Parallel random numbers, simulation, and reproducible search

Survey Paper

Martin Benito-Rodriguez
Université Clermont Auvergne
Clermont-Ferrand, France

***Abstract*—This article is a popularization of David Hill's article [1]. The generation of pseudorandom numbers in a parallel context must be of good quality in the field of parallel stochastic simulations. One has to make sure that the results are reproducible to do research, but one can have very different results if one does not know the right methods. After a parallel stochastic simulation, we can verify the results with a sequential counterpart.**

Keywords; generators; pseudo-random number; parallelism; simulation; stochastic; reproducibility; independent flows; High Performance Computing

I. INTRODUCTION

In the experimental sciences, reproducibility is at the heart of research, the same is true for computer simulations.

To test, verify and find errors and ensure the validity of experiments and computers, we must replicate the results, that is, be able to find the same results with the same algorithms and the same input data.

However, it is becoming increasingly clear that even if all of the above parameters are met, reproducibility is not assured. And the need for reproducibility is more and more important these last years. In the field of parallel stochastic simulation, it is mainly the lack of academic education that is responsible for the lack of reproducibility. We need to understand how to run reproducible parallel stochastic simulations of good quality without the parallel flows influencing each other.

Since 1998 and the article "Don't Trust Parallel Monte Carlo!" by Peter Hellekalek [1], many advances have been made. These advances are mainly due to the requirements of nuclear safety which needs random number generators with a good distribution. Other fields such as nuclear medicine need for their simulations more than 10^20 pseudo-random numbers to make calculations on several thousands of processors. The random numbers must be generated by different independent streams, which requires the use of parallelism techniques.

II. INDEPENDENT STOCHASTIC FLOWS FOR PARALLEL SIMULATIONS

We will see here two methods to have random number streams with parallelism.

A. Partition a single stream of random numbers:

The most basic method for partitioning a pseudo-random number generator is to use a central server that manages the single generator. Methods based on cellular boolean automata can generate pseudo-random numbers in a parallel context, but they are not used in high performance computing.

The sequence *splitting/blocking/regular spacing* method consists in dividing the sequence into N contiguous blocks.

The *random spacing* method initializes the generator with different random parameters.

The last method is the *leap frog* method where random numbers are distributed between the different processes as when cards are dealt to players.

B. Produce multiple independent streams

To have our independent flows, we will need independent parallel generators that belong to the same family, this is called *parametrization*.

For the Mersenne Twisters (MT) family of generators [2], the tool for creating the most independent generators is *Dynamic Creator* (DC) [3].

TinyMT and MTGP (used for GP-GPU) share the DC approach.

We can also use the Multiplicative Lagged Fibonacci Generators (MLFG) [4].

Each of the advantages and disadvantages of these techniques are explained in the article [5].

III. Testing and compatibility

An important thing to know is that we cannot mathematically prove the independence of parallel streams. But there are still avenues of research to avoid correlations between different parallel streams [6] [7].

Generators for parallel computing must be tested with a collection of tests called BigCrush Test U01 [8]. Generators that pass the tests are said to be "crush-resistant" [9], but they are not perfect.

One can also use the advice of Srinivasan and Mascagni [10], the article is based on Mascagni's proposals for the generation of parallel pseudorandom numbers [11].

There are still other techniques listed in this article [12].

The following algorithms for random number distribution in a parallel environment are interesting for multicore architectures.

-MRG32k3a

-MLFG6331_64

-MTGP/TinyMT

-Phylox/Threefry

For stochastic simulations, it is better to use several types of generators to determine the stochastic variability. Those presented here belong to different families and have different structures.

IV. Creation of reproducible parallel stochastic simulations

For parallel stochastic simulation, it is difficult to get good results in different contexts. To rephrase: it is difficult to obtain the same results by changing parameters like the number of processors. To keep a good quality, we can follow several tracks: an object-oriented approach [13][14], the use of modern and statistically robust generators, the use of parallelization techniques adapted to the generator, a parallel design method starting from the sequential program design. The details to understand this last point are as follows:

-create a sequential program with reference data as input, but with independent stochastic processes that have their own pseudo-random number stream.

-The independent stochastic flows must be created with one of the methods shown above: several independent parallel generators or a generator that creates several independent flows. It is necessary to remember to save the parameters of the flows to find the same sequence (to ensure reproducibility).

-distribution of the calculation between all the processes

-Verification that parallel execution with a different number of processes gives the same result as sequential execution if the initialization and parameters are identical. Verify the reduction procedure with its sequential counterpart. Before large-scale deployment, repeatability should be verified with a meaningful sequential execution.

This technique allows to have reproducible parallel stochastic simulations and a comparison with a sequential implementation. Even if this system does not solve all the reproducibility problem, it remains an interesting advance.

However, GP-GPU architectures do not necessarily allow digital reproducibility. According to Intel, bit-to-bit reproducibility is impossible with an Intel Xeon Phi [15]. One should also be even more careful in a hybrid context, e.g. with different compilers or processors.

To have perfectly identical results from one experiment to another with the same inputs and parameters in a parallel environment, we would have to be able to control the order of all the operations. But if we want to have a good level of reproducibility today, we need to adopt methods such as those mentioned in the article, continue to do research and communicate between computer scientists to have good practices.

References

[1] P. Hellekalek, “Don’t Trust Parallel Monte Carlo!” Proc. Parallel and Distributed Simulation Conf., 1998, pp. 82–89.

[2] M. Matsumoto and T. Nishimura, “Mersenne Twister: A 623-Dimensionnally Equidistributed Uniform Pseudorandom Number Generator,” ACM Trans. Modeling and Computer Simulations, vol. 8, no. 1, 1998, pp. 3–30

[3] M. Matsumoto and T. Nishimura, “Dynamic Creation of Pseudorandom Number Generators,” Monte Carlo and Quasi-Monte Carlo Methods, H. Niederreiter and J. Spanier, eds., Springer, 1998, pp. 56–69

[4] M. Mascagni and A. Srinivasan, “Parameterizing Parallel Multiplicative Lagged-Fibonacci Generators,” Parallel Computing, vol. 30, 2004, pp. 899–916

[5] D.R.C. Hill et al., “Distribution of Random Streams for Simulation Practitioners,” Concurrency and Computation: Practice and Experience, vol. 25, no. 10, 2013, pp. 1427–1442.

[6] A.D. De Matteis and S. Pagnutti, “Parallelization of Random Number Generators and Long-Range Correlations,” Numerische Mathematik, vol. 53, no. 5, 1988, pp. 595–608.

[7] A.D. De Matteis and S. Pagnutti, “Controlling Correlations in Parallel Monte Carlo,” Parallel Computing, vol. 21, no. 1, 1995, pp. 73–84.

[8] P. L’Ecuyer and R. Simard, “TestU01: A C Library for Empirical Testing of Random Number Generators,” ACM Trans. Mathematical Software, vol. 33, no. 4, 2007; www.iro.umontreal.ca/~lecuyer/myftp/.../testu01.pdf

[9] J.K. Salmon et al., “Parallel Random Numbers: As Easy as 1, 2, 3,” Proc. Int’l Conf. High Performance Computing, Networking, Storage and Analysis, 2011, pp. 11–12.

[10] A. Srinivasan, M. Mascagni, and D. Ceperley, “Testing Parallel Random Number Generators,” Parallel Computing, vol. 29, no. 1, 2003, pp. 69–94.

[11] M. Mascagni, D. Ceperley, and A. Srinivasan, “SPRNG: A Scalable Library for Pseudorandom Number Generation,” ACM Trans. Mathematical Software, vol. 26, no. 3, 2000, 436–461.

[12] P.D. Coddington and S.-H. Ko, “Random Number Generator for Parallel Computers,” Proc. 12th Int’l Supercomputing Conf., 1998, pp. 282–288.

[13] D.R.C. Hill et al., “Distribution of Random Streams for Simulation Practitioners,” Concurrency and Computation: Practice and Experience, vol. 25, no. 10, 2013, pp. 1427–1442

[14] O.J. Dahl and K. Nygaard, “Simula: An ALGOLbased Simulation Language,” Comm. ACM, vol. 9, 1966, pp. 671–678.

[15] M. Taufer et al., “Improving Numerical Reproducibility and Stability in Large-Scale Numerical Simulations on GPUs,” IEEE Int’l Symp. Parallel and Distributed Processing, 2010, pp. 1–9.