Available online at www.sciencedirect.com
ScienceDirect
Mathematics and Computers in Simulation 135 (2017) 3–17
MATHEMATICS AND COMPUTERS IN SIMULATION
www.elsevier.com/locate/matcom

Original articles

Random numbers for parallel computers: Requirements and methods, with emphasis on GPUs

Pierre L'Ecuyer[a,b,*], David Munger[a], Boris Oreshkin[a], Richard Simard[a]

[a] *DIRO, Pavillon Aisenstadt, Université de Montréal, C.P.6128, Succ. Centre-Ville, Montréal (QC), Canada H3C 3J7*
[b] *Inria Rennes Bretagne Atlantique, Campus Universitaire de Beaulieu, 35042 Rennes Cedex, France*

Received 20 March 2014; received in revised form 17 April 2015; accepted 27 May 2016
Available online 7 June 2016

Abstract

We examine the requirements and the available methods and software to provide (or imitate) uniform random numbers in parallel computing environments. In this context, for the great majority of applications, independent streams of random numbers are required, each being computed on a single processing element at a time. Sometimes, thousands or even millions of such streams are needed. We explain how they can be produced and managed. We devote particular attention to multiple streams for GPU devices.

Keywords: Random number generators; Multiple streams; Parallel computing; Simulation; Monte Carlo

1. Introduction

Random number generators (RNGs) are fundamental ingredients for stochastic simulation and for the application of Monte Carlo methods in general. Conceptually, these RNGs are designed to produce sequences of real numbers that behave approximately as the realizations of independent random variables uniformly distributed over the interval $(0, 1)$, or i.i.d. $\mathcal{U}(0, 1)$ for short [23,24,26,34,60]. These numbers are then transformed to simulate random variables from other distributions, stochastic processes, and other types of random objects [11,19].

In this review paper, we examine the design and implementation of RNG facilities for parallel processing, with a special attention to discrete *graphical processing units* (GPU) devices. Such facilities are increasingly important, because computing power now improves by increasing the number of *processing elements* (PEs), that can operate in parallel in a computer, rather than by increasing clock speeds of the PEs [4,56], mainly because faster clock speeds produce too much heat.

Parallel computers are generally organized in a hierarchic fashion, both in terms of PEs and memory. A single chip may contain several PEs (or cores) that execute instructions in parallel. In many cases, several PEs can execute different instructions in parallel, almost independently, with only sparse communication between them. In some cases,

* Corresponding author at: DIRO, Pavillon Aisenstadt, Université de Montréal, C.P.6128, Succ. Centre-Ville, Montréal (QC), Canada H3C 3J7.
E-mail address: lecuyer@iro.umontreal.ca (P. L'Ecuyer).

http://dx.doi.org/10.1016/j.matcom.2016.05.005

certain groups of PEs on the chip work in the *single instruction multiple data* (SIMD) mode, which means that they must all execute the same instructions at each time step, usually with different data. This happens in array processors and GPU devices, for example, which are used as accelerators to perform the same instructions in parallel on an array (or vector) of different data. The purpose of this restriction is to speed up the processing by reducing the work required for task scheduling. On the other hand, it puts important constraints on the design of algorithms that run on such hardware.

Most often, there is a mixture of these two cases. The largest supercomputers in 2015 have over a million PEs. The PEs execute *threads*, where a thread can be seen as a sequence of instructions that can execute on a single PE (although the notions of thread and PE can have various meanings that differ in the scientific and commercial literature). The memory has a matching hierarchical organization, in which each PE can have access to a small amount of very fast-access memory, then groups of PEs share a larger memory that is slower to access, and there can be many levels of that. There is a large variety of such hierarchic organizations. On some common chips, all the PEs have fast access to a large shared memory (with a single address space) of several Gbytes. On others, such as discrete GPU chips, each PE has a fast read–write access only to a small private memory for its data, fast read-only access to a larger shared memory, and much slower access to the shared read–write memory. The read and write access times depend on the type of memory and on how it is physically connected to the PEs or to the host (on the same chip, on the same board, in the same machine, in a different machine from a same cluster, etc.).

In recent years, there has been a strong interest and much published articles on RNGs for GPU devices. Those devices were originally designed for fast high-quality image rendering on computer screens and video-game consoles, but are now widely used as low-cost alternatives to accelerate computations in several other areas such as statistics, computational finance, and simulation in general, and have been extended to general-purpose GPUs (GPGPUs). This special interest for RNGs that run on GPU devices comes from the strong restrictions (mentioned earlier) that one faces when computing on those devices, and which are not present on other common types of processors such as ordinary CPUs (single-core or multi-core) and parallel processors typically found in the largest supercomputers. Our review gives special emphasis to GPUs for this reason. A traditional GPU board (named *discrete* GPU) contains specialized chips, each one containing multiple copies of a hardware unit with PEs that can execute a *warp* (or *wavefront*) of (typically 32 or 64) threads (also called *work items*) in parallel, in a SIMD fashion. The actual number of physical PEs is typically less than 32 (e.g., it can be 16, with one instruction per thread every two or four clock cycles). There are several variations in the architectural design, and explaining this is beyond the scope of the present article. The important aspects for our purposes are the SIMD restriction for groups of threads and the limited size of fast access private memory per thread, mentioned earlier.

To take advantage of new parallel-computing architectures, and account for their specificity and constraints, algorithms and software in general, and for RNGs in particular, must be adapted and sometimes radically redesigned. Feeding all the PEs with a single source of random numbers is generally unacceptable, first because it would create too much overhead and a large bottleneck. In highly-parallel systems, one may need thousands or even millions of virtual RNGs. They can be either different RNGs or copies of the same RNG starting from different states, that run in parallel without exchanging data between one another, and behave from the user's viewpoint just like independent RNGs. These virtual RNGs are often called *streams* of random numbers.

Another important requirement is *reproducibility of the results*: it is often required that simulations must be exactly replicable and produce exactly the same results on different computers and architectures, either parallel or purely sequential, and when running the program again on the same computer. The latter is important for debugging and in the (frequent) situation where we want to simulate a complex system with slightly different configurations or decision making rules (e.g., to optimize these rules), while making sure that exactly the same random numbers are used at exactly the same places in all configurations of the system, and repeat this n times independently. To reduce the variance in the comparisons, it is important to use well-synchronized *common random numbers* (CRNs) [24,32,33,36,45,41]; that is, to ensure that the same random numbers are used for the same purpose across configurations. To achieve this, one would use a separate stream for each required source of random numbers in the system, and use one substream for each of the n independent runs [24,32,33,36,41].

Reliable software facilities that provide RNGs with multiple "independent" streams and substreams have been available for some time for that purpose [33,37,45,38]. Most were initially designed primarily for simulations over a single processor, for synchronizing CRNs, but they can be used as well to provide multiple streams for parallel processing. In particular, the RngStream package of [45] has been incorporated in the `parallel` package which