that was released with Java Development Kit (JDK) 1.1 to develop a native interface for Java programmers to the NCSA SPRNG library. This means that installations still using JDK 1.0 which had its own Native Method Interface will have to upgrade to JDK 1.1 to be able to use our library. Even though the JDK 1.0 Native Method Interface is also supported in JDK 1.1, we preferred JNI because of the following [8]:

- Native code accesses fields in Java objects as members of C structures. However, "Java Language Specification" does not specify how objects are laid out in memory. If a JVM lays out objects differently in memory, then the programmer will have to recompile the native method libraries

- Sun plans to discontinue supporting JDK 1.0 native methods. This means that native methods relying on the old style interface will have to be rewritten.

Vendor specific native method interfaces like Netscape's Java Runtime Interface (JRI) or Microsoft's Raw Native Interface were not considered because they will not be universally available.

JNI is an efficient virtual machine independent standard that allows Java code that runs inside a JVM to interoperate with applications and libraries written in other programming languages like C/C++ or assembler. JNI is useful in circumstances where the application requirements are too platform dependent or when there is a need to rapidly port a legacy library.

2 Implementation

We have implemented our parallel pseudo random number generator as a class called Sprng for which the interface was kept as close as possible to the standard Java random number library. The interface is given below. A discussion on some aspects of the serial and parallel random number generation follows. The user must understand these issues to be able to use the library to its full advantage.

```
public class Sprng {

  // The following four methods are
  // accessor and mutator for the
  //``seed'' and ``parameter''.
  // The default value for seed and
  // parameter is 0.
  public static int getSeed()
  public static int getParam()
  public static void setSeed(int
  argSeed)
  public static void setParam(int
               argParam)

  // This allows a user to set the
  // maximum number of streams
  // that will be created.
  public static void
      setMaxStreamNum(int
argMaxStreamNum)

  // This method may be used to
  // explicitly set the interface to
  // multiple before using the
  // constructor
  Sprng(byte[]) public static void
setMultipleInterface()

  // The following four methods
  // are to be used for ``single
  // interface'' only.

  public Sprng() throws
  SprngException
  public Sprng(int argSeed, int
   argParam)
throws SprngException
  public void reInitializeStream()
   throws SprngException
  public void reInitializeStream(int
   argSeed, int argParam)

  // The following two constructors
  // are to be used with ``multiple
  // interface'' only
  public Sprng(int stream) throws
  SprngException
  public Sprng(int stream, int
   argSeed, int argParam)
     throws SprngException

  // This may be used with either
  // ``single interface'' or with
  // ``multiple interface".  Note
  // that you can only create a
  // single stream with the
  // "single interface".
  public Sprng(byte[] buffer)
   throws SprngException

  // This method stores the state of
  //the stream in a byte array
  public byte[] packSprng()

  public double nextDouble()
  public float nextFloat()
  public int nextInt()

  // This prints the state of the
  // stream to the standard
  // output using "C" streams
  public void printSprng()

  // This method frees the memory
  // associated with a stream.  Any
  // calls to the object after the
  // memory is freed may cause a
  //program crash
  public int freeSprng()

  //  This method may be used to
  // spawn other streams from a
  // given stream
  public Sprng[] spawnSprng(int
   noToSpawn)
throws SprngException

}
```

In serial computation, users typically generate a sequence of pseudo random numbers which must not have a intra-stream correlation for the results to be accurate. For naturally parallel applications like Monte Carlo simulations, an identical algorithm can be run in each thread with no further communication between the threads. A different pseudo random number stream is generated for each thread. If one has N processors, this allows one to gather statistics N times faster. If the numbers produced by various streams are correlated then the results are not truly independent and the results could be incorrect or the efficiency could be degraded. Thus a good parallel pseudo random number generator has no inter-stream correlation and no intra-stream correlation.

Since we deal with the possibility of several pseudo random number streams, we need to be able to distinguish between various streams. The user supplies a stream number when constructing a Sprng object. A slightly simplified object construction is possible if the user just needs one pseudo random number stream in his program. In this case he does not have to supply a stream number but is restricted to just one stream in that program. We considered making two separate classes one for single streams and the other for multiple streams but decided against it. It seems to violate the principle of doing just one thing in a class but we thought that the two are too closely related to be classified separately, the only difference arising in construction and availability of some methods.

The following demonstrate a simple use of `Sprng`. The user first creates a pseudo random number stream and then invokes `nextFloat()`, `nextInt()` and `nextDouble()` to obtain approximately one million random numbers. The state of the stream is then saved to a file called `single.state` in this program. If the user wants to continue from this previously saved state, he may do so by creating an object using the constructor `Sprng(byte[])`. Of course he will have to read in the saved state into a byte array from the file.

```
import java.io.*;

class Single {
  public static void main(String[]
  args) throws SprngException {
    Sprng z = new Sprng(); for
    (int i=0; i<333333; i++) {
      z.nextInt(); z.nextFloat();
      z.nextDouble();
    } try {
      DataOutputStream dos = new
               DataOutputStream
(new BufferedOutputStream
 (new FileOutputStream(
                  "single.state")));
      dos.write(z.packSprng());
    } catch(IOException e) {}
  }
}
```

The following code allows a stream to be recreated from a previously saved state as done in the code for class `Single` above.

```
import java.io.*;

class Single2 {
  public static void main(String[]
             args) throws Exception {

    byte[] buffer = new byte[32000];
    Sprng z;

    try {
      DataInputStream dis = new
      DataInputStream
(new BufferedInputStream
   (new
   FileInputStream("single.state")));
      dis.read(buffer);
      dis.close();
    } catch (IOException e) {
      System.out.println(e.getMessage());
      System.exit(0);
    }

    z = new Sprng(buffer);
    System.out.println(" Printing
    5 random numbers in [0,1): ");

    for(int i=0; i<5; i++)
      System.out.println(z.nextInt());

  }
}
```

A more interesting application is possible using the multiple interface as shown below. Here we take advantage of the Java thread library to create multiple threads and then create a stream for each thread. The Java threads may automatically take advantage of multiple processors if the native thread library is available [9] (the default “green” threads will run on a single processor).

```
class Multiple extends Thread {

  public static int number = 0;
  static Sprng [] spawnedStreams;

  public int streamNum;
  Sprng z;


  Multiple() throws SprngException {
    streamNum = number++;
    z = new Sprng(streamNum);
    if (number==1)  spawnedStreams
    = z.spawnSprng(5);
  }

  public void run() {
    for (int i=0; i<2; i++) {
      System.out.println("Stream
      Number: "+streamNum+
" produced" +
 z.nextInt());
    }
    yield();
    System.out.println("Stream
    Number: "+streamNum+
" produced" +
spawnedStreams[streamNum].
nextInt());
  }

  public static void main(String[]
  args) throws SprngException  {
    Sprng.setMaxStreamNum(10);
    for (int j=0; j<5; j++)
      new Multiple().start();
    System.out.println("All threads
    started!");
  }


}
```

In this example, the user first sets the maximum stream number to ten and then creates five threads. Each thread creates its own stream. In addition, the first thread also "spawns" off five extra streams from its stream. The streams spawned in the first thread are globally accessible to other streams so they can use these streams to obtain pseudo random numbers.

3 Comparison

One of the simplest uses of pseudo random number streams is to estimate the value of definite integrals using the Monte Carlo method. Although it is usually beneficial in evaluating multiple integrals in dimensions greater than four to use Monte Carlo [10], our example is in two dimensions. We estimate the value of π by determining the probability that pairs of numbers are in the unit circle ($\pi/4$).

```
import java.util.*;

class PI {
  static Sprng z;
  //static Random z;

  static int countInCircle(int
  noOfSample) {
    int noInCircle=0;
    double x, y;
    for (int i = 0; i < noOfSample;
    i++) {
      x = 2*z.nextDouble() -1;
      y = 2*z.nextDouble() -1;

      if ( (x*x + y*y) < 1)
      noInCircle++;
    }
    return noInCircle;
  }


  public static void main(String[]
  args) {
    int noInCircle=0,
    noOfSample=99999;
    byte[] buffer = new byte[32000];
    try {
      z = new Sprng(0, 16807, 0);
    } catch (SprngException e) {
      System.out.println(e.getMessage());
      System.exit(0);
    }

    //z = new Random(16807);

    noInCircle=countInCircle(noOfSample);

    double p, pi;
    p = Math.PI/4.0;
    pi =
    (4.0*noInCircle)/noOfSample;
    double error = Math.abs(Math.PI
    - pi);
    double stderror = 4 *
    Math.sqrt(p*(1-p)/noOfSample);

    System.out.println("The
    calculated value of PI is " +
    pi + " from " +
       + noOfSample
       + " samples"
       );
    System.out.println("Error =
    " + error + ";  STD Error =
    " + stderror);

  }
}
```

We used all the different generators available with Sprng and compared it with `java.util.Random` available with Java util package and various random number generators available with RngPack [11] developed by Paul Houle at Cornell University. We used wall clock time for our comparison on a Ultra Sparc running Solaris 2.0. The following table shows the execution time for 10^5 samples.

Generator	*Time (seconds)*
java.util.Random	195.39
Sprng (LCG)	79.81
Sprng (LCG 64)	65.88
Sprng (LFG)	59.55
Sprng (CMRG)	81.53
RanPack Ranmar	137.03
RanPack Ranlux (default luxury level)	777.95
RanPack Ranecu	121.48

4 Conclusions

We have successfully ported several research quality pseudo random number generators for use by Java programmers. They have been found to be considerably faster than other available generators. The Java random number streams generate exactly the same bits as generated by C/C++ and FORTRAN code. This is important for reproducibility and testability. The results of the statistical tests establishing their superior quality are available from [7]. The library is available from:

`http://www.ncsa.uiuc.edu/Apps/CMP/RNG/RNG-home.html`

Currently, the generators come as a Java class `Sprng` with the C language kernel accessed using the Java Native Interface. The distribution comes with `makefile` for a few platforms to compile the C code as a dynamically loadable shared object. The user just needs to select a particular platform and random number generator.

Since we used JNI for building the native interface, `Sprng` inherits all the problems associated with JNI. It is not pointer safe and thus insecure. The native part of the library code is not "write once, run anywhere"; rather users will have to compile the C implementation code as a shared object on every platform they want to use it. Once installed, the applications using `Sprng` however will be completely portable.

We plan to implement `Sprng` totally in Java to achieve portability. It will be interesting to see how a pure Java implementation will compare in terms of performance with the native implementation. SPRNG itself will be extended to have several more high-quality, fast parallel generators.

References

[1] The Java Grande Forum. *"The Java Grande Forum CHARTER DOCUMENT (draft)."* http://www.npac.syr.edu/javagrande/jgfcharter.html

[2] The Java Grande Forum. *"Report of First Meeting of Java Grande Forum."* http://www.npac.syr.edu/javagrande/JavaGrandeForum.html

[3] A. Srinivasan, D. M. Ceperley, M. Mascagni. *“Random Number Generators for Parallel Applications.”* http://www.ncsa.uiuc.edu/Apps/CMP/RNG/www/toc.html

[4] D. E. Knuth. *“The Art of Computer Programming, Vol 2: Seminumerical Algorithms, Second Edition.”* Addison-Wesley, Reading, Massachussets, 1981.

[5] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *“Numerical Recipes in FORTRAN, Second Edition.”* Cambridge University Press, New York, NY, 1994.

[6] Sun Microsystems, Inc. *“Class java.util.Random”* http://java.sun.com/products/jdk/1.1/docs/api/java.util.Random.html#_top_

[7] D. Ceperley, M. Mascagni, A. Srinivasan. *“SPRNG Documentation”* http://www.ncsa.uiuc.edu/Apps/CMP/RNG/www/toc.html

[8] Sun Microsystems, Inc. *“Java Native Interface Specification.”* http://www.javasoft.com/products/jdk/1.1/docs/guide/jni/spec/jniTOC.doc.html

[9] Bil Lewis, Daniel J. Berg. *“Pthreads Primer: A Guide to Multithreaded Programming.”* SunSoft Press, 1996, pp. 245-247

[10] Kenneth A. Berman, Jerome L. Paul. *“Fundamentals of Sequential and Parallel Algorithms.”* PWS Publishing Company, Boston, MA, 1997, pp. 645-648

[11] P. Houle. *“RngPack 1.0.”* http://www.honeylocust.com/RngPack/