Random Number Generation Tools for Distributed Simulation on Modern HPC Architectures

Prof. Michael Mascagni

Department of Computer Science
Department of Mathematics
Department of Scientific Computing
Graduate Program in Molecular Biophysics
Florida State University, Tallahassee, FL 32306 **USA**
AND
Applied and Computational Mathematics Division, ITL
National Institute of Standards and Technology, Gaithersburg, MD 20899-8910 **USA**
E-mail: mascagni@fsu.edu or mascagni@math.ethz.ch
or mascagni@nist.gov
URL: http://www.cs.fsu.edu/∼mascagni

Research supported by ARO, DOE, NASA, NATO, NIST, and NSF
with equipment donated by Intel and Nvidia

HPCS Plenary Talk: July 25, 2014

Outline of the Talk

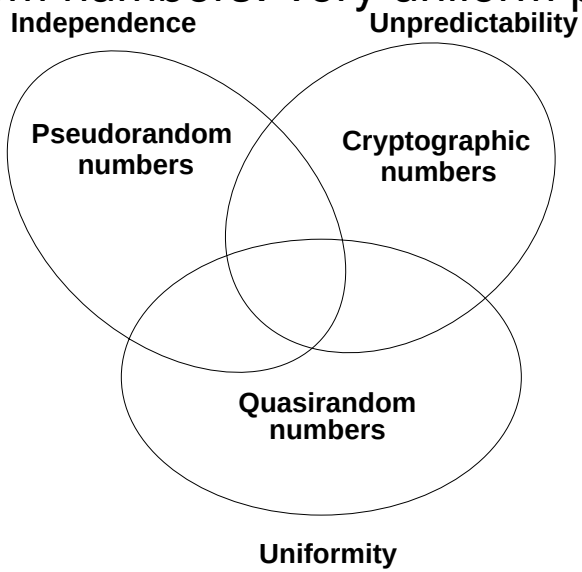
Monte Carlo Methods: Numerical Experimental that Use Random Numbers

- A Monte Carlo method is any process that consumes random numbers

1. Each calculation is a numerical experiment
   - Subject to known and unknown sources of error
   - Should be reproducible by peers
   - Should be easy to run anew with results that can be combined to reduce the variance
2. Sources of errors must be controllable/isolatable
   - Programming/science errors under your control
   - Make possible RNG errors approachable
3. Reproducibility
   - Must be able to rerun a calculation with the same numbers
   - Across different machines (modulo arithmetic issues)
   - Parallel and distributed computers?

What are Random Numbers Used For?

- There are many types of random numbers
  1. “Real" random numbers: a mathematical idealization
  2. Random numbers based on a “physical source” of randomness
  3. Computational Random numbers
  1. Pseudorandom numbers: deterministic sequence that passes tests of randomness
  2. Cryptographic numbers: totally unpredictable
  3. Quasirandom numbers: very uniform points



Parallelization of RNGs

- Because pseudorandom number are generated deterministically and have a finite state (seed), random number streams are periodic
- With different utilization of cycle structure, the parallelization of RNGs permits two general approaches
  1. With a single full-period cycle one has to use cycle splitting
  2. With multiple full-period cycles one can parameterize a generator
     - 2.1 One can parameterize the seed: lagged-Fibonacci generators
     - 2.2 One can vary a parameter in the generator's defining relation: the multiplier in a linear congruential generator (not discussed further)
- A new approach, which has been extensively used with quasirandom numbers is to use many different scramblers of a single pseudorandom stream

Possible Parametrization

- Most of the RNGs have multiple cycles (not necessarily all of them being the same size)
- There are two main choices for how to parameterize
  1. Use different types of generators for each computational process
  2. Use same generators with different sets of parameters for each computational process
- Finding good sets of parameters is the most important issue in parametrization
- Parameterized sequences are difficult to re-serialize

Cycle Splitting

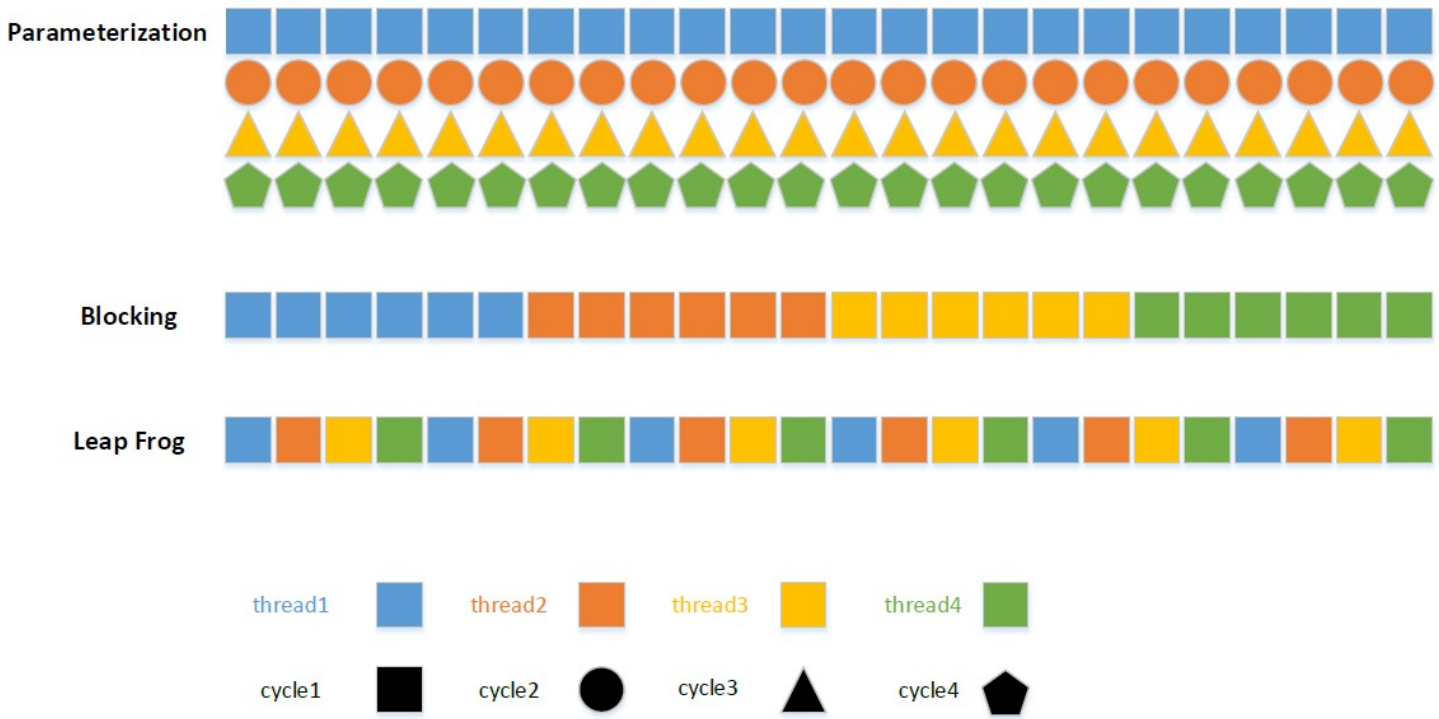
- Reasons why you cannot use parameterization
  1. Not every generator has multiple full-period cycles
  2. Some applications need (or are used to getting) random numbers from the same cycle
- Cycle splitting distributed numbers from a single cycle via substreams to different computational process
- One uses the same generator and same parameter for each computational process but with a different starting point or decimation

Cycle Splitting

- There are generally two ways of sharing one threads across several computational processes
- Blocking: Starting points are spaced far apart, using different sections of the entire cycle
- If a RNG sequence is given by $x_0, x_1, x_2 \ldots$, and the block size is B: a blocking scheme assigns the ith computational process to $x_{iB}, x_{iB+1}, x_{iB+2}, \ldots$
- Leap-frogging: the numbers in the computational process have the same interval in the original cycle
- If the total number of computational processes is t, the leap-frogging scheme will assign the ith computational process to $x_i, x_{i+t}, x_{i+2t}, \ldots$

Cycle Splitting Illustration

Parallelization of one level



Cycle Splitting

- The major concern for blocking is overlapping between substreams
- Leap-frogging is decimation and the numbers produced actually satisfy a different recurrence
- Both need fast leap-ahead algorithms allow efficient implementation
  1. A fast leap-ahead algorithm allows leaping n steps forward with $O(\log_2 n)$ work compared to the RNGs single step
  2. Fast leap-ahead algorithms vary from generator to generator

Parallel Neutronics: A Difficult Example

1. The structure of parallel neutronics
   - Use a parallel queue to hold unfinished work
   - Each processor follows a distinct neutron
   - Fission event places a new neutron(s) in queue with initial conditions
2. Problems and solutions
   - Reproducibility: each neutron is queued with a new generator (and with the next generator)
   - Using the binary tree mapping prevents generator reuse, even with extensive branching
   - A global seed reorders the generators to obtain a statistically significant new but reproducible result

Many Parameterized Streams Facilitate Implementation/Use

1. Advantages of using parameterized generators
   - Each unique parameter value gives an “independent” stream
   - Each stream is uniquely numbered
   - Numbering allows for absolute reproducibility, even with MIMD queuing
   - Effective serial implementation + enumeration yield a portable scalable implementation
   - Provides theoretical testing basis

Many Parameterized Streams Facilitate Implementation/Use

2. Implementation details
   - Generators mapped canonically to a binary tree
   - Extended seed data structure contains current seed and next generator
   - Spawning uses new next generator as starting point: assures no reuse of generators
3. All these ideas in the **S**calable **P**arallel **R**andom **N**umber **G**enerators (SPRNG) library: `http://sprng.org`

Many Different Generators and A Unified Interface

1. Advantages of having more than one generator
   - An application exists that stumbles on a given generator
   - Generators based on different recursions allow comparison to rule out spurious results
   - Makes the generators real experimental tools
2. Two interfaces to the `SPRNG` library: simple and default
   - Initialization returns a pointer to the generator state: `init_SPRNG()`
   - Single call for new random number: `SPRNG()`
   - Generator type chosen with parameters in `init_SPRNG()`
   - Makes changing generator very easy
   - Can use more than one generator *type* in code
   - Parallel structure is extensible to new generators through dummy routines

New Directions for SPRNG

- SPRNG was originally designed for distributed-memory multiprocessors
- HPC architectures are increasingly based on commodity chips with architectural variations
  1. Microprocessors with more than one processor core (multicore)
  2. The IBM cell processor (not very successful even though it was in the Sony Playstation)
  3. Microprocessors with accelerators, most popular being GPUs (video games)
  4. Intel Phi
- We will consider only two of these:
  1. Multicore support using OpenMP
  2. GPU support using CUDA (Nvidia) and/or OpenCL (standard)
  3. Eventually OpenACC for GPU support?

New Directions for SPRNG

- SPRNG currently uses independent full-period cycles for each processor
  1. Organizes the independent use of generators without communication
  2. Permits reproducibility
  3. Initialization of new full-period generators is slow for some generators
- A possible solution
  1. Keep the independent full-period cycles for “top-level" generators
  2. Within these (multicore processor/GPU) use cycle splitting to service threads
  3. Precomputed parameter sets to improve initialization and branching performance

SPRNG for Multicore

- Wide-spread availability of microprocessors that have multiple processor cores
  1. This is true for all the major laptop/desktop microprocessors
  2. It is also true for Arm-based microprocessors, and most mobile processors
- Monte Carlo methods are highly suitable for efficient parallel execution (did you attend my tutorial?)
- It is important for enabling Monte Carlo on multicore architectures that we have RNG support as well
- We have chosen to do this by creating OpenMP-SPRNG to support using SPRNG via the ubiquitous OpenMP paradigm
- OpenMP is implemented through compiler directives and is support by most C/C++ and FORTRAN compilers

SPRNG is Already Thread Safe

- SPRNG code was originally designed to be thread safe
- This originally permitted multi-threaded implementations using such tools as pthreads
- Common solutions using pthreads synchronization – critical section, monitors, locking, barriers, ···
- SPRNG should not use those techniques, as synchronization harms performance and scalability
- Our solution is complete separation: one thread gets one substream, i.e. the data resource of each substream is not shared by threads
- Substreams identify threads using the OpenMP Function: omp_get_thread_num

Class Structure

- Here is the class stucture that was imposed on SPRNG as a results of OpenMP-SPRNG

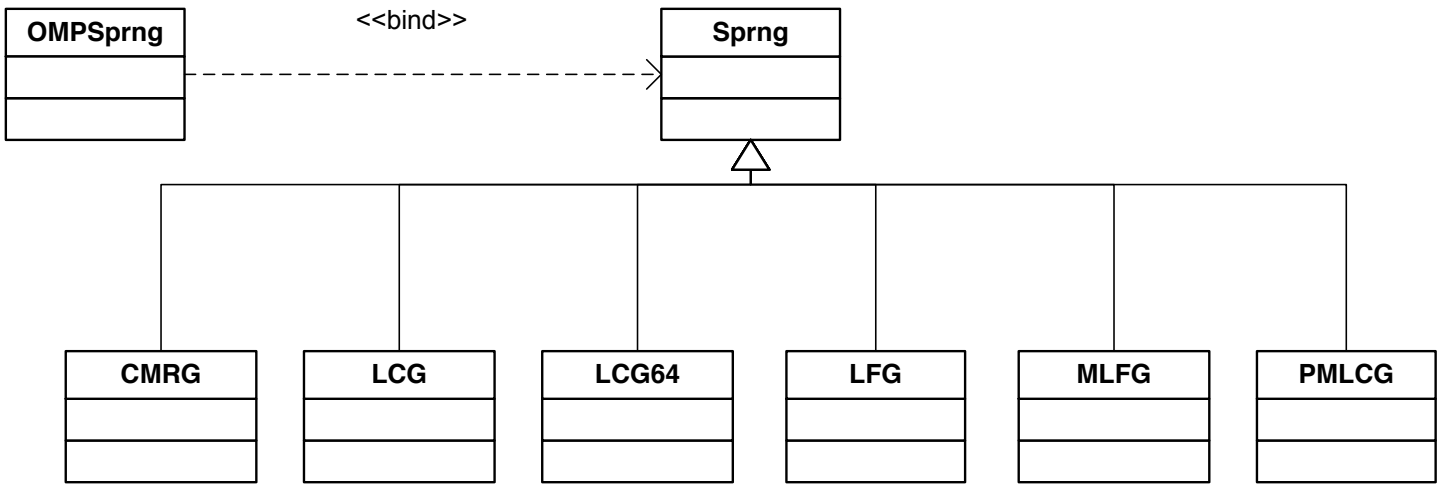


Figure: Static Class Structure

Delegation Sequence

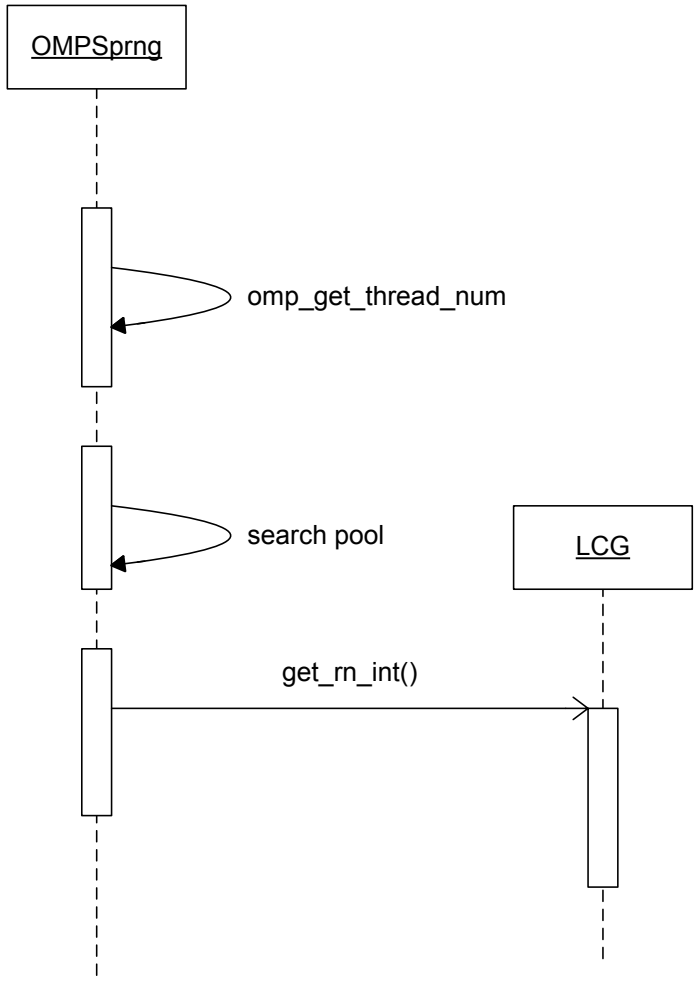


Figure: Delegation Sequence

Cycle Splitting in SPRNG for OpenMP

- Cycle splitting capability is added for SPRNG to support OpenMP
- Provides parallelization for applications with specific substream requirements
- Cycle splitting schemes were added to parallel and regular random number generator classes
- Both blocking and leap-frogging are supported

Time For Generating 2.048×10^9 Random Numbers

CPU	*Memory*	*Compiler*	*OS*
Intel(R) Xeon(R) E7340 2.4GHz	128GB	g++ 4.1.2	Linux 2.6.18-194.11.1.e15

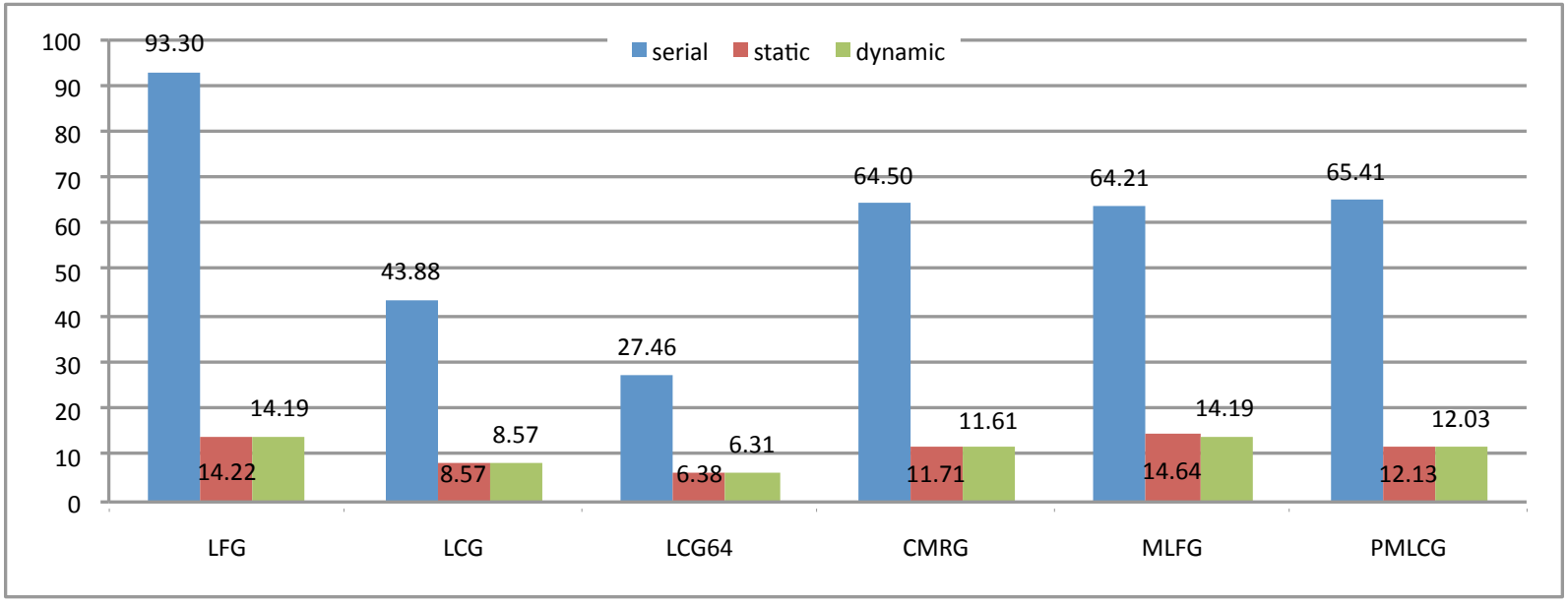


Figure: Time for generating random numbers, 8 threads

Relative Error of Integrating Gaussian Function By LCGs

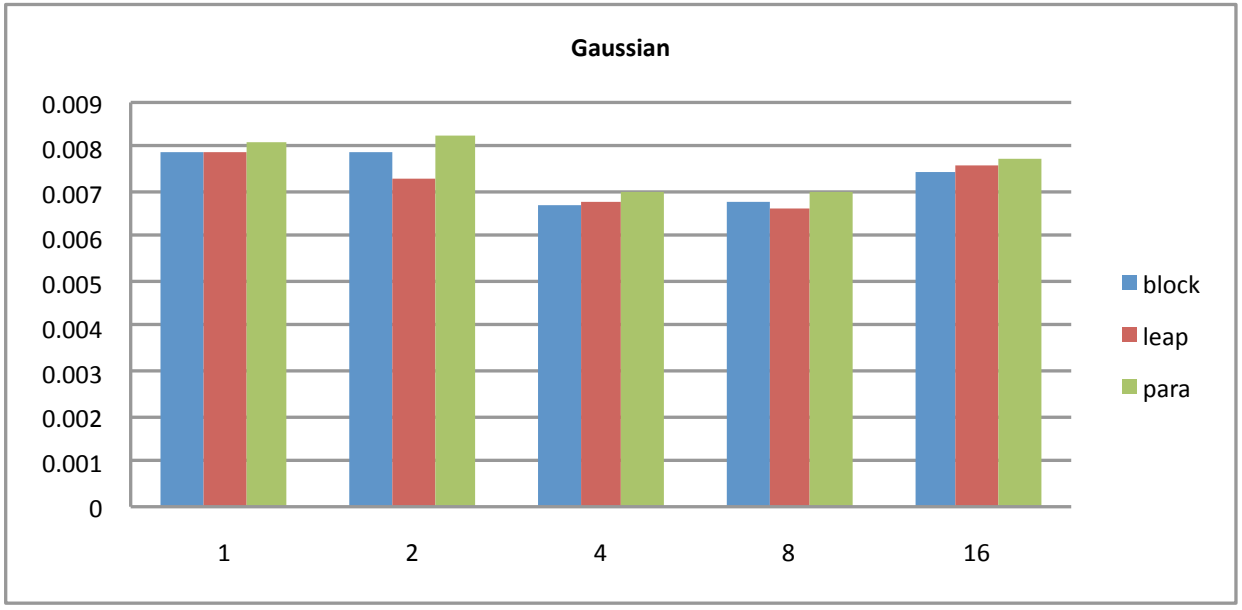


Figure: 20 Dimensions, with different threads

Reproducibility

Schedule	Thread Safe	Reproducibility
static	Yes	Yes
dynamic	Yes	No
guided	Yes	No
runtime	Yes	Maybe
auto	Yes	No

The CUDA Programming Model

- There are currently three paradigms that support Graphical Processing Unit (GPU) programming
  1. CUDA (Compute Unified Device Architecture) is a parallel programming model created by NVIDIA
  2. OpenCL is a standard to support accelerator programming that was developed by a consortium (platform agnostic)
  3. OpenACC is, like OpenMP, a compiler directive-based approach and will become part of OpenMP 4.0
- We model the CPU as host and the GPU as device
- Three level of parallelism: grid, block, and thread
- Different memory size and usage: global memory, shared memory, local memory, and texture memory
- Functions are called by using: func«<#blocks, #threads»>(), blocks and threads are used
- The GPU can be thought of as a Single Instruction Multiple Data (SIMD) processor

Lagged-Fibonacci Generators for GPUs

- General form:

$$x_n = x_{n-k} \square x_{n-\ell} \pmod{2^b}, \quad \ell > k$$

- Three types of Lagged-Fibonacci Generators (LFGs) are often defined, but we are interested only in the last two
  1. $\square = \oplus$: These are an efficient implementation of b bitwise shift-register sequences stacked one upon the other
  2. $\square = +$: Additive LFGs (ALFGs) are simple, easy to implement, efficient, and are linear generalizations of shift-register sequences
  3. $\square = \times$: Multiplicative LFGs (MLFGs) are not linear generators, thus they have better statistical properties
- ALFGs and MLFGs are easy to parameterize as we will see

The Cycle Structure of LFGs

- LFGs have multiple maximal-period cycles
- With proper initialization, the generator can be put all possible $(2^{(\ell-1)(b-1)})$ maximal-period cycles

m.s.b				l.s.b.	
b_{b-1}	b_{b-2}	...	b_1	b_0	
□	□	...	0	0	x_{l-1}
0	□	...	□	0	x_{l-2}
⋮	⋮	⋮	⋮	⋮	⋮
□	0	...	□	0	x_1
□	□	...	□	1	x_0

Conversions: ALFG $\Longleftrightarrow$ MLFG

- If one MLFG x_i is initially even, eventually all the outputs will be even, so we use only odds
- Odd integers modulo a power-of-two can be represented as

$$x_n = \pm 1 \times 3^{\text{odd}} \pmod{2^b}$$

- From ALFG to MLFG we can use this fact directly

$$y_n = 3^{x_{n-k}} \times 3^{x_{n-\ell}} = 3^{x_{n-k}+x_{n-\ell}}$$

- From MLFG to ALFG we have to use discrete logarithm modulo a power-of-two:

$$x_n = (-1)^{y_n} 3^{z_n} \pmod{2^b}$$

where y_n and z_n are:

$$y_n = y_{n-k} + y_{n-\ell} \pmod 2$$
$$z_n = z_{n-k} + z_{n-\ell} \pmod{2^{b-2}}$$

The Special Structure of LFGs

- Not all numbers are used in the LFG state array
- If we closely examine the equation:

$$x_n = x_{n-k} \square x_{n-\ell} \pmod{2^b}, \quad \ell > k$$

- k, the small gap, numbers can be generated simultaneously
- This allows for efficient k-fold vectorization for the SIMD GPU architecture
- This has not been previously utilized to implement vectorization for LFG generation

Fast Leap-Ahead Algorithms for LFGs to Support Cycle Splitting

- Can use polynomial arithmetic with fast multiply-square algorithm for computing a monomial modulo a polynomial
- $X_n = A^n X_0 \mod 2^b$ with fast multiply-square algorithm

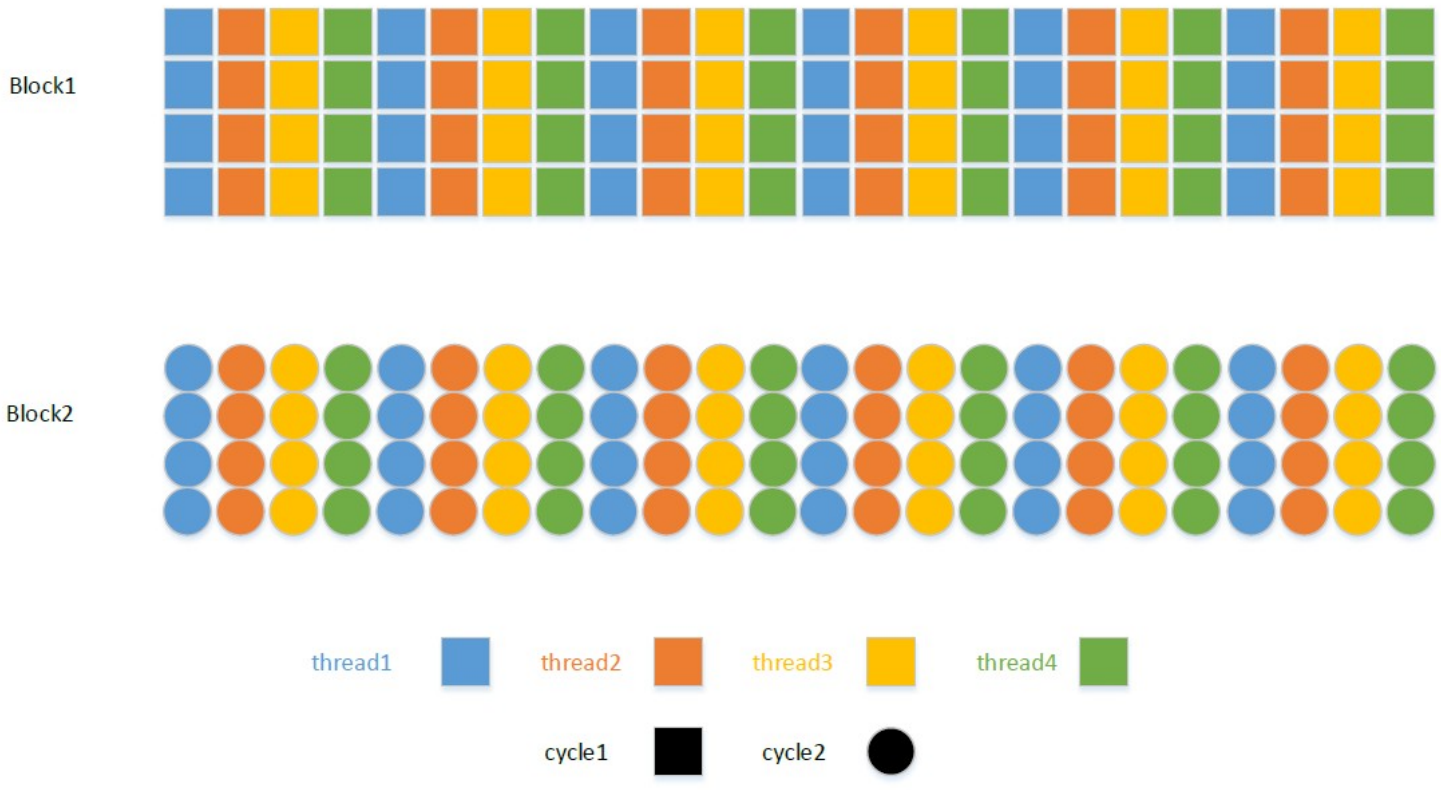
$$A = \begin{pmatrix} 0 & 0 & 0 & \dots & 0 & 1 & 0 & \dots & 0 & 0 & 1 \\ 1 & 0 & 0 & \dots & 0 & 0 & 0 & \dots & 0 & 0 & 0 \\ 0 & 1 & 0 & \dots & 0 & 0 & 0 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 0 & 0 & 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & 0 & \dots & 1 & 0 & 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & 0 & \dots & 0 & 1 & 0 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 0 & 0 & 0 & \dots & 1 & 0 & 0 \\ 0 & 0 & 0 & \dots & 0 & 0 & 0 & \dots & 0 & 1 & 0 \end{pmatrix}$$

Parallelization Schemes

- Less than the theoretical maximum number of schemes of 5 are used because of memory restrictions
- For host APIs, three schemes are used: pure parameterization, parameterization + leap-frogging, and parameterization + original order
- For device APIs, two schemes are used: pure parameterization, parameterization + leap-frogging
- Pure parameterization uses an independent cycle for each thread

Parallelization Diagram

Parallelization Using Two Levels: Parameterization + Leap-Frogging

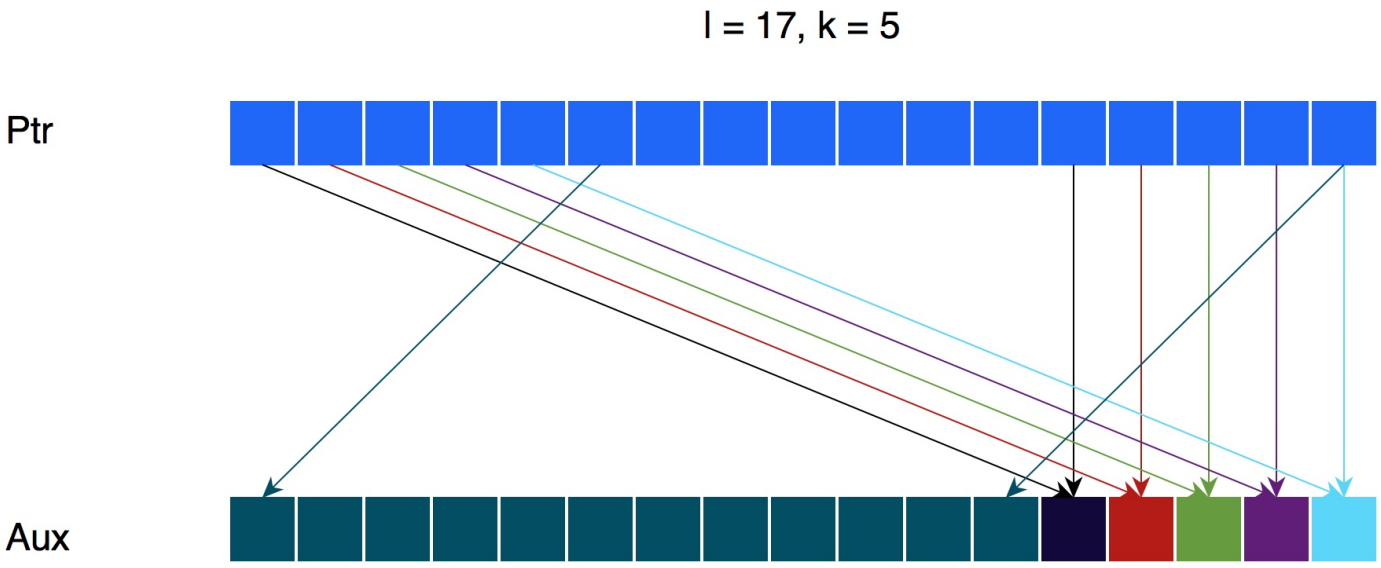


Why Blocking is Not Used

- Blocking is a compromised version of the original order
- Blocking suffers from overlapping and inability of adding new threads, with a fixed block size
- Blocking also needs separate state arrays for each thread, which is gives it no advantage over parameterization in memory usage

Design of LFGs

The implementation of LFGs for GPU



The Host API

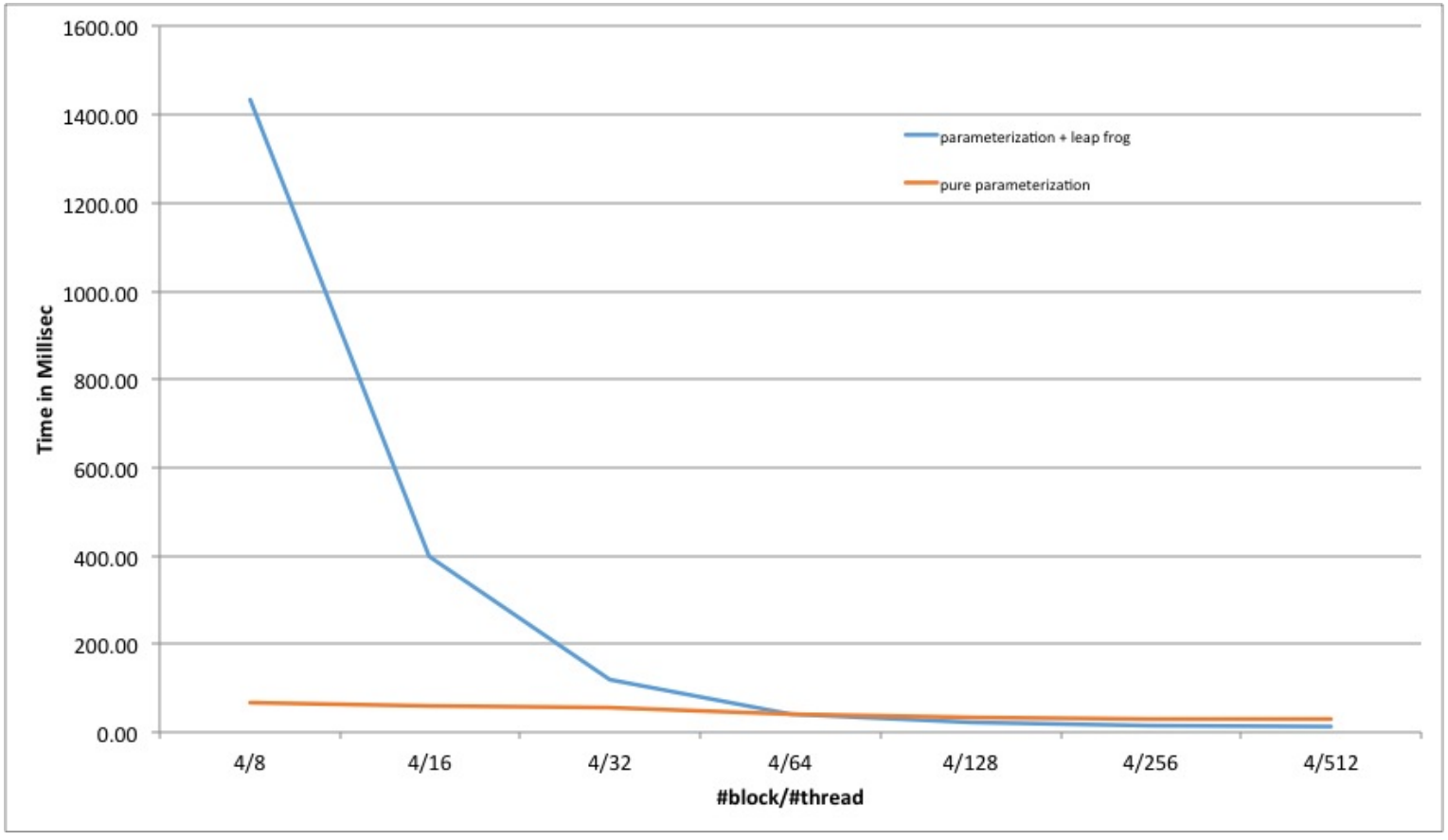
- Generate a large block of numbers and leave it in global memory
- `alfg_leap()`, `alfg_original()`, and `alfg_para()` are functions for different schemes of ALFGs
- `mlfg_leap()`, `mlfg_original()`, and `mlfg_para()` are functions for different schemes of MLFGs
- Initialization is done within each function

The Device API

- Initialization function is decoupled from the generation function
- `lfg_init()` initializes state arrays for different blocks
- `lfg_update()` calculates new seeds and stores them in the aux array
- `lfg_arrayswitch()` switches the state array and aux array
- `lfg()` read seeds
- No dirty reads are possible
- This is very memory efficient compared to all CURAND (Nvidia CUda RANDom) generators
- The generator state can be as small as 34 64-bit integers, compared to 1024 for MTGP32 from CURAND

Scalability

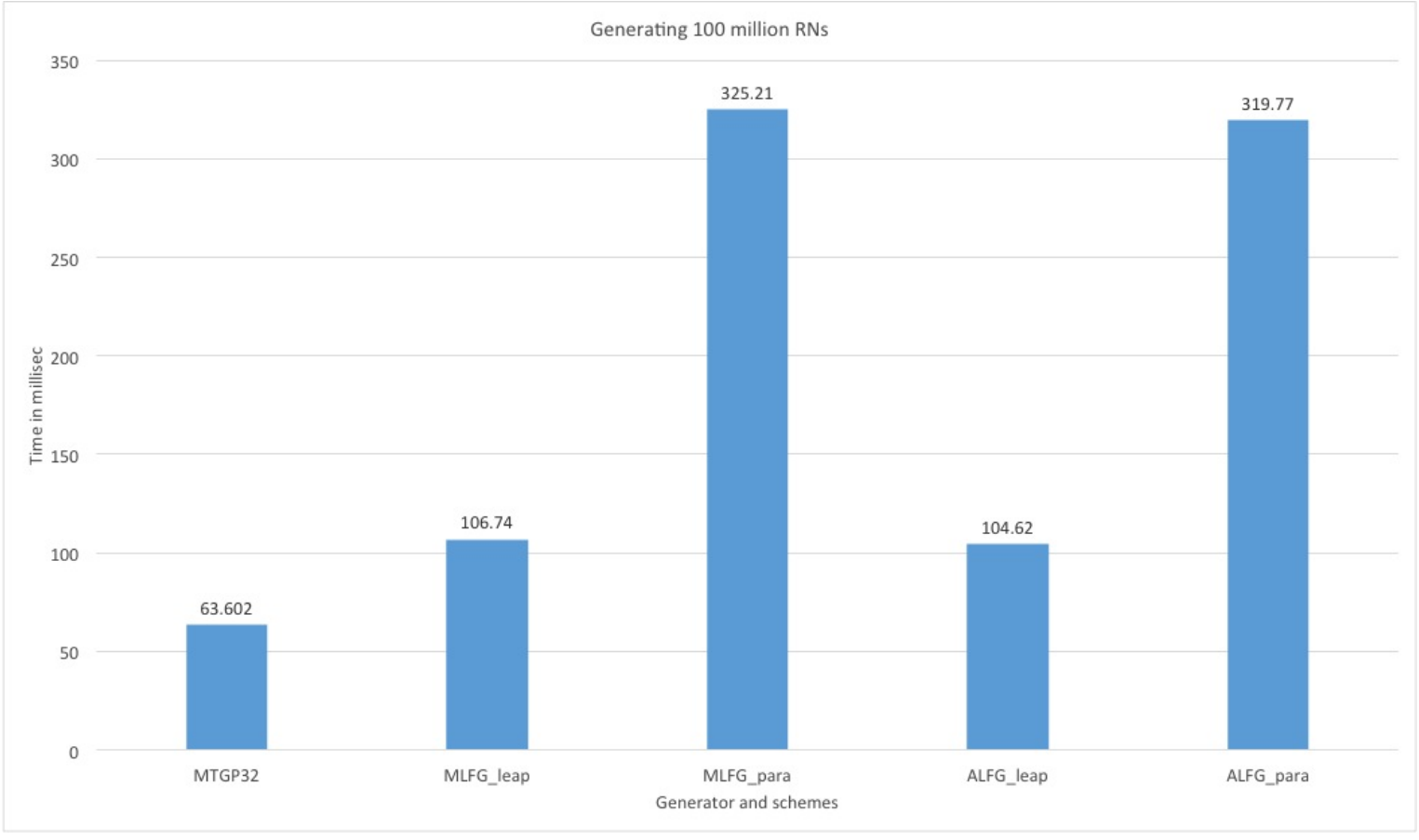
The Scalability of ALFGs and MLFGs Are Similar



Speed

- The speed of ALFGs and MLFGs are similar using 16 blocks and 256 threads
- MTGP is faster because it uses precomputed parameters, therefore, the costly initialization stage is bypassed
- Precomputed parameter sets will be implemented in the next stage for the LFGs for SPRNG

The Speed of Computing Random Numbers



ALFG with top 32-bit

```
========= Summary results of BigCrush =========

 Version:          TestU01 1.2.3
 Generator:        alfgtop32
 Number of statistics:  160
 Total CPU time:   04:37:02.61
 The following tests gave p-values outside [0.001, 0.9990]:
 (eps  means a value < 1.0e-300):
 (eps1 means a value < 1.0e-15):

       Test                          p-value
 ----------------------------------------------
 36  Gap, r = 0                        eps
 37  Gap, r = 20                       eps
 ----------------------------------------------
 All other tests were passed
```

Statistical Results for LFGs

ALFG with the low 32-bits

```
========= Summary results of BigCrush =========

 Version:          TestU01 1.2.3
 Generator:        alfglow32
 Number of statistics:  160
 Total CPU time:   04:36:51.86
 The following tests gave p-values outside [0.001, 0.9990]:
 (eps  means a value < 1.0e-300):
 (eps1 means a value < 1.0e-15):

       Test                          p-value
 ----------------------------------------------
 12  CollisionOver, t = 21        1 -  5.8e-7
 36  Gap, r = 0                      eps
 37  Gap, r = 20                     eps
 ----------------------------------------------
 All other tests were passed
```

Statistical Results for LFGs

ALFG with the top 32-bits with decimation by 8

```
========= Summary results of BigCrush =========

 Version:          TestU01 1.2.3
 Generator:        alfgtop32_dec8
 Number of statistics:  160
 Total CPU time:   09:42:21.15

 All tests were passed
```

Statistical Results for LFGs

MLFG with the top 32-bits

```
========= Summary results of BigCrush =========

 Version:          TestU01 1.2.3
 Generator:        mlfgtop32
 Number of statistics:  160
 Total CPU time:   06:21:20.06
 The following tests gave p-values outside [0.001, 0.9990]:
 (eps  means a value < 1.0e-300):
 (eps1 means a value < 1.0e-15):

       Test                          p-value
 ----------------------------------------------
 73  GCD                             2.5e-4
 ----------------------------------------------
 All other tests were passed
```

The Reason Not to Rely on Statistical Tests

MLFG with the top 32-bits plus 1

```
========= Summary results of BigCrush =========

 Version:          TestU01 1.2.3
 Generator:        mlfgtop32plus1
 Number of statistics:  160
 Total CPU time:   06:24:31.70

 All tests were passed
```

Experience with SPRNG for Multicore via OpenMP

- We have implemented an OpenMP version of SPRNG for multicore using these ideas
- OpenMP is now built into the main compilers, so it is easy to access
- Our experience has been
  1. Works as expected giving one access to high-quality, fast, and well tested RNGs on all the cores
  2. Permits forensic reproducibility: must know the number of threads that were used
  3. Near perfect parallelization is expected and seen
  4. Performance comparison with independent spawning vs. cycle splitting is not as dramatic as expected
- Backward (forensic) reproducibility is something that we can provide, but forward reproducibility is trickier
- This is work with Drs. Haohai Yu (Microsoft) and Dr. Yue Qiu

Experience with SPRNG on GPUs via CUDA

- SPRNG for the GPU is simple in principal, but harder for users to effectively structure efficient code
  1. The same technique that was used for multicore work for GPUs with many of the same issues
  2. The concept of reproducibility has to be modified as well
  3. Successful exploitation of GPU threads requires that SPRNG calls be made to insure that the data and the execution are on the GPU, in our case the “Device API"
- Software development may not be the hardest aspect of this work for a complete version of SPRNG for the GPU
  1. Clear documentation with descriptions of common coding errors will be essential for success
  2. An extensive collection of examples will be necessary to provide most users with code close to their own to help them use the GPU efficiently for Monte Carlo
- This is work with Drs. Timothy Andersen (Daniel H. Wagner Associates) and Dr. Yue Qiu

Future SPRNG Work

- SPRNG for use on Intel Phi
- An MS student, Rakesh Rajappa is doing some work on the Phi
- Creating a clean version of the SPRNG v4.4 code is taking place this summer with doctoral student, John Thrasher
- Making the SPRNG website more user friendly
- A comprehensive version of SPRNG for modern HPC architectures
  1. A free version for academic/government/nonprofit users
  2. A licensed version to fund long-term support from for profit users

Bibliography

- [M. Mascagni, T. Anderson, H. Yu and Y. Qiu (2014)] Papers on SPRNG for Multicore and GPU 1 submitted, 3 in preparation
- [H. Chi, M. Mascagni and T. Warnock (2005)] On the Optimal Halton Sequence, *Mathematics and Computers in Simulation*, **70(1)**: 9–21.
- [M. Mascagni and H. Chi (2004)] Parallel Linear Congruential Generators with Sophie-Germain Moduli, *Parallel Computing*, **30**: 1217–1231.
- [M. Mascagni and A. Srinivasan (2004)] Parameterizing Parallel Multiplicative Lagged-Fibonacci Generators, *Parallel Computing*, **30**: 899–916.
- [M. Mascagni and A. Srinivasan (2000)] Algorithm 806: SPRNG: A Scalable Library for Pseudorandom Number Generation, *ACM Transactions on Mathematical Software*, **26**: 436–461.