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

Paul Coddington and Andrew Newell

Distributed & High Performance Computing Group
School of Computer Science, University of Adelaide

JPDC Workshop, April 2004

THE UNIVERSITY OF ADELAIDE
AUSTRALIA

Motivation

- Many large-scale (Grande) applications require lots of high-quality random numbers, e.g. Monte Carlo simulation.
- Grande applications can be performed in parallel on multiple processors using Java threads (or MPI).
- Random Number Generators (RNGs) in the standard Java libraries are inadequate for some Grande applications.
- Some work has been done to extend java.util.Random to address its shortcomings - but no support for concurrency.
- We have added this support in JAPARA.
- Parallelizing standard RNG algorithms is straightforward, but there are some subtleties involved.

RNG Algorithms

- Linear Congruential Generator
  - $X(i) = a * X(i-1) + b \mod M$
  - Period of repetition is M.
- Lagged Fibonacci Generator
  - $X(i) = X(i-p)\ \text{op}\ X(i-q) \mod M$
  - Period is product of M and a power of largest lag p
- Multiple Recursive Generator
  - $X(i) = A1 * X(i-1) + A2 * X(i-2) + \ldots + Ak * X(i-k) \mod M$
  - Period is product of M and a power of k
- Combined Generators
  - $X(i) = Y(i)\ \text{op}\ Z(i) \mod M$, with Y and Z random sequences
  - Period is product of the periods of the two sequences

Random Number Generators in Java

Three RNG APIs in the standard Java libraries:

- java.util.Random
  - Main general-purpose RNG interface
  - `next()` method provides 32 random bits (using a 48-bit LCG)
  - `nextFloat(), nextDouble(), nextInt(), nextLong(), nextBoolean(),` etc.
  - Nice design, but missing features for large-scale scientific computing
- java.Math provides a Random() method
  - Just a simpler interface to `nextDouble()` of `java.util.Random`
- java.security.SecureRandom
  - Targeted at cryptography applications
  - Not suitable for scientific applications - too slow and not reproduceable

java.util.Random API

```
Random();
Random(long seed);

public void setSeed(long seed);

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 double nextGaussian();
```

Additional Features Required

- Simple mechanism for choosing different algorithms.
- Adequate period of repetition for large-scale simulations.
- Excellent default generator.
- Mechanisms for checkpointing the state.
- Arrays of random numbers.
- Support for efficient concurrency (currently uses synchronized methods).

Previous Work

- JAPARA builds on previous work by Mathew, Coddington and Hawick (Proc. HPCN'99).
- Addresses all the inadequacies *except* concurrency.
- Extends the java.util.Random API, so programmers can use the standard Java API calls, with some optional extras.
- Uses the approach of java.security.SecureRandom for choosing from different RNG algorithms
  - `getInstance(String algorithmName)`
  - instantiates a RNG object of class algorithmName
  - Don't need to modify program to use different generator

Extensions to java.util.Random API

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

// Enable checkpointing of generator state
public Object getState();
public void setState(Object seeds);
public Object readState(String filename);
public void writeState(String filename);

// 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);
```

Other Existing Work

- RngPack and randomX provide Java RNGs with a choice of algorithms, but
  - Use customized API
  - Don’t support concurrency
- L’Ecuyer et al. developed a Streams and Substreams package implemented in Java
  - Uses customized API
  - Could in principle be used as a parallel RNG
  - Doesn’t provide a choice of algorithms

Parallel RNGs

- All common RNG algorithms can be parallelized efficiently.
- Approach is to initiate a new, independent generator instance on each process.
- This way, the only data dependency (communication or synchronization) is in initialization (constructor) of generator.
- Straightforward, but non-trivial. Need to ensure that:
  - there is no overlap or correlation between sequences on different processes (or threads).
  - does not create correlations in subsequences on each process
- Has been done for MPI (e.g. SPRNG) and HPF.

Parallelizing RNGs

- Leapfrog
  - Each process p of N gets X(p), X(p+N), X(p+2N), …
- Sequence splitting or Boosting
  - Choose boost large enough so provide long subsequences on each process, but small enough to allow lots of processes.
- Independent sequences
  - Choosing seeds correctly in LFGs gives non-overlapping sequences.
  - But need to ensure seeds are not correlated, otherwise sequences will be correlated.
- Parameterization
  - Mechanism for choosing different generator parameters for each process, i.e. a different generator for each process

Design for Introducing Concurrency

- Want to conform as closely as possible to existing API.
- User instantiates a separate RNG instance for each thread.
- How to specify that after the first generator object is instantiated, all new instances should have seeds set automatically?
- Use a new ParallelRandom class extending existing sequential RNG class (which extends java.util.Random)?
- Don’t override existing constructor syntax - maybe the programmer really does want to set the seeds on each thread.
  - Random() - initialize using seed set from the clock
  - Random(seed) - initialize using specified seed
- Add an additional constructor? But what parameter to pass?

API Extensions for Concurrency

- Require the concept of two different types of seeds:
  - a class seed (static variable in Java)
  - a seed for each different RNG object (instance variable in Java)
- Need a new method to specify that the parallel RNG algorithm should seed the generator
  - setSequenceSeed() is specified in the paper
  - setSeed() is perhaps a better way of indicating what is happening
- An (internal) synchronized method needed to update class seed and set the seed for each new RNG instance.
- Better approach is to just store and update an instance counter rather than a class seed as the static variable
  - Increment counter for every new RNG instance
  - This enables parallel initialization of RNG instances on each thread

JAPARA

- JAva PArallel RAndom Number Generator (JAPARA) .
- Keeps same functionality as our previous work, so can just be used as an improved sequential RNG for Java.
- Provides a choice of several well-known RNG algorithms
  - Combined Multiple Recursive Generator (new default)
  - 64-bit LCG
  - Lagged Fibonacci using multiplication
  - Two combined generators (RANECU, RANMAR)
- Provides efficient concurrent implementation of all these algorithms.

Parallelization Used in JAPARA

- Use boosting for LCG-type generators
  - CMRG, 64-bit LCG, RANECU, all with period > 2^63
  - Choose boost large enough so that subsequences on each thread are long enough, e.g. 2^52
  - But small enough to allow lots of threads, e.g. > 2^11

- Use independent sequences for LFG-type generators
  - LFG with multiplication, RANMAR
  - Choose seeds so non-overlapping subsequences on each thread
  - Need to be sure seeds for each generator instance are not correlated, so use a RNG (simple LFG or 32-bit LCG) to generate the array of seeds required for the LFG.

Using JAPARA

Main thread initializes generator in the usual way, either

```
Random myRNG = new Random(seed);
```

```
Random myRNG = getInstance(generatorName);
myRNG.setSeed(seed);
```

Other threads get the generator to set the seed

```
Random myRNG = new Random();
myRNG.setSeed();
```

```
Random myRNG = getInstance(generatorName);
myRNG.setSeed();
```

Testing the RNG Implementations

- All the RNG algorithms used in JAPARA have previously been subjected to many (sequential) statistical and application tests, so are known to be very high quality.
- We have developed a few simple tests in Java (histogram, average) as a sanity check on our implementation.
- Need to do some more rigorous tests.
- Very little research has been done on testing the quality of parallel RNGs
  - Sequential tests on subsequences on each thread
  - Parallel applications (e.g. Monte Carlo simulations of Ising model)
- We have implemented a program to check correlations between parallel sequences - preliminary results look OK.

Conclusions and Future Work

- JAPARA is an improved random number generator library for Java, aimed primarily at high-end applications.
- JAPARA extends the standard java.util.Random generator to overcome some problems and add useful features, particularly concurrency.

- More thorough testing of randomness properties of JAPARA generators is needed.
- Also test it with a few different parallel Java applications.
- Then release JAPARA code.
- Develop version of JAPARA for Java MPI programs.