Computational Physics with GPUs
Lecture 4: Simulating spin models II

Martin Weigel

Applied Mathematics Research Centre, Coventry University, Coventry, United Kingdom and
Institut für Physik, Johannes Gutenberg-Universität Mainz, Germany

41st Heidelberg Physics Graduate Days
Heidelberg, October 8–12, 2018
Slides and exercises

Check out the lecture notes and example code at

http://users.complexity-coventry.org/~weigel/GPU/

Any questions? Contact me at

Martin.Weigel@mail.com
Outline

1. Random number generators
2. Local updates (again)
3. Generalized ensembles
Random number generators

RNG: definition

Stochastic simulations such as Monte Carlo and molecular dynamics (with a thermostat) require a reliable stream of “randomness”.
RNG: definition

Stochastic simulations such as Monte Carlo and molecular dynamics (with a thermostat) require a reliable stream of “randomness”.

Approaches:

- true randomness from, e.g., fluctuations in a resistor: too slow
RNG: definition

Stochastic simulations such as Monte Carlo and molecular dynamics (with a thermostat) require a reliable stream of “randomness”.

Approaches:

- true randomness from, e.g., fluctuations in a resistor: too slow
- pseudorandom number generator: deterministic sequence of (typically integer) numbers with the following properties
RNG: definition

Stochastic simulations such as Monte Carlo and molecular dynamics (with a thermostat) require a reliable stream of “randomness”.

Approaches:

- true randomness from, e.g., fluctuations in a resistor: too slow
- pseudorandom number generator: deterministic sequence of (typically integer) numbers with the following properties
  - based on a state vector
  - with a finite period
  - reproducible if using the same seed
  - typically produce uniform distribution on $[0, \text{NMAX}]$ or $[0, 1]$
  - further distributions (such as Gaussian) generated from transformations
RNG: definition

Stochastic simulations such as Monte Carlo and molecular dynamics (with a thermostat) require a reliable stream of “randomness”.

Approaches:

- **true randomness** from, e.g., fluctuations in a resistor: too slow
- **pseudorandom** number generator: deterministic sequence of (typically integer) numbers with the following properties
  - based on a **state** vector
  - with a finite **period**
  - **reproducible** if using the same **seed**
  - typically produce **uniform** distribution on \([0, \text{NMAX}]\) or \([0, 1]\)
  - further distributions (such as Gaussian) generated from transformations

- generally two types of pseudo RNGs considered
  - for **general purposes**, including simulations
  - or for **cryptographic purposes**, requiring sufficient randomness to prevent efficient stochastic inference
The story of R250

John von Neumann

“Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.” (1951)
John von Neumann

“Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.” (1951)

For any pseudo RNG (or RNG, for short) there must exist an algorithm/test that distinguishes the generated sequence from a truly random sequence.
The story of R250

John von Neumann

“Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.” (1951)

For any pseudo RNG (or RNG, for short) there must exist an algorithm/test that distinguishes the generated sequence from a truly random sequence. (If nothing else, this can be the algorithm generating the sequence itself!)
The story of R250

John von Neumann

“Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.” (1951)

For any pseudo RNG (or RNG, for short) there must exist an algorithm/test that distinguishes the generated sequence from a truly random sequence. (If nothing else, this can be the algorithm generating the sequence itself!)

Monte Carlo Simulations: Hidden Errors from “Good” Random Number Generators

Alan M. Ferrenberg and D. P. Landau
Center for Simulational Physics, The University of Georgia, Athens, Georgia 30602

Y. Joanna Wong
IBM Corporation, Supercomputing Systems, Kingston, New York 12401
(Received 29 July 1992)

The Wolff algorithm is now accepted as the best cluster-flipping Monte Carlo algorithm for beating “critical slowing down.” We show how this method can yield incorrect answers due to subtle correlations in “high quality” random number generators.

PACS numbers: 75.40.Mg, 05.70.Kk, 64.60.Fr

The explosive growth in the use of Monte Carlo simulations in diverse areas of physics has prompted extensive investigation of new methods and of the reliability of both old and new techniques. Monte Carlo simulations are
Coin-Tossing Computers Found to Show Subtle Bias

By MALCOLM W. BROWN
Published: January 12, 1993

WHEN scientists use computers to try to predict complex trends and events, they often apply a type of calculation that requires long series of random numbers. But instructing a computer to produce acceptably random strings of digits is proving maddeningly difficult.

In deciding which team kicks off a football game, the toss of a real coin is random enough to satisfy all concerned. But the cost of even a slightly nonrandom string of electronic coin tosses can be devastating to both practical problem-solving and pure theory, and a new investigation has revealed that nonrandom computer tosses are much more common than many scientists had assumed.

Mathematical "models" designed to predict stock prices, atmospheric warming, airplane skin friction, chemical reactions, epidemics, population growth, the outcome of battles, the locations of oil deposits and hundreds of other complex matters increasingly depend on a statistical technique called Monte Carlo Simulation, which in turn depends on reliable and inexhaustible sources of random numbers.

Monte Carlo Simulation, named for Monaco's famous gambling casino, can help to represent very complex interactions in physics, chemistry, engineering, economics and environmental dynamics mathematically. Mathematicians call such a representation a "model," and if a model is accurate enough, it produces the same responses to manipulations that the real thing would do. But Monte Carlo modeling contains a dangerous flaw: if the supposedly random numbers that must be pumped into a simulation actually form some subtle, nonrandom pattern, the entire simulation (and its predictions) may be wrong.
Random number testing

A sequence $u_i$ of pseudo-random numbers is perfect iff all sequences $(u_0, \ldots, u_{t-1})$ are uniformly distributed over $[0, 1]^t$ for arbitrary $t$. Clearly, this cannot be the case, already because of the finite period.

Derived statistical tests:
- Test for uniformity
- Correlation tests
- Comparison to combinatorial identities
- Comparison to other known statistical results
- Application tests (e.g., Ising model)

On the other hand, there are cryptographic tests based on the lack of predictability.

No RNG can pass every conceivable test, so a bad RNG is one that fails simple tests, and a good RNG is one that only fails only very complicated tests.

Test batteries:
- DieHard (1995) by G. Marsaglia, now outdated
- TestU01 (2002/2009) by P. L’Ecuyer and co-workers, quasi standard
Random number testing

A sequence $u_i$ of pseudo-random numbers is perfect iff all sequences $(u_0, \ldots, u_{t-1})$ are uniformly distributed over $[0, 1]^t$ for arbitrary $t$. Clearly, this cannot be the case, already because of the finite period.

- Derived statistical tests:
Random number testing

A sequence $u_i$ of pseudo-random numbers is perfect iff all sequences $(u_0, \ldots, u_{t-1})$ are uniformly distributed over $[0, 1]^t$ for arbitrary $t$. Clearly, this cannot be the case, already because of the finite period.

- Derived statistical tests:
  - test for uniformity
  - correlation tests
  - comparison to combinatorial identities
  - comparison to other known statistical results
  - application tests (e.g., Ising model)
Random number testing

A sequence $u_i$ of pseudo-random numbers is perfect iff all sequences $(u_0, \ldots, u_{t-1})$ are uniformly distributed over $[0, 1]^t$ for arbitrary $t$. Clearly, this cannot be the case, already because of the finite period.

- Derived statistical tests:
  - test for uniformity
  - correlation tests
  - comparison to combinatorial identities
  - comparison to other known statistical results
  - application tests (e.g., Ising model)

On the other hand, there are cryptographic tests based on the lack of predictability.
Random number testing

A sequence $u_i$ of pseudo-random numbers is perfect iff all sequences $(u_0, \ldots, u_{t-1})$ are uniformly distributed over $[0, 1]^t$ for arbitrary $t$. Clearly, this cannot be the case, already because of the finite period.

- Derived statistical tests:
  - test for uniformity
  - correlation tests
  - comparison to combinatorial identities
  - comparison to other known statistical results
  - application tests (e.g., Ising model)

On the other hand, there are cryptographic tests based on the lack of predictability.

No RNG can pass every conceivable test, so a bad RNG is one that fails simple tests, and a good RNG is one that only fails only very complicated tests.
Random number testing

A sequence $u_i$ of pseudo-random numbers is perfect iff all sequences $(u_0, \ldots, u_{t-1})$ are uniformly distributed over $[0, 1]^t$ for arbitrary $t$. Clearly, this cannot be the case, already because of the finite period.

- Derived statistical tests:
  - test for uniformity
  - correlation tests
  - comparison to combinatorial identities
  - comparison to other known statistical results
  - application tests (e.g., Ising model)

On the other hand, there are cryptographic tests based on the lack of predictability.

No RNG can pass every conceivable test, so a bad RNG is one that fails simple tests, and a good RNG is one that only fails only very complicated tests.

Test batteries:

- DieHard (1995) by G. Marsaglia, now outdated
- TestU01 (2002/2009) by P. L’Ecuyer and co-workers, quasi standard
Requirements for parallel computing

In applications such as Monte Carlo of lattice systems, we want to update many spins in parallel.
Requirements for parallel computing

In applications such as Monte Carlo of lattice systems, we want to update many spins in parallel. A single “RNG process” producing and handing out the numbers would be a severe bottleneck, impeding scaling.
Requirements for parallel computing

In applications such as Monte Carlo of lattice systems, we want to **update many spins in parallel**. A single “RNG process” producing and handing out the numbers would be a severe bottleneck, impeding scaling.

- hence, **each thread needs its own RNG** (potentially millions of them)
Requirements for parallel computing

In applications such as Monte Carlo of lattice systems, we want to update many spins in parallel. A single “RNG process” producing and handing out the numbers would be a severe bottleneck, impeding scaling.

- hence, each thread needs its own RNG (potentially millions of them)
- to minimize the pressure on the bus, on registers and shared memory, the RNG state needs to be as small as possible
Requirements for parallel computing

In applications such as Monte Carlo of lattice systems, we want to update many spins in parallel. A single “RNG process” producing and handing out the numbers would be a severe bottleneck, impeding scaling.

- hence, each thread needs its own RNG (potentially millions of them)
- to minimize the pressure on the bus, on registers and shared memory, the RNG state needs to be as small as possible
- the streams of all RNG instances must be sufficiently uncorrelated to yield reliable results together
Requirements for parallel computing

In applications such as Monte Carlo of lattice systems, we want to update many spins in parallel. A single “RNG process” producing and handing out the numbers would be a severe bottleneck, impeding scaling.

- hence, each thread needs its own RNG (potentially millions of them)
- to minimize the pressure on the bus, on registers and shared memory, the RNG state needs to be as small as possible
- the streams of all RNG instances must be sufficiently uncorrelated to yield reliable results together

This could be reached by

  (a) division of the stream of a long-period generator into non-overlapping sub-streams to be produced and consumed by the different threads of the application, or
Requirements for parallel computing

In applications such as Monte Carlo of lattice systems, we want to update many spins in parallel. A single “RNG process” producing and handing out the numbers would be a severe bottleneck, impeding scaling.

- hence, each thread needs its own RNG (potentially millions of them)
- to minimize the pressure on the bus, on registers and shared memory, the RNG state needs to be as small as possible
- the streams of all RNG instances must be sufficiently uncorrelated to yield reliable results together

This could be reached by

(a) division of the stream of a long-period generator into non-overlapping sub-streams to be produced and consumed by the different threads of the application, or

(b) use of very large period generators such that overlaps between the sequences of the different instances are improbable, if each instance is seeded differently, or
Requirements for parallel computing

In applications such as Monte Carlo of lattice systems, we want to update many spins in parallel. A single “RNG process” producing and handing out the numbers would be a severe bottleneck, impeding scaling.

- hence, each thread needs its own RNG (potentially millions of them)
- to minimize the pressure on the bus, on registers and shared memory, the RNG state needs to be as small as possible
- the streams of all RNG instances must be sufficiently uncorrelated to yield reliable results together

This could be reached by

(a) division of the stream of a long-period generator into non-overlapping sub-streams to be produced and consumed by the different threads of the application, or
(b) use of very large period generators such that overlaps between the sequences of the different instances are improbable, if each instance is seeded differently, or
(c) setup of independent generators of the same class of RNGs using different lags, multipliers, shifts etc.
Simplest choice satisfying these requirements is linear congruential generator (LCG):

\[ x_{n+1} = ax_n + c \pmod{m}. \]
Simplest choice satisfying these requirements is linear congruential generator (LCG):

\[ x_{n+1} = ax_n + c \pmod{m}. \]

- for \( m = 2^{32} \) or \( 2^{32} - 1 \), the maximal period is of the order \( p \approx m \approx 10^9 \), much too short for large-scale simulations.
Linear congruential generators

Simplest choice satisfying these requirements is linear congruential generator (LCG):

\[ x_{n+1} = ax_n + c \pmod{m}. \]

- for \( m = 2^{32} \) or \( 2^{32} - 1 \), the maximal period is of the order \( p \approx m \approx 10^9 \), much too short for large-scale simulations
- one should actually use at most \( \sqrt{p} \) numbers of the sequence
Simplest choice satisfying these requirements is linear congruential generator (LCG):

\[ x_{n+1} = ax_n + c \pmod{m}. \]

- for \( m = 2^{32} \) or \( 2^{32} - 1 \), the maximal period is of the order \( p \approx m \approx 10^9 \), much too short for large-scale simulations
- one should actually use at most \( \sqrt{p} \) numbers of the sequence
- for \( m = 2^{32} \), modulo can be implemented as overflow, but then period of lower rank bits is only \( 2^k \)
Simplest choice satisfying these requirements is linear congruential generator (LCG):

\[ x_{n+1} = a x_n + c \pmod{m}. \]

- for \( m = 2^{32} \) or \( 2^{32} - 1 \), the maximal period is of the order \( p \approx m \approx 10^9 \), much too short for large-scale simulations
- one should actually use at most \( \sqrt{p} \) numbers of the sequence
- for \( m = 2^{32} \), modulo can be implemented as overflow, but then period of lower rank bits is only \( 2^k \)
- has poor statistical properties, e.g., \( k \)-tuples of (normalized) numbers lie on hyper-planes
Simplest choice satisfying these requirements is linear congruential generator (LCG):

\[ x_{n+1} = a x_n + c \pmod{m}. \]

(Source: Wikipedia)
Simplest choice satisfying these requirements is linear congruential generator (LCG):

\[ x_{n+1} = ax_n + c \pmod{m}. \]

- for \( m = 2^{32} \) or \( 2^{32} - 1 \), the maximal period is of the order \( p \approx m \approx 10^9 \), much too short for large-scale simulations
- one should actually use at most \( \sqrt{p} \) numbers of the sequence
- for \( m = 2^{32} \), modulo can be implemented as overflow, but then period of lower rank bits is only \( 2^k \)
- has poor statistical properties, e.g., \( k \)-tuples of (normalized) numbers lie on hyper-planes
- state is just 4 bytes per thread
Simplest choice satisfying these requirements is linear congruential generator (LCG):

\[ x_{n+1} = ax_n + c \pmod{m}. \]

- for \( m = 2^{32} \) or \( 2^{32} - 1 \), the maximal period is of the order \( p \approx m \approx 10^9 \), much too short for large-scale simulations.
- one should actually use at most \( \sqrt{p} \) numbers of the sequence.
- for \( m = 2^{32} \), modulo can be implemented as overflow, but then period of lower rank bits is only \( 2^k \).
- has poor statistical properties, e.g., \( k \)-tuples of (normalized) numbers lie on hyper-planes.
- state is just 4 bytes per thread.
- can easily skip ahead via \( x_{n+t} = a_t x_n + c_t \) with

\[ a_t = a^t \pmod{m}, \quad c_t = \sum_{i=1}^{t} a^i c \pmod{m}. \]
Random number generators

Linear congruential generators

Simplest choice satisfying these requirements is linear congruential generator (LCG):

\[ x_{n+1} = ax_n + c \pmod{m}. \]

- for \( m = 2^{32} \) or \( 2^{32} - 1 \), the maximal period is of the order \( p \approx m \approx 10^9 \), much too short for large-scale simulations
- one should actually use at most \( \sqrt{p} \) numbers of the sequence
- for \( m = 2^{32} \), modulo can be implemented as overflow, but then period of lower rank bits is only \( 2^k \)
- has poor statistical properties, e.g., \( k \)-tuples of (normalized) numbers lie on hyper-planes
- state is just 4 bytes per thread
- can easily skip ahead via \( x_{n+t} = a_t x_n + c_t \) with

\[ a_t = a^t \pmod{m}, \quad c_t = \sum_{i=1}^{t} a^i c \pmod{m}. \]

- can be improved by choosing \( m = 2^{64} \) and truncation to 32 most significant bits, period \( p = m \approx 10^{18} \) and 8 bytes per thread
LCGs: implementation

The implementation is indeed very simple and can be performed in-line:

```
#define A32 1664525
#define C32 1013904223

unsigned int ran;
CONVERT( ran = A32 * ran + C32 );
```

The output function for converting from \([0, \text{INTMAX}]\) to \([0, 1]\) could be implemented in different ways:

```
#define MULT32 2.328306437080797 e-10
#define CONVERT(x) ( MULT32 *(( unsigned int )(x)))
```

```c
// #define CONVERT(x) _curand_uniform(x)  
// #define CONVERT(x) __fdividef( __uint2float_rz(x) ,( float )0x100000000 );
```
LCGs: implementation

The implementation is indeed very simple and can be performed in-line:

```
#define A32 1664525
#define C32 1013904223

unsigned int ran;
CONVERT(ran = A32*ran+C32);
```
The implementation is indeed very simple and can be performed in-line:

```
#define A32 1664525
#define C32 1013904223

unsigned int ran;
CONVERT(ran = A32*ran+C32);
```

The output function for converting from [0, INTMAX] to [0, 1] could be implemented in different ways:

```c
#define MULT32 2.328306437080797e-10f
#define CONVERT (x) (MULT32 *((unsigned int)(x)))
//# define CONVERT (x) _curand_uniform (x)
//# define CONVERT (x) __fdividef ( __uint2float_rz (x) ,(float)0 x100000000 );
```
The implementation is indeed very simple and can be performed in-line:

```c
#define A32 1664525
#define C32 1013904223

unsigned int ran;
CONVERT(ran = A32*ran+C32);
```

The output function for converting from \([0, \text{INTMAX}]\) to \([0, 1]\) could be implemented in different ways:

```c
#define MULT32 2.328306437080797e-10f

#define CONVERT(x) (MULT32*(((unsigned int)(x)))
//#define CONVERT(x) _curand_uniform(x)
//#define CONVERT(x) __fdividef(__uint2float_rz(x),(float)0x100000000);
```
LCG: performance

How well do they perform?

Characteristic zig-zag pattern due to commensurability (or not) of block number of with number of multiprocessors.

Peak performance at $58 \times 10^9$ (LCG32) and $46 \times 10^9$ (LCG64) random numbers per second, respectively.
Random number generators

LCG: performance

How well do they perform?

Characteristic zig-zag pattern due to *commensurability* (or not) of block number of with number of multiprocessors.
LCG: performance

How well do they perform?

Characteristic zig-zag pattern due to commensurability (or not) of block number of with number of multiprocessors.

Peak performance at $58 \times 10^9$ (LCG32) and $46 \times 10^9$ (LCG64) random numbers per second, respectively.
LCG: overall benchmarks

Use these LCG generators for the previously developed simulation code for the 2D Ising model.
Use these LCG generators for the previously developed simulation code for the 2D Ising model. Exact results are available for comparison.
LCG: overall benchmarks

Use these LCG generators for the previously developed simulation code for the 2D Ising model. Exact results are available for comparison. Test case of $1024 \times 1024$ system at $\beta = 0.4$, $10^7$ sweeps.

- checkerboard update uses random numbers in different way than sequential update
- linear congruential generators can skip ahead: “right” way uses non-overlapping sub-sequences
- “wrong” way uses sequences from random initial seeds, many of which must overlap
# RNG quality: Ising results

Table: Internal energy $e$ per spin and specific heat $C_V$ for a $1024 \times 1024$ Ising model with periodic boundary conditions at $\beta = 0.4$.

<table>
<thead>
<tr>
<th>method</th>
<th>$e$</th>
<th>$\Delta_{\text{rel}}$</th>
<th>$C_V$</th>
<th>$\Delta_{\text{rel}}$</th>
<th>$t_{\text{up}}^{k=1}$</th>
<th>$t_{\text{up}}^{k=100}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>exact</td>
<td>1.106079207</td>
<td>0</td>
<td>0.8616983594</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sequential update (CPU)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LCG32</td>
<td>1.1060788(15)</td>
<td>-0.26</td>
<td>0.83286(45)</td>
<td>-63.45</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LCG64</td>
<td>1.1060801(17)</td>
<td>0.49</td>
<td>0.86102(60)</td>
<td>-1.14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fibonacci, $r = 512$</td>
<td>1.1060789(17)</td>
<td>-0.18</td>
<td>0.86132(59)</td>
<td>-0.64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>checkerboard update (GPU)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LCG32</td>
<td>1.0944421(14)</td>
<td>-8259.05</td>
<td>0.80316(48)</td>
<td>-121.05</td>
<td>0.2221</td>
<td>0.0402</td>
</tr>
<tr>
<td>LCG32, random</td>
<td>1.1060775(18)</td>
<td>-0.97</td>
<td>0.86175(56)</td>
<td>0.09</td>
<td>0.2221</td>
<td>0.0402</td>
</tr>
<tr>
<td>LCG64</td>
<td>1.1061058(19)</td>
<td>13.72</td>
<td>0.86179(67)</td>
<td>0.14</td>
<td>0.2311</td>
<td>0.0471</td>
</tr>
<tr>
<td>LCG64, random</td>
<td>1.1060803(18)</td>
<td>0.62</td>
<td>0.86215(63)</td>
<td>0.71</td>
<td>0.2311</td>
<td>0.0471</td>
</tr>
<tr>
<td>MWC, same $a$</td>
<td>1.1060800(18)</td>
<td>0.45</td>
<td>0.86161(60)</td>
<td>-0.15</td>
<td>0.2293</td>
<td>0.0435</td>
</tr>
<tr>
<td>MWC, different $a$</td>
<td>1.1060797(18)</td>
<td>0.28</td>
<td>0.86168(62)</td>
<td>-0.03</td>
<td>0.2336</td>
<td>0.0438</td>
</tr>
<tr>
<td>Fibonacci, $r = 521$</td>
<td>1.1060890(15)</td>
<td>6.43</td>
<td>0.86099(66)</td>
<td>-1.09</td>
<td>0.2601</td>
<td>0.0661</td>
</tr>
<tr>
<td>Fibonacci, $r = 1279$</td>
<td>1.1060800(19)</td>
<td>0.40</td>
<td>0.86084(53)</td>
<td>-1.64</td>
<td>0.2904</td>
<td>0.0700</td>
</tr>
<tr>
<td>XORWOW (cuRAND)</td>
<td>1.1060654(15)</td>
<td>-9.13</td>
<td>0.86167(65)</td>
<td>0.04</td>
<td>0.7956</td>
<td>0.0576</td>
</tr>
<tr>
<td>XORShift/Weyl</td>
<td>1.1060788(18)</td>
<td>-0.23</td>
<td>0.86184(53)</td>
<td>0.27</td>
<td>0.2613</td>
<td>0.0721</td>
</tr>
<tr>
<td>Philox4x32_7</td>
<td>1.1060778(18)</td>
<td>-0.79</td>
<td>0.86109(65)</td>
<td>-0.93</td>
<td>0.2399</td>
<td>0.0523</td>
</tr>
<tr>
<td>Philox4x32_10</td>
<td>1.1060777(17)</td>
<td>-0.85</td>
<td>0.86188(61)</td>
<td>0.30</td>
<td>0.2577</td>
<td>0.0622</td>
</tr>
</tbody>
</table>
LCG: overall benchmarks

Use these LCG generators for the previously developed simulation code for the 2D Ising model. Exact results are available for comparison. Test case of $1024 \times 1024$ system at $\beta = 0.4$, $10^7$ sweeps.

- checkerboard update uses random numbers in different way than sequential update
- linear congruental generators can skip ahead: “right” way uses non-overlapping sub-sequences
- “wrong” way uses sequences from random initial seeds, many of which must overlap

TestU01 results:

- poor for LCG32
- acceptable for LCG64
RNG quality: TestU01 results

Table: The memory footprint is measured in bits per thread. For the TestU01 results, if (too many) failures in SmallCrush are found, Crush and BigCrush are not attempted; likewise with failures in Crush. The performance column shows the peak number of 32-bit uniform floating-point random numbers produced per second on a fully loaded GTX 480 device.

<table>
<thead>
<tr>
<th>generator</th>
<th>bits/thread</th>
<th>failures in TestU01</th>
<th>Ising test</th>
<th>perf. $\times 10^9$/s</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>SmallCrush</td>
<td>Crush</td>
<td>BigCrush</td>
</tr>
<tr>
<td>LCG32</td>
<td>32</td>
<td>12</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>LCG32, random</td>
<td>32</td>
<td>3</td>
<td>14</td>
<td>—</td>
</tr>
<tr>
<td>LCG64</td>
<td>64</td>
<td>None</td>
<td>6</td>
<td>—</td>
</tr>
<tr>
<td>LCG64, random</td>
<td>64</td>
<td>None</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>MWC</td>
<td>64 + 32</td>
<td>1</td>
<td>29</td>
<td>—</td>
</tr>
<tr>
<td>Fibonacci, $r = 521$</td>
<td>$\geq 80$</td>
<td>None</td>
<td>2</td>
<td>—</td>
</tr>
<tr>
<td>Fibonacci, $r = 1279$</td>
<td>$\geq 80$</td>
<td>None</td>
<td>(1)</td>
<td>2</td>
</tr>
<tr>
<td>XORWOW (cuRAND)</td>
<td>192</td>
<td>None</td>
<td>None</td>
<td>$1/3$</td>
</tr>
<tr>
<td>MTGP (cuRAND)</td>
<td>$\geq 44$</td>
<td>None</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>XORShift/Weyl</td>
<td>32</td>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Philox4x32_7</td>
<td>(128)</td>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Philox4x32_10</td>
<td>(128)</td>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
</tbody>
</table>
LCG: overall benchmarks

Use these LCG generators for the previously developed simulation code for the 2D Ising model. Exact results are available for comparison. Test case of $1024 \times 1024$ system at $\beta = 0.4$, $10^7$ sweeps.

- checkerboard update uses random numbers in different way than sequential update
- linear congruential generators can skip ahead: “right” way uses non-overlapping sub-sequences
- “wrong” way uses sequences from random initial seeds, many of which must overlap

TestU01 results:

- poor for LCG32
- acceptable for LCG64

General conclusion: fast, but not good enough
So linear congruential generators are not in general good enough. What are the other options?
Other generators

So linear congruential generators are not in general good enough. What are the other options?

- Multiply with carry: Marsaglia generator, part of (some versions of) cuRAND
Other generators

So linear congruential generators are not in general good enough. What are the other options?

- Multiply with carry: Marsaglia generator, part of (some versions of) cuRAND
- Lagged Fibonacci generators

\[ x_n = a_s x_{n-s} \otimes a_r x_{n-r} \pmod{m}, \]

Easily parallelized, good properties for large lags, but memory intensive.
Other generators

So linear congruential generators are not in general good enough. What are the other options?

- Multiply with carry: Marsaglia generator, part of (some versions of) cuRAND
- Lagged Fibonacci generators
  
  \[ x_n = a_s x_{n-s} \odot a_r x_{n-r} \pmod{m}, \]

  Easily parallelized, good properties for large lags, but memory intensive.
- Variants of Mersenne twister.
Other generators

So linear congruential generators are not in general good enough. What are the other options?

- Multiply with carry: Marsaglia generator, part of (some versions of) cuRAND
- Lagged Fibonacci generators
  \[x_n = a_s x_{n-s} \otimes a_r x_{n-r} \pmod{m},\]
  Easily parallelized, good properties for large lags, but memory intensive.
- Variants of Mersenne twister.
- XORShift generator using words of, e.g., 1024 bits: fast and excellent properties.
Other generators

So linear congruential generators are not in general good enough. What are the other options?

- Multiply with carry: Marsaglia generator, part of (some versions of) cuRAND
- Lagged Fibonacci generators

\[ x_n = a_s x_{n-s} \otimes a_r x_{n-r} \pmod{m}, \]

Easily parallelized, good properties for large lags, but memory intensive.

- Variants of Mersenne twister.
- XORShift generator using words of, e.g., 1024 bits: fast and excellent properties.
- “Cryptographic” generators: Philox and friends, very well suited and good properties, now part of cuRAND.
Random number generators

Other generators

So linear congruential generators are not in general good enough. What are the other options?

- Multiply with carry: Marsaglia generator, part of (some versions of) cuRAND
- Lagged Fibonacci generators

\[ x_n = a_s x_{n-s} \otimes a_r x_{n-r} \pmod{m}, \]

Easily parallelized, good properties for large lags, but memory intensive.

- Variants of Mersenne twister.
- XORShift generator using words of, e.g., 1024 bits: fast and excellent properties.
- “Cryptographic” generators: Philox and friends, very well suited and good properties, now part of cuRAND.

Comprehensive discussion in

Outline

1. Random number generators
2. Local updates (again)
3. Generalized ensembles
Consider Metropolis kernel for the 2D Ising model discussed before:

```c
__global__ void metro_checkerboard_three ( spin_t *s, int * ranvec , int offset )
{
    int n = blockDim.x* blockIdx .x + threadIdx .x;
    int cur = blockDim .x* blockIdx .x + threadIdx .x + offset *(N /2) ;
    int north = cur + (1 -2* offset )*(N /2) ;
    int east = (( north +1) %L) ? north + 1 : north -L +1;
    int west = ( north %L) ? north - 1 : north +L -1;
    int south = (n - (1 -2* offset )*L + N /2) %(N /2) + (1 - offset )*(N /2) ;
    int ide = s[ cur ]*( s[ west ]+s[ north ]+s[ east ]+s[ south ]);
    if( fabs ( RAN ( ranvec [n]) *4.656612 e -10 f) < tex1Dfetch ( boltzT , ide +2* DIM )) {
        s[ cur ] = -s[ cur ];
    }
}
```

How can measurements of the internal energy, say, be incorporated?
Consider Metropolis kernel for the 2D Ising model discussed before:

**GPU code v3 - kernel**

```c
__global__ void metro_checkerboard_three(spin_t *s, int *ranvec, int offset)
{
    int n = blockDim.x*blockIdx.x + threadIdx.x;
    int cur = blockDim.x*blockIdx.x + threadIdx.x + offset*(N/2);
    int north = cur + (1-2*offset)*(N/2);
    int east = ((north+1)%L) ? north + 1 : north-L+1;
    int west = (north%L) ? north - 1 : north+L-1;
    int south = (n - (1-2*offset)*L + N/2)%N/2 + (1-offset)*(N/2);

    int ide = s[cur]*(s[west]+s[north]+s[east]+s[south]);
    if(fabs(RAN(ranvec[n])*4.656612e-10f) < tex1Dfetch(boltzT, ide+2*DIM)) {
        s[cur] = -s[cur];
    }
}
```

How can measurements of the internal energy, say, be incorporated?

⇒ local changes can be tracked
Consider Metropolis kernel for the 2D Ising model discussed before:

```c
__global__ void metro_checkerboard_three(spin_t *s, int *ranvec, int offset)
{
    int n = blockDim.x*blockIdx.x + threadIdx.x;
    int cur = blockDim.x*blockIdx.x + threadIdx.x + offset*(N/2);
    int north = cur + (1-2*offset)*(N/2);
    int east = ((north+1)%L) ? north + 1 : north-L+1;
    int west = (north%L) ? north - 1 : north+L-1;
    int south = (n - (1-2*offset)*L + N/2)%(N/2) + (1-offset)*(N/2);

    int ide = s[cur]*(s[west]+s[north]+s[east]+s[south]);
    if(fabs(RAN(ranvec[n])*4.656612e-10f) < tex1Dfetch(boltzT, ide+2*DIM)) {
        s[cur] = -s[cur];
    }
}
```

How can measurements of the internal energy, say, be incorporated?
Consider Metropolis kernel for the 2D Ising model discussed before:

```c
__global__ void metro_checkerboard_three(spin_t *s, int *ranvec, int offset) {
    int n = blockDim.x*blockIdx.x + threadIdx.x;
    int cur = blockDim.x*blockIdx.x + threadIdx.x + offset*(N/2);
    int north = cur + (1-2*offset)*(N/2);
    int east = ((north+1)%L) ? north + 1 : north-L+1;
    int west = (north%L) ? north - 1 : north+L-1;
    int south = (n - (1-2*offset)*L + N/2)%(N/2) + (1-offset)*(N/2);

    int ide = s[cur]*(s[west]+s[north]+s[east]+s[south]);
    if(fabs(RAN(ranvec[n])*4.656612e-10f) < tex1Dfetch(boltzT, ide+2*DIM)) {
        s[cur] = -s[cur];
    }
}
```

How can measurements of the internal energy, say, be incorporated?

⇒ local changes can be tracked
energy changes

```c
__global__ void metro_checkerboard_three(spin_t *s, int *ranvec, int offset) {
    int n = blockDim.x*blockIdx.x + threadIdx.x;
    ...
    int south = (n - (1-2*offset)*L + N/2)%(N/2) + (1-offset)*(N/2);

    int ide = s[cur]*(s[west]+s[north]+s[east]+s[south]);
    int ie = 0;
    if(fabs(RAN(ranvec[n])*4.656612e-10f) < tex1Dfetch(boltzT, ide+2*DIM)) {
        s[cur] = -s[cur];
        ie -= 2*ide;
    }
```
__global__ void metro_checkerboard_three(spin_t *s, int *ranvec, int offset) {
    int n = blockDim.x*blockIdx.x + threadIdx.x;
    ...
    int south = (n - (1-2*offset)*L + N/2)%N/2 + (1-offset)*(N/2);
    int ide = s[cur]*(s[west]+s[north]+s[east]+s[south]);
    int ie = 0;
    if(fabs(RAN(ranvec[n])*4.656612e-10f) < tex1Dfetch(boltzT, ide+2*DIM)) {
        s[cur] = -s[cur];
        ie -= 2*ide;
    }
}

__shared__ int deltaE[THREADS];
deltaE[n] = ie;

for(int stride = THREADS>>1; stride > 0; stride >>= 1) {
    __syncthreads();
    if(n < stride) deltaE[n] += deltaE[n+stride];
}

if(n == 0) result[blockIdx.y*GRIDL+blockIdx.x] += deltaE[0];
Access pattern for reduction:

```
0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7
```

```
0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7
```
Recall Hamiltonian:

\[ \mathcal{H} = - \sum_{\langle i, j \rangle} J_{ij} s_i s_j, \]

where \( J_{ij} \) are quenched random variables.
Ising Spin glass

Recall Hamiltonian:

\[ \mathcal{H} = - \sum_{\langle i,j \rangle} J_{ij} s_i s_j, \]

where \( J_{ij} \) are quenched random variables. For reasonable equilibrium results, average over thousands of realizations is necessary.
Ising Spin glass

Recall Hamiltonian:

\[ \mathcal{H} = - \sum_{\langle i, j \rangle} J_{ij} s_i s_j, \]

where \( J_{ij} \) are quenched random variables. For reasonable equilibrium results, average over thousands of realizations is necessary.

- same domain decomposition (checkerboard)
- slightly bigger effort due to non-constant couplings
- higher performance due to larger independence?
- very simple to combine with parallel tempering
Spin glass: performance

![Graph showing the performance of CPU, Tesla C1060, and GTX 480 with varying $N_{\text{replica}}$. The $t_{\text{flip}}$ values are depicted on a logarithmic scale, highlighting the difference in performance between the CPU and GPUs.](image)
Spin glasses: continued

Seems to work well with

- 15 ns per spin flip on CPU
- 70 ps per spin flip on GPU

but not better than ferromagnetic Ising model.
Spin glasses: continued

Seems to work well with

- 15 ns per spin flip on CPU
- 70 ps per spin flip on GPU

but not better than ferromagnetic Ising model.

Further improvement: use multi-spin coding

- Synchronous multi-spin coding: different spins in a single configurations in one word
- Asynchronous multi-spin coding: spins from different realizations in one word
Local updates (again)

Implementation

for(int i = 0; i < SWEEPS_LOCAL; ++i) {
    float r = RAN(ranvecS[n]);
    if(r < boltzD[4]) sS(x1,y) = -sS(x1,y);
    else {
        p1 = JSx(x1m,y) ^ sS(x1,y) ^ sS(x1m,y); p2 = JSx(x1,y) ^ sS(x1,y) ^ sS(x1p,y);
        p3 = JSy(x1,ym) ^ sS(x1,y) ^ sS(x1,ym); p4 = JSy(x1,y) ^ sS(x1,y) ^ sS(x1,yp);
        if(r < boltzD[2]) {
            ido = p1 | p2 | p3 | p4;
            sS(x1,y) = ido ^ sS(x1,y);
        } else {
            ido1 = p1 & p2; ido2 = p1 ^ p2;
            ido3 = p3 & p4; ido4 = p3 ^ p4;
            ido = ido1 | ido3 | (ido2 & ido4);
            sS(x1,y) = ido ^ sS(x1,y);
        }
    }
}
__syncthreads();

r = RAN(ranvecS[n]);
if(r < boltzD[4]) sS(x2,y) = -sS(x2,y);
else {
    p1 = JSx(x2m,y) ^ sS(x2,y) ^ sS(x2m,y); p2 = JSx(x2,y) ^ sS(x2,y) ^ sS(x2p,y);
    p3 = JSy(x2,ym) ^ sS(x2,y) ^ sS(x2,ym); p4 = JSy(x2,y) ^ sS(x2,y) ^ sS(x2,yp);
    if(r < boltzD[2]) {
        ido = p1 | p2 | p3 | p4;
        sS(x2,y) = ido ^ sS(x2,y);
    } else {
        ido1 = p1 & p2; ido2 = p1 ^ p2;
        ido3 = p3 & p4; ido4 = p3 ^ p4;
        ido = ido1 | ido3 | (ido2 & ido4);
        sS(x2,y) = ido ^ sS(x2,y);
    }
}
__syncthreads();
Spin glasses: continued

Seems to work well with

- 15 ns per spin flip on CPU
- 70 ps per spin flip on GPU

but not better than ferromagnetic Ising model.

Further improvement: use multi-spin coding

- Synchronous multi-spin coding: different spins in a single configurations in one word
- Asynchronous multi-spin coding: spins from different realizations in one word

⇒ brings us down to about 2 ps per spin flip
JANUS, a modular massively parallel and reconfigurable FPGA-based computing system.
JANUS, a modular massively parallel and reconfigurable FPGA-based computing system.

<table>
<thead>
<tr>
<th>MODEL</th>
<th>Algorithm</th>
<th>Max size</th>
<th>perf(s)</th>
<th>JANUS</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>3D Ising EA</td>
<td>Metropolis</td>
<td>96³</td>
<td>16 ps</td>
<td>45×</td>
<td>190×</td>
</tr>
<tr>
<td>3D Ising EA</td>
<td>Heat Bath</td>
<td>96³</td>
<td>16 ps</td>
<td>60×</td>
<td></td>
</tr>
<tr>
<td>$Q = 4$ 3D Glassy Potts</td>
<td>Metropolis</td>
<td>16³</td>
<td>64 ps</td>
<td>1250×</td>
<td>1900×</td>
</tr>
<tr>
<td>$Q = 4$ 3D disordered Potts</td>
<td>Metropolis</td>
<td>88³</td>
<td>32 ps</td>
<td>125×</td>
<td></td>
</tr>
<tr>
<td>$Q = 4$, $C_m = 4$ random graph</td>
<td>Metropolis</td>
<td>24000</td>
<td>2.5 ns</td>
<td>2.4×</td>
<td>10×</td>
</tr>
</tbody>
</table>
JANUS, a modular massively parallel and reconfigurable FPGA-based computing system.

<table>
<thead>
<tr>
<th>MODEL</th>
<th>Algorithm</th>
<th>JANUS</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>3D Ising EA</td>
<td>Metropolis</td>
<td>96(^3)</td>
<td>AMSC</td>
</tr>
<tr>
<td>3D Ising EA</td>
<td>Heat Bath</td>
<td>96(^3)</td>
<td>60×</td>
</tr>
<tr>
<td>(Q = 4) 3D Glassy Potts</td>
<td>Metropolis</td>
<td>16(^3)</td>
<td>1250×</td>
</tr>
<tr>
<td>(Q = 4) 3D disordered Potts</td>
<td>Metropolis</td>
<td>88(^3)</td>
<td>1900×</td>
</tr>
<tr>
<td>(Q = 4, C_m = 4) random graph</td>
<td>Metropolis</td>
<td>24000</td>
<td>NO MSC</td>
</tr>
</tbody>
</table>

Costs:

- **Janus**: 256 units, total cost about 700,000 Euros
- **Same performance with GPU**: 64 PCs (2000 Euros) with 2 GTX 295 cards (500 Euros) \(\Rightarrow\) 200,000 Euros
- **Same performance with CPU only** (assuming a speedup of \(\sim 50\)): 800 blade servers with two dual Quadcore sub-units (3500 Euros) \(\Rightarrow\) 2,800,000 Euros
Outline

1 Random number generators

2 Local updates (again)

3 Generalized ensembles
For sufficiently large lattices, one achieves spin-flip times as low as 20 ps, about 250 times faster than a single CPU core. The number of threads is limited by the number of spins.

M. Weigel (Coventry/Mainz)
For sufficiently large lattices, one achieves spin-flip times as low as 20 ps, about 250 times faster than a single CPU core.
For sufficiently large lattices, one achieves spin-flip times as low as 20 ps, about 250 times faster than a single CPU core.

The number of threads is limited by the number of spins.
One way out is to use many Markov chains in parallel. This can be done, in particular, for multicanonical simulations that are used for problems with complex free-energy landscapes and systems with 1st order transitions.
Generalized ensembles

Parallel multicanonical simulations

One way out is to use many Markov chains in parallel. This can be done, in particular, for multicanonical simulations that are used for problems with complex free-energy landscapes and systems with 1st order transitions.

Parallel multicanonical simulations (Zierenberg et al., 2013)
Parallel multicanonical simulations

One way out is to use many Markov chains in parallel. This can be done, in particular, for multicanonical simulations that are used for problems with complex free-energy landscapes and systems with 1st order transitions.

Parallel multicanonical simulations (Zierenberg et al., 2013)

In practise, each walker is represented by a single thread in a grid.
Parallel muca (cont’d)

Each walker samples its own histogram, all of them are combined for the next weight update,

\[ H^{(n)}(E) = \sum_i H_i^{(n)}(E). \]
Each walker samples its own histogram, all of them are combined for the next weight update,

\[ H^{(n)}(E) = \sum_i H_i^{(n)}(E). \]

This scheme can be efficiently implemented on MPI clusters (Zierenberg et al., 2013) and on GPUs.
Combining histograms

How does one best go about in combining the histograms $H^{(n)}(E)$? (At least) three solutions come to mind.

Each walker keeps its own histogram in global memory. Individual histograms are then combined in a separate kernel. ⇒ good memory layout for sampling histograms is not good for combining them.

Each walker stores a list (time series) of energies encountered at each step. The lists are used to create the (total) histogram in a separate kernel. ⇒ memory coalescence, but long lists.

All threads write directly into one global histogram, using atomic operations. ⇒ good in case of few collisions, which is the case for not too small systems.

```
atomicAdd (d_histogram + E, 1);
```

(Atomic operations guarantee the absence of data races, where the result of an operation depends on whether another parallel thread accesses the data in between read and write operations.)
Combining histograms

How does one best go about in combining the histograms \( H^{(n)}(E) \)? (At least) three solutions come to mind

- Each walker keeps its own histogram in global memory. Individual histograms are then combined in a separate kernel.
  ⇒ good memory layout for sampling histograms is not good for combining them.
Combining histograms

How does one best go about in combining the histograms $H^{(n)}(E)$? (At least) three solutions come to mind

- Each walker keeps its own histogram in global memory. Individual histograms are then combined in a separate kernel.
  ⇒ good memory layout for sampling histograms is not good for combining them

- Each walker stores a list (time series) of energies encountered at each step. The lists are used to create the (total) histogram in a separate kernel.
  ⇒ memory coalescence, but long lists
Combining histograms

How does one best go about in combining the histograms $H^{(n)}(E)$? (At least) three solutions come to mind

- Each walker keeps its own histogram in global memory. Individual histograms are then combined in a separate kernel.
  ⇒ good memory layout for sampling histograms is not good for combining them

- Each walker stores a list (time series) of energies encountered at each step. The lists are used to create the (total) histogram in a separate kernel.
  ⇒ memory coalescence, but long lists

- All threads write directly into one global histogram, using atomic operations.
  ⇒ good in case of few collisions, which is the case for not too small systems

  ```
  atomicAdd(d_histogram + E, 1);
  ```
Combining histograms

How does one best go about in combining the histograms $H^{(n)}(E)$? (At least) three solutions come to mind

- Each walker keeps its own histogram in global memory. Individual histograms are then combined in a separate kernel.  
  ⇒ good memory layout for sampling histograms is not good for combining them

- Each walker stores a list (time series) of energies encountered at each step. The lists are used to create the (total) histogram in a separate kernel.  
  ⇒ memory coalescence, but long lists

- All threads write directly into one global histogram, using atomic operations.  
  ⇒ good in case of few collisions, which is the case for not too small systems

  \[
  \text{atomicAdd}(\text{d_histogram} + E, 1);
  \]

(Atomic operations guarantee the absence of data races, where the result of an operation depends on whether another parallel thread accesses the data in between read and write operations.)
Population annealing
Population annealing algorithm (Hukushima + Iba, 2003; Machta, 2010):

1. Set up an equilibrium ensemble of $R$ independent copies of the system at inverse temperature $\beta_0$. Typically $\beta_0 = 0$, where this can be easily achieved.
Population annealing

Population annealing algorithm (Hukushima + Iba, 2003; Machta, 2010):

1. Set up an equilibrium ensemble of $R$ independent copies of the system at inverse temperature $\beta_0$. Typically $\beta_0 = 0$, where this can be easily achieved.

2. To create an approximately equilibrated sample at $\beta_i > \beta_{i-1}$, resample configurations with their relative Boltzmann weight $\exp[-(\beta_i - \beta_{i-1})E_j]/Q$, where $Q = \sum \exp(-(\beta_i - \beta_{i-1})E_j)$. 

3. Update each copy (replica) by $\theta$ rounds of an MCMC algorithm at inverse temperature $\beta_i$.

4. Calculate estimates for observable quantities $O$ as population averages $\sum_j O_j/R$.

5. Goto step 2 until target temperature is reached.

To improve it, all configurations undergo evolution with a standard Markov chain Monte Carlo (MCMC) algorithm ('single spin flips').
Population annealing algorithm (Hukushima + Iba, 2003; Machta, 2010):

1. Set up an equilibrium ensemble of $R$ independent copies of the system at inverse temperature $\beta_0$. Typically $\beta_0 = 0$, where this can be easily achieved.

2. To create an approximately equilibrated sample at $\beta_i > \beta_{i-1}$, resample configurations with their relative Boltzmann weight $\exp[-(\beta_i - \beta_{i-1})E_j]/Q$, where $Q = \sum \exp(-(\beta_i - \beta_{i-1})E_j)$.

3. Calculate estimates for observable quantities $\mathcal{O}$ as population averages $\sum_j \mathcal{O}_j/R$. 

To improve it, all configurations undergo evolution with a standard Markov chain Monte Carlo (MCMC) algorithm ('single spin flips').
Population annealing algorithm (Hukushima + Iba, 2003; Machta, 2010):

1. Set up an equilibrium ensemble of $R$ independent copies of the system at inverse temperature $\beta_0$. Typically $\beta_0 = 0$, where this can be easily achieved.

2. To create an approximately equilibrated sample at $\beta_i > \beta_{i-1}$, resample configurations with their relative Boltzmann weight $\exp[-(\beta_i - \beta_{i-1})E_j]/Q$, where $Q = \sum \exp(-(\beta_i - \beta_{i-1})E_j)$.

3. Update each copy (replica) by $\theta$ rounds of an MCMC algorithm at inverse temperature $\beta_i$.

4. Calculate estimates for observable quantities $O$ as population averages $\sum_j O_j/R$.

5. Goto step 2 until target temperature is reached.
Population annealing

Population annealing algorithm (Hukushima + Iba, 2003; Machta, 2010):

1. Set up an equilibrium ensemble of $R$ independent copies of the system at inverse temperature $\beta_0$. Typically $\beta_0 = 0$, where this can be easily achieved.

2. To create an approximately equilibrated sample at $\beta_i > \beta_{i-1}$, resample configurations with their relative Boltzmann weight $\exp[-(\beta_i - \beta_{i-1})E_j]/Q$, where $Q = \sum \exp(-(\beta_i - \beta_{i-1})E_j)$.

3. Update each copy (replica) by $\theta$ rounds of an MCMC algorithm at inverse temperature $\beta_i$.

4. Calculate estimates for observable quantities $O$ as population averages $\sum_j O_j/R$.

5. Go to step 2 until target temperature is reached.

To improve it, all configurations undergo evolution with a standard Markov chain Monte Carlo (MCMC) algorithm (‘single spin flips’).
Population annealing algorithm (Hukushima + Iba, 2003; Machta, 2010):

1. Set up an equilibrium ensemble of $R$ independent copies of the system at inverse temperature $\beta_0$. Typically $\beta_0 = 0$, where this can be easily achieved.

2. To create an approximately equilibrated sample at $\beta_i > \beta_{i-1}$, resample configurations with their relative Boltzmann weight $\exp[-(\beta_i - \beta_{i-1})E_j]/Q$, where $Q = \sum \exp(-(\beta_i - \beta_{i-1})E_j)$.

3. Update each copy (replica) by $\theta$ rounds of an MCMC algorithm at inverse temperature $\beta_i$.

4. Calculate estimates for observable quantities $\mathcal{O}$ as population averages $\sum_j \mathcal{O}_j/R$.

5. Goto step 2 until target temperature is reached.

To improve it, all configurations undergo evolution with a standard Markov chain Monte Carlo (MCMC) algorithm (‘single spin flips’).
Massively parallel approach

The approach is naturally suitable for an implementation on massively parallel hardware such as GPUs.

![Image of GPU]


Massively parallel approach

The approach is naturally suitable for an implementation on massively parallel hardware such as GPUs.

<table>
<thead>
<tr>
<th>$L$</th>
<th>$t_{SF}$ [ns]</th>
<th>$t_{SF}$ [ns]</th>
<th>speedup</th>
<th>$t_{SF}$ [ns]</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>23.1</td>
<td>0.092</td>
<td>251</td>
<td>0.0096</td>
<td>2406</td>
</tr>
<tr>
<td>32</td>
<td>22.9</td>
<td>0.094</td>
<td>243</td>
<td>0.0095</td>
<td>2410</td>
</tr>
<tr>
<td>64</td>
<td>22.6</td>
<td>0.095</td>
<td>238</td>
<td>0.0098</td>
<td>2306</td>
</tr>
<tr>
<td>128</td>
<td>22.6</td>
<td>0.098</td>
<td>230</td>
<td>0.0098</td>
<td>2306</td>
</tr>
<tr>
<td>256</td>
<td>22.5</td>
<td>0.099</td>
<td>227</td>
<td>0.0098</td>
<td>2295</td>
</tr>
</tbody>
</table>

Parallel scaling

Compare MCMC and PA regarding parallel scaling.
Parallel scaling

Compare MCMC and PA regarding parallel scaling.

Consider total work of parallel implementation. For MCMC we have

\[ W \propto pE + T. \]

and statistical errors are \( \propto 1/\sqrt{T} \). On the other hand, for PA one needs

\[ W \propto R. \]
Parallel scaling

Compare MCMC and PA regarding parallel scaling.

Consider total work of parallel implementation. For MCMC we have

\[ W \propto pE + T. \]

and statistical errors are \( \propto 1/\sqrt{T} \). On the other hand, for PA one needs

\[ W \propto R. \]

The parallel speedup is hence

\[
S = \frac{T_1}{T_p} = \begin{cases} 
\frac{E+T}{E+T/p} & \frac{p \to \infty}{p \to \infty} \quad 1 + \frac{T}{E} \quad \text{MCMC}, \\
p & \quad \infty \quad \text{PA}
\end{cases}
\]
Parallel scaling

Compare MCMC and PA regarding parallel scaling.

Consider total work of parallel implementation. For MCMC we have

\[ W \propto pE + T. \]

and statistical errors are \( \propto 1/\sqrt{T} \). On the other hand, for PA one needs

\[ W \propto R. \]
## Benchmark results for various models considered:

<table>
<thead>
<tr>
<th>System</th>
<th>Algorithm</th>
<th>$L$</th>
<th>CPU ns/flip</th>
<th>C1060 ns/flip</th>
<th>GTX 480 ns/flip</th>
<th>speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>2D Ising</td>
<td>Metropolis</td>
<td>32</td>
<td>8.3</td>
<td>2.58</td>
<td>1.60</td>
<td>3/5</td>
</tr>
<tr>
<td>2D Ising</td>
<td>Metropolis</td>
<td>16384</td>
<td>8.0</td>
<td>0.077</td>
<td>0.034</td>
<td>103/235</td>
</tr>
<tr>
<td>2D Ising, $k = 1$</td>
<td>Metropolis</td>
<td>16384</td>
<td>8.0</td>
<td>0.292</td>
<td>0.133</td>
<td>28/60</td>
</tr>
<tr>
<td>3D Ising</td>
<td>Metropolis</td>
<td>512</td>
<td>14.0</td>
<td>0.13</td>
<td>0.067</td>
<td>107/209</td>
</tr>
<tr>
<td>2D Heisenberg</td>
<td>Metro. double</td>
<td>4096</td>
<td>183.7</td>
<td>4.66</td>
<td>1.94</td>
<td>39/95</td>
</tr>
<tr>
<td>2D Heisenberg</td>
<td>Metro. single</td>
<td>4096</td>
<td>183.2</td>
<td>0.74</td>
<td>0.50</td>
<td>248/366</td>
</tr>
<tr>
<td>2D Heisenberg</td>
<td>Metro. fast math</td>
<td>4096</td>
<td>183.2</td>
<td>0.30</td>
<td>0.18</td>
<td>611/1018</td>
</tr>
<tr>
<td>2D spin glass</td>
<td>Metropolis</td>
<td>32</td>
<td>14.6</td>
<td>0.15</td>
<td>0.070</td>
<td>97/209</td>
</tr>
<tr>
<td>2D spin glass</td>
<td>Metro. multi-spin</td>
<td>32</td>
<td>0.18</td>
<td>0.0075</td>
<td>0.0023</td>
<td>24/78</td>
</tr>
<tr>
<td>2D Ising</td>
<td>Swendsen-Wang</td>
<td>10240</td>
<td>77.4</td>
<td>—</td>
<td>2.97</td>
<td>—/26</td>
</tr>
<tr>
<td>2D Ising</td>
<td>multicanonical</td>
<td>64</td>
<td>42.1</td>
<td>—</td>
<td>0.33</td>
<td>—/128</td>
</tr>
<tr>
<td>2D Ising</td>
<td>Wang-Landau</td>
<td>64</td>
<td>43.6</td>
<td>—</td>
<td>0.94</td>
<td>—/46</td>
</tr>
</tbody>
</table>
This lecture
In this session we considered a number of advanced features of Monte Carlo simulations on GPU, including the choice and implementation of suitable parallel random number generators, the implementation of measurement routines using parallel reductions, as well as generalized-ensemble simulations such as the multicanonical and population annealing methods that replace parallelism via domain decomposition by parallel simulations of system copies.
This lecture

In this session we considered a number of advanced features of Monte Carlo simulations on GPU, including the choice and implementation of suitable parallel random number generators, the implementation of measurement routines using parallel reductions, as well as generalized-ensemble simulations such as the multicanonical and population annealing methods that replace parallelism via domain decomposition by parallel simulations of system copies.

Next lecture

In the next and final lecture, we leave the issue of spin models behind and look into some advanced GPU computing features such as the integration with MPI and the use of OpenACC.
Summary and outlook

This lecture

In this session we considered a number of advanced features of Monte Carlo simulations on GPU, including the choice and implementation of suitable parallel random number generators, the implementation of measurement routines using parallel reductions, as well as generalized-ensemble simulations such as the multicanonical and population annealing methods that replace parallelism via domain decomposition by parallel simulations of system copies.

Next lecture

In the next and final lecture, we leave the issue of spin models behind and look into some advanced GPU computing features such as the integration with MPI and the use of OpenACC.

Reading

Ising and Heisenberg models:

- Add some of the improvements discussed in this lecture. In most cases, this also requires changes to the driver code (execution configuration, memory layout etc.), not just insertion of the kernel code.
- Compare timings for different kernel versions and different block sizes etc.
- Change the code for simulations of the Heisenberg model. Check the stability.
- Add the necessary statements for measuring energies and magnetizations. Use parallel reductions.

Histograms: check the code sample histogram.tgz and write the corresponding kernel(s) to implement a code that creates a histogram out of a sequence of events.