

CLEARSPPEED WHITE PAPER: The ClearSpeed Random Number Generator Library

Abstract

This paper gives details of pseudo-random number generation (pRNG) on ClearSpeed parallel architectures.

Several methods of parallelizing random number generators are described along with the capabilities of the ClearSpeed Random Number Generator (RNG) library.

Random Number Generation from ClearSpeed

ClearSpeed RNG Library

Many application areas involving stochastic modeling require random number generation. For example, in computational finance it is common for the value of derivatives to be calculated using Monte Carlo simulations. To aid these application areas, ClearSpeed provides a random number generation library which provides common pseudo-random number generators and deals with parallelizing the random number streams.

Random number streams and states

Pseudo-random number generation produces a stream of numbers uniformly distributed in a certain range. A generator is initialized with a particular seed number which sets the internal state of the generator. Each call to the generator then produces a number based on this state and updates the state in preparation for the next number.

Pseudo-random number generators (pRNGs) do not produce genuinely random distributions so there will be patterns within the stream. The patterns (for example, correlations between sub-sequences) can be characterized and tested in certain well established ways. One of the most common characterizations is the period of the generator. Since generators are deterministic, the sequence will eventually repeat and the period of the generator is the length of the sequence before repetition.

Parallel pRNGS

Pseudo-random number generators are essentially serial devices. Each number is based on the previous updated state. This provides a challenge for a parallel device such as the ClearSpeed CSX600 processor.

In general, parallel pRNGs are based on corresponding serial pRNGs. There are

several techniques that can be applied to parallelize a serial pRNG. There are, however, many types of serial pRNG and certain parallelization strategies only apply to certain types of generators.

Parallelization techniques – independent seeds

One method is to give each parallel processing element (PE) its own independent state and initialize the whole parallel array with different seeds. This method is simple to implement and may be appropriate if, for example, one is running a separate Monte Carlo simulation run on each PE.

If one is aggregating the results of the streams on each PE, however, then the difficulty is ensuring that the distribution properties (underlying correlations, periodicity) of the combined streams are “random enough”. If the combined result has poor distribution properties then it can skew the overall results. Choosing seeds to try and ensure good randomness properties is still an open problem and varies with each type of generator.

Leapfrogging and splitting

Suppose a pseudo-random number generator produces the stream:

$$r_0, r_1, r_2, r_3, r_4, \dots$$

There are techniques available to split that single stream up amongst the parallel array. The leapfrog and splitting methods are suitable where the results are to be aggregated.

For example, running one Monte Carlo simulation across the whole array would be suitable for leapfrog or splitting since one simulation uses one random number stream (split up amongst the PEs). However, these methods may not be suitable for running independent simulations on each PE, since each PE will only get a sub-sequence, and that sub-sequence may have skewing



correlations that the entire stream does not have.

The ClearSpeed CSX600 processor has 96 PEs, so we shall use that number of elements for the examples presented here.

The leapfrog technique will split the stream such that PE 0 gets numbers $r_0, r_{96}, r_{192}, \dots$, PE 1 gets $r_1, r_{97}, r_{193}, \dots$ and so on. The effect is similar to when a deck of cards is dealt out amongst different players. The stream split amongst the whole array looks like:

PE 0	PE 1	...	PE95	PE96
r_0	r_1	...	r_{94}	r_{95}
r_{96}	r_{97}	...	r_{190}	r_{191}
r_{192}	r_{193}	...	r_{186}	r_{187}
...

Block splitting requires the programmer to specify at initialization how many numbers each processing element will receive (say 1000). In this case PE 0 gets number r_0, r_1, \dots, r_{999} , PE1 gets $r_{1000}, r_{1001}, \dots, r_{1999}$ and so on. The stream split amongst the whole array looks like:

PE 0	PE 1	...	PE 95	PE 96
r_0	r_{1000}	...	r_{95000}	r_{96000}
r_1	r_{1001}	...	r_{95001}	r_{96001}
r_2	r_{1002}	...	r_{95002}	r_{96002}
...
r_{999}	r_{1999}	...	r_{95999}	r_{96999}

Given the serial nature of many pRNGs, it would seem impossible to generate these sub-sequences independently using true parallelism. However, for linear congruential generators (LCG), this is possible.

LCGs work by generating numbers according to the recurrence relation:

$$r_N = (a \times r_{(N-1)} + c) \bmod N$$

In this case it is possible to work out for any integer k , an a' and c' such that:

$$r_N = (a' \times r_{(N-k)} + c') \bmod N$$

This allows us to calculate jumps in the random number sequence on one processing element without referring to the state of the other elements.

On the ClearSpeed CSX600 we can use this technique to split pseudo random generators across the processing array on the chip and also across multiple CSX600s on multiple ClearSpeed Advance™ accelerators in a system.

The Mersenne Twister and parallelization

The Mersenne Twister is an important pseudo random number generator invented in 1997 by Makoto Matsumoto and Takuji Nishimura. It is actually a family of pRNGs with differing periods. A common version of the generator is the MT19937 generator. This generator has a period of $2^{19937} - 1$. All the generators generate numbers according to the following relation:

$$r_{(k+n)} = T[A[r_{(k+m)} \oplus (x_k^u | x_{(k+1)}^l)]]$$

The specific generator is specified by the values of n, m, k , the bitmasks u and l and the twisting and tempering transformations A and T . The details of the recurrence are too involved to be described here, and readers are directed to the original Matsumoto and Nishimura paper¹ for details. However, the complexity of the recurrence has two corollaries:

- the state of the generator is larger than for simple LCG generators and numbers are generated several at a time (for example, for the MT19937 variant 624 numbers are generated at once),
- there is no known closed form to leapfrog ahead in the sequence.



Since we cannot jump ahead with this generator, the leapfrog and block splitting methods are not applicable. The multiple seed method will still work but it is difficult to choose the seeds to give good randomness properties.

The authors of the generator do have another method of parallelization. It is called *dynamic creation*² and involves selecting different parameter values (of the bitmasks u and l, and the T function) for each PE. This way, each stream will be independent but the combined streams will still have good randomness properties. The dynamic creation only needs to be done once and then the results can be used for many different simulation runs.

The dynamic creation algorithm is available in the public domain (from the Mersenne Twister's creator's website³). The ClearSpeed RNG library can take the output of this algorithm and execute the resulting parallel Mersenne Twisters. The library also supplies pre-generated parameters for up to 960 MT2203 generators.

One drawback of this method is that the dynamic creation is so computationally intensive that it becomes infeasible for the MT19937 generator and will only work in practice for lower period generators (though these still have considerably long periods, the MT2203 generator has a period of $2^{2203} - 1$ for example).

Parallel pRNGs and the CSX600

The techniques in this document have been implemented as part of the ClearSpeed RNG library for the Cn language, a parallel extension to C. All the techniques are inherently Single Instruction Multiple Data (SIMD) in their nature, only the data (parameters/state) of the generation change across the PE array, the control flow is the same across the array.

```

poly cs_mcg31m1_state sp;
cs_mcg31m1_stream s;

poly double x;
int i;

cs_init_rng_multiseed(
    mcg31m1,
    &s,
    &sp,
    1234);

for (i=0; i<10; i++)
    x +=
cs_drand(mcg31m1, &s);
    
```

Figure 1: Sample RNG library code

RNG library usage

The ClearSpeed RNG library provides a simple uniform interface to a range of pseudo random number generators.

There is a stream type for each generator. Once this stream type is initialized, random numbers can be taken from the stream. How the parallelization happens depends on the initializing call.

Figure 1 shows a sample piece of code that sums the first ten double precision numbers provided by the stream which uses the mcg31m1 generator and has been initialized using the multiseed method.

When using a stream each call to get a random number will calculate a random number for every PE in parallel.

	Multiseed	Leapfrog	Splitting	DC
rand48	✓	✓	✓	
mcg31m1	✓	✓	✓	
mcg59	✓	✓	✓	
MT	✓			✓

Figure 2: Parallelization techniques

CLEARSPD RNG LIBRARY

The generators currently available in the library are:

- rand48
- mcg31m1
- mcg59
- Mersenne twister variants

The first three of these are LCG generators, so the look-ahead method described previously will work. Figure 2 summarizes the parallelization techniques available for each type of generator. The random number generators in the library have been optimized for the ClearSpeed CSX architecture and provide a very high level of performance.

Summary

This white paper has explained the challenge of random number generation on parallel architectures. The ClearSpeed random number generator library provides users with an off-the-shelf solution for developers who need random number streams in their code.

¹ M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator", ACM Trans. on Modeling and Computer Simulation Vol. 8, No. 1, January pp.3-30 (1998)

² Makoto Matsumoto and Takuji Nishimura, "Dynamic Creation of Pseudorandom Number Generators", Monte Carlo and Quasi-Monte Carlo Methods 1998, Springer, 2000, pp 56--69.

³ <http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html>

The logo for ClearSpeed, featuring the word "ClearSpeed" in a blue, sans-serif font. The "Clear" part is in a lighter blue, and "Speed" is in a darker blue. A horizontal bar with a blue-to-yellow-to-orange gradient is positioned below the text.