Exploitation of the Potential of Cell Architecture for Parallel Random Number Generation

K.Gunaranjan and Pallav Kumar Baruah
Dept. of Mathematics and Computer Science, Sri Sathya Sai University.

Abstract

The Cell Broadband Engine Architecture designed by Sony, Toshiba and IBM is a heterogeneous multicore processor. It has tremendous potential for performing scientific computations. Random number generators are one of the most common numerical library functions used in scientific applications[1]. SPRNG is a widely used library for parallel random number generation. In this work, we ported SPRNG onto the Cell Broadband Engine (Cell BE). We report optimizations that enable SPRNG to perform an order of magnitude faster than the initial port, which will help developers of computation-intensive numerical libraries to optimize their code on the Cell BE processor. We have also added additional parallelization strategies to SPRNG, and discuss the unsafe uses, which should be avoided. This will enable users of random number generators to make more effective use of the libraries.

1. Introduction

The introduction of multi-core processors has opened up the possibility of parallel computation on a single chip. The Cell Broadband Engine Architecture provides supercomputing power on a single chip. It consists of a PowerPC core (PPE), which acts as the controller for eight SIMD cores called Synergistic Processing Elements (SPEs), which handle most of the computational workload. At 3.2 GHz, each SPE gives a theoretical 25.6 GFlops of single precision performance, giving it tremendous potential to perform scientific computations.

Scientific and technical computing problems, solved using stochastic simulation-based computational methods such as Monte Carlo methods, are based on random sampling. These methods require a random number generator (RNG) of high quality that produces numbers at a high speed. Quality here refers not only to the randomness properties of individual streams of random numbers but to the correlation properties between streams that are used in parallel. The Parallel Random Number Generators parallelize a sequential generator by taking the elements of the sequence of pseudo-random numbers it generates and distributing them among the processors. We will discuss the Leap frog facility and Leap ahead facility in section 5.

The Scalable Parallel Random Number Generators Library (SPRNG) is a set of libraries for scalable and portable random number generation, and has been developed especially to suit large-scale, parallel Monte Carlo applications. It is an easy-to-use package and runs on almost any computing architecture. SPRNG was developed at NCSA and Florida State University. SPRNG provides a variety of good quality sequential and parallel random number generators, with good performance in terms of the number of random numbers generated per second.

We also report optimizations of one of the most popular SPRNG generators, the Linear Congruential Generator (LCG), on the SPEs of the Cell processor, that enable SPRNG to perform an order of magnitude faster than the initial port.

2. Parallel Pseudo Random Number Generator and SPRNG

SPRNG [1] is a library of parallel RNGs which uses parameterization. Its generators have been subjected to some of the largest tests of parallel random number generators [2]. It also contains a variety of generators, so that users can verify their results by comparing the results obtained from different generators. Each of these generators has several variants which can be selected using certain parameters in an initialization routine. SPRNG also contains a facility to dynamically create new random number streams which will be different from streams on any other processor. Furthermore, this is accomplished without any inter-processor communication. SPRNG also contains a facility to checkpoint and load the state of random number sequences in a machine independent manner. More details on SPRNG and on parallel random number generation are available in [1,2,5,6].

3. Implementation of SPRNG on Cell

The Cell BE is an architecture for distributed computing and SPRNG was designed to provide support for distributed multiprocessor streams. The SPRNG on Cell BE is used mainly for applications that are parallelized and run on more than one SPE. We used the *SPE centric model* where the PPE spawns threads on the SPEs and then waits for the threads to complete. All the computations are done on the SPEs. Each SPE calls a different random number sequence and executes library calls sequentially with the random number sequence associated with it.

The generators in SPRNG ported onto the Cell BE are given in the table 1.

.

Name	Generator	Period
LCG	48-bit Linear Congruential Generator	2^{48}
LCG64	64-bit Linear Congruential Generator	2^{64}
LFG	Modified Additive Lagged Fibonacci Generator	Up to 2^{1310}
MLFG	Multiplicative Lagged Fibonacci Generator	$> 10^{23}$
CMRG	Combined Multiple Recursive Generator	2^{219}

Table 1: Generators ported onto the Cell

We have also ported the test suite in SPRNG which can be used to test the quality of parallel and sequential random number generators. The port also includes applications built by SPRNG to test the correctness of the distribution and which time each of the generators.

4. Optimizations

Here we report optimizations that were implemented to improve the performance of LCG after the initial port of SPRNG on Cell. On building the SPRNG library, executables are created to time each generator provided. A shell script called timesprng is provided that will run each of these timing executables. The output will give the time taken to generate a million random numbers and also the number of random numbers generated per second, in millions . These results are based on the timings in C programs. Table 2 has the results after the initial port of SPRNG onto Cell.

Generator	MRS
Integer	$44 * 10^6$/s
Float	$45 *10^6$/s
Double	$45 *10^6$/s

Table 2: Results after the Initial Port

4.1 Array Implementation

Here we implemented a facility which returns an array of numbers. This facilitates some compiler optimizations, yielding better performance. Table 3 shows the performance results for an array size of *8*. One observation we made here is that the optimal performance we reached with array size 8. The performance results for larger array sizes were not so encouraging.

Generator	MRS
Integer	$65 *10^6$
Float	$67 *10^6$
Double	$60 *10^6$

Table 3: Array Implementation (array size 8)

4.2 Optimized Multiplication routines

Since the multiplication routines formed the core of the code, we optimized the 64-bit multiplication using the algorithm that is similar to the conventional multiplication performed by hand. We used code developed by Neil Costigen from Dublin City University. This led to a improvement in the performance . The results are shown in the table 4.

Generator	MRS
Integer	$400 * 10^6$
Float	$402 * 10^6$
Double	$400 * 10^6$

Table 4: Optimized Multiplication routines

4.3 Vectorization based on using the Recurrence Relation

In any random number generator, the state after any iteration depends on the value of the previous state. Using the profiling tool on the SDK 2.1, it was observed that a large fraction of time being stalled was due to data dependency . The recurrence relation can be used to remove the data dependency that exists between iterations. This allows computations for different iterations to be simultaneously done. Here we show the results obtained for the LCG generator. The recurrence of the LCG is given by $x_{i+1} = ax_i + b \bmod 2^{48}$, where x_i are the states, and a and b are constants. This recurrence yeilds [3,4] $x_{i+1} = \alpha x_i + \beta \bmod 2^{48}$, where $\alpha = a^2 \bmod 2^{48}$ and $\beta = b + ab \bmod 2^{48}$. The terms α and β can be precomputed. Now, if we know x_0 and x_1, then we can, in principle, simultaneously compute x_2 and x_3. We see that x_3 can be computed independently of x_2 . This allows us to compute two terms every iteration. Vectorization was used to achieve this. The results (table 5)obtained after this were very encouraging.

Generator	MRS
Integer	$449 * 10^6$
Float	$569 * 10^6$
Double	$465 * 10^6$

Table 5: Vectorization using the recurrence relation

5. New Features

In this section we discuss about some new features that have been added to SPRNG i.e., Leap Ahead facility and Leap Frog Facility.

5.1 Leap Ahead Facility

We Implemented the Leap Ahead Facility which helps with parallelization based on the blocking strategy. It can help when a large number of streams is required, but only a small amount of numbers from each stream is used. It is also very useful for generators with large state space like the LFG i.e. If the user stores the initialization parameters and the leap, we can reconstruct the sequence and move to the correct position in the sequence.

5.2 Leap Frog Facility

We implemented a new function ***sprng_leapfrog(void *p, int init_leap, int subsequent_leaps)***. This causes the random number stream pointed to by *p* to leap to a point *init_leap* elements away in the sequence. This is based on the Leap Ahead Facility.

6. Conclusions and future work

The CBEA has a growing range of development tools and has many standard libraries. Optimal performance for any application can be achieved by exploiting the functionality provided by these libraries and using the computational power of the eight SPEs. The optimizations implemented by us increased the performance by an order of magnitude compared with the initial porting. The optimization process described here may benefit writers of other libraries for the Cell BE processor. We have also added newer features into SPRNG which will enable a greater variety of parallelization strategies.

As our future work, we plan to optimize the other generators and include the additional features in them too.

7. Acknowledgements

We thank Neil Costigan from Dublin City University for providing an optimized implementation of 64-bit integer multiplication on the SPEs. We also thank the Sony-Toshiba-IBM Cell Center of Competence at Georgia Tech for providing use of their IBM QS20 Cell blades, and IBM for providing access to their Cell blades under the VLP program. Most of all, we express our gratitude to Sri Sathya Sai Baba, the chancellor of Sri Sathya Sai University, for bringing us all together to perform this work

8. References

1) M. Mascagni and A. Srinivasan, *SPRNG: A Scalable Library for Pseudorandom Number Generation*, ACM Transactions on Mathematical Software, vol 26 (2000) 436-461.

2) A. Srinivasan, M. Mascagni, and D.M. Ceperley, *Testing Parallel Random Number Generators*, Parallel Computing, vol 29 (2003) 69-94.

3) S.L. Anderson, *Random Number Generators on Vector Supercomputers and Other Advanced rchitectures*, SIAM Review, vol 32 (1990) 221-251.

4) D.E. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, third dition, Addison-Wesley, 1998, pages 10-11.

5) M. Mascagni and A. Srinivasan, *Parameterizing Parallel Multiplicative Lagged-Fibonacci Generators*, Parallel Computing, vol 30 (2004) 899-916.

6) A. Srinivasan, D.M. Ceperley, and M. Mascagni, *Random Number Generators for Parallel Applications*, in Advances in Chemical Physics, Volume 105, Monte Carlo Methods in Chemical Physics, Editors: D. Ferguson, J.I. Siepmann, D.G. Truhlar, John Wiley and Sons,Inc, 1999, pages 13-36.