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

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 $[0, \mathrm{NMAX}]$ 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 $[0, \mathrm{NMAX}]$ 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)

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.

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)

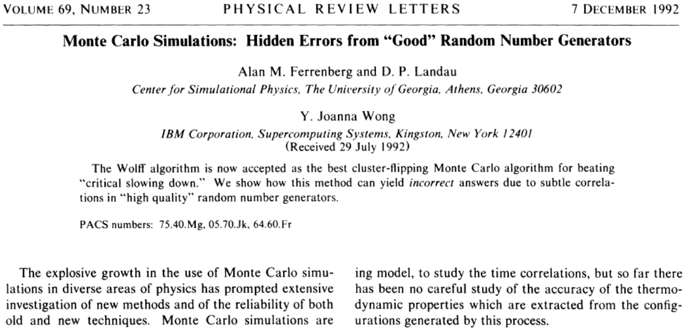
For any pseudo RNG (or RNG, for short) there must exist an algorithm/test that distinguishes the generated sequence from a truly random sequence. (If nothing else, this can be the algorithm generating the sequence itself!)

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!)



VOLUME 69, NUMBER 23 PHYSICAL REVIEW LETTERS 7 DECEMBER 1992

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

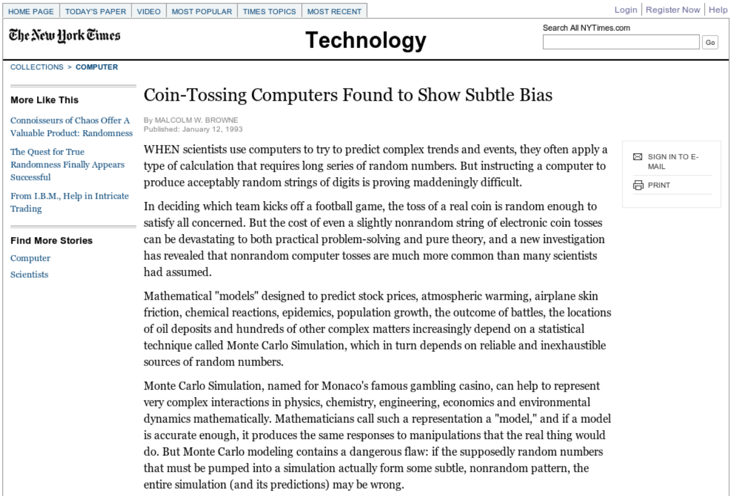
Y. Joanna Wong
IBM Corporation, Supercomputing Systems, Kingston, New York 12401
(Received 29 July 1992)

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.Jk, 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 ing model, to study the time correlations, but so far there has been no careful study of the accuracy of the thermodynamic properties which are extracted from the configurations generated by this process.

The story of R250


HOME PAGE | TODAY'S PAPER | VIDEO | MOST POPULAR | TIMES TOPICS | MOST RECENT

Login | Register Now | Help

The New York Times

Technology

Search All NYTimes.com

Go

COLLECTIONS > COMPUTER

More Like This

Connoisseurs of Chaos Offer A Valuable Product: Randomness

The Quest for True Randomness Finally Appears Successful

From I.B.M., Help in Intricate Trading

Find More Stories

Computer

Scientists

Coin-Tossing Computers Found to Show Subtle Bias

By MALCOLM W. BROWNE
Published: January 12, 1993

SIGN IN TO E-MAIL

PRINT

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.

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 $(u_0, \dots, u_{t-1})$ 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.

Linear congruential generators

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

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

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

Linear congruential generators

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

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

Linear congruential generators

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

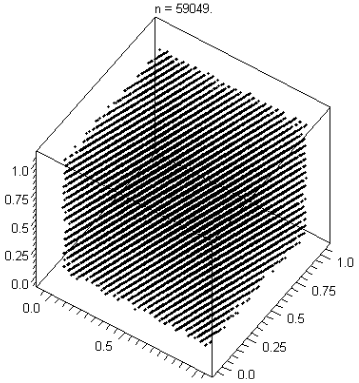
$$x_{n+1} = ax_n + c \pmod{m}.$$

- 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

Linear congruential generators

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

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



(Source: Wikipedia)

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

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

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

Linear congruential generators

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
- 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:

LCGs: implementation

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

LCG implementation

```
#define A32 1664525
#define C32 1013904223

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

LCGs: implementation

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

LCG implementation

```
#define A32 1664525
#define C32 1013904223

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

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

LCGs: implementation

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

LCG implementation

```
#define A32 1664525
#define C32 1013904223

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

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

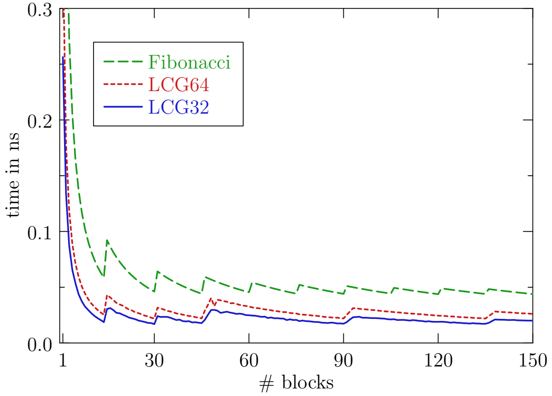
LCG implementation

```
#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);
```

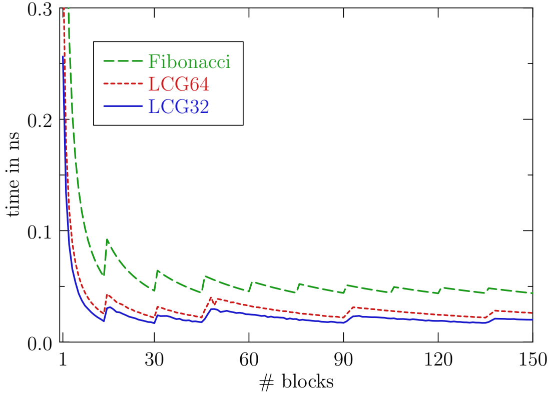
LCG: performance

How well do they perform?



LCG: performance

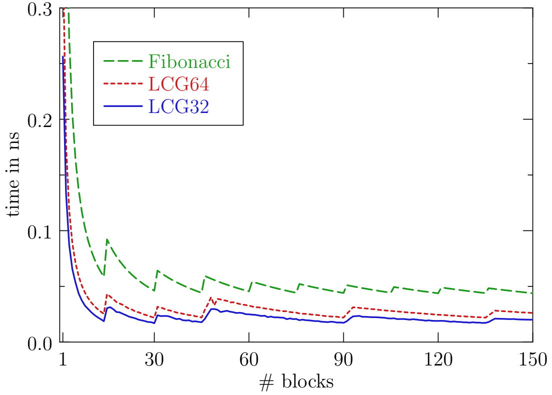
How well do they perform?



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 58×10^9 (LCG32) and 46×10^9 (LCG64) random numbers per second, respectively.

LCG: overall benchmarks

Use these LCG generators for the previously developed simulation code for the 2D Ising model.

LCG: overall benchmarks

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 1024×1024 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 1024×1024 Ising model with periodic boundary conditions at $\beta = 0.4$.

method	e	Δ_{rel}	C_V	Δ_{rel}	$t_{\text{up}}^{k=1}$	$t_{\text{up}}^{k=100}$
exact	1.106079207	0	0.8616983594	0		
			sequential update (CPU)			
LCG32	1.1060788(15)	−0.26	0.83286(45)	−63.45		
LCG64	1.1060801(17)	0.49	0.86102(60)	−1.14		
Fibonacci, $r = 512$	1.1060789(17)	−0.18	0.86132(59)	−0.64		
			checkerboard update (GPU)			
LCG32	1.0944121(14)	−8259.05	0.80316(48)	−121.05	0.2221	0.0402
LCG32, random	1.1060775(18)	−0.97	0.86175(56)	0.09	0.2221	0.0402
LCG64	1.1061058(19)	13.72	0.86179(67)	0.14	0.2311	0.0471
LCG64, random	1.1060803(18)	0.62	0.86215(63)	0.71	0.2311	0.0471
MWC, same a	1.1060800(18)	0.45	0.86161(60)	−0.15	0.2293	0.0435
MWC, different a	1.1060797(18)	0.28	0.86168(62)	−0.03	0.2336	0.0438
Fibonacci, $r = 521$	1.1060890(15)	6.43	0.86099(66)	−1.09	0.2601	0.0661
Fibonacci, $r = 1279$	1.1060800(19)	0.40	0.86084(53)	−1.64	0.2904	0.0700
XORWOW (cuRAND)	1.1060654(15)	−9.13	0.86167(65)	0.04	0.7956	0.0576
XORShift/Weyl	1.1060788(18)	−0.23	0.86184(53)	0.27	0.2613	0.0721
Philox4x32_7	1.1060778(18)	−0.79	0.86109(65)	−0.93	0.2399	0.0523
Philox4x32_10	1.1060777(17)	−0.85	0.86188(61)	0.30	0.2577	0.0622

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 1024×1024 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

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.

generator	bits/thread	failures in TestU01			Ising test	perf.
		SmallCrush	Crush	BigCrush		$\times 10^9$/s
LCG32	32	12	—	—	failed	58
LCG32, random	32	3	14	—	passed	58
LCG64	64	None	6	—	failed	46
LCG64, random	64	None	2	8	passed	46
MWC	$64 + 32$	1	29	—	passed	44
Fibonacci, $r = 521$	≥ 80	None	2	—	failed	23
Fibonacci, $r = 1279$	≥ 80	None	(1)	2	passed	23
XORWOW (cuRAND)	192	None	None	1/3	failed	19
MTGP (cuRAND)	≥ 44	None	2	2	—	18
XORShift/Weyl	32	None	None	None	passed	18
Philox4x32_7	(128)	None	None	None	passed	41
Philox4x32_10	(128)	None	None	None	passed	30

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 1024×1024 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

Other generators

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

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

  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.

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

M. Mansen, M. Weigel, and A. K. Hartmann, Eur. Phys. J. Special Topics 210, 53 (2012.)

Outline

Ising model: Measurements

Consider Metropolis kernel for the 2D Ising model discussed before:

Ising model: Measurements

Consider Metropolis kernel for the 2D Ising model discussed before:

GPU code v3 - kernel

```
__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];
  }
}
```

Ising model: Measurements

Consider Metropolis kernel for the 2D Ising model discussed before:

GPU code v3 - kernel

```
__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?

Ising model: Measurements

Consider Metropolis kernel for the 2D Ising model discussed before:

GPU code v3 - kernel

```
__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?

$\Rightarrow$ local changes can be tracked

Ising model: Measurements (cont'd)

energy changes

```
__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;
  }
```

Ising model: Measurements (cont'd)

energy changes

```
__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;
  }
```

butterfly sum

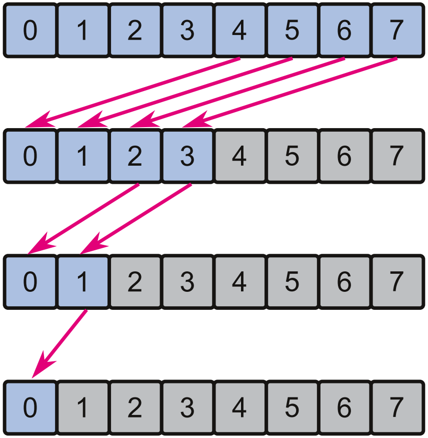
```
  __shared__ int deltaE[THREADS];
  deltaE[n] = ie;

  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];
}
```

Ising model: Measurements (cont'd)

Access pattern for reduction:



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.

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

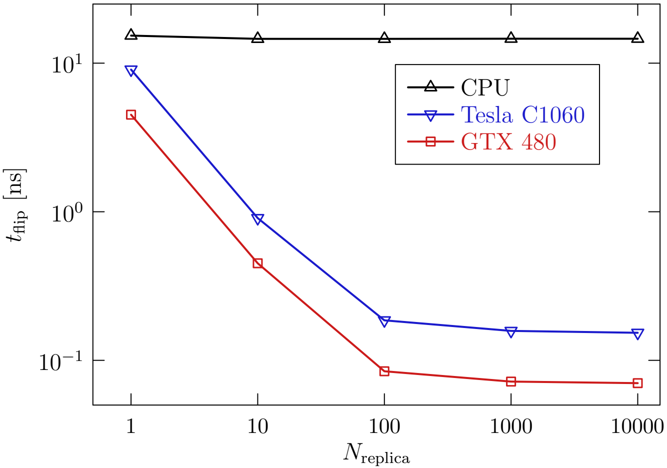
Recall Hamiltonian:

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

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



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

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.

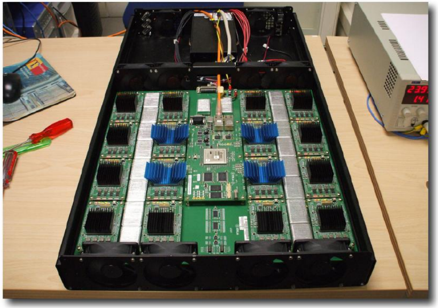
Further improvement: use multi-spin coding

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

$\Rightarrow$ brings us down to about 2 ps per spin flip

Janus

JANUS, a modular massively parallel and reconfigurable FPGA-based computing system.



Janus

JANUS, a modular massively parallel and reconfigurable FPGA-based computing system.

		JANUS		PC		
MODEL	Algorithm	Max size	perfs	AMSC	SMSC	NO MSC
3D Ising EA	Metropolis	96^3	16 ps	$45\times$	$190\times$	
3D Ising EA	Heat Bath	96^3	16 ps	$60\times$		
$Q=4$ 3D Glassy Potts	Metropolis	16^3	64 ps	$1250\times$	$1900\times$	
$Q=4$ 3D disordered Potts	Metropolis	88^3	32 ps	$125\times$		$1800\times$
$Q=4$, $C_m=4$ random graph	Metropolis	24000	2.5 ns	$2.4\times$		$10\times$

Janus

JANUS, a modular massively parallel and reconfigurable FPGA-based computing system.

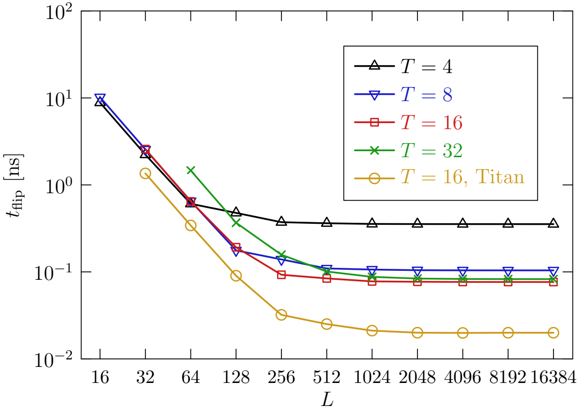
		JANUS		PC		
MODEL	Algorithm	Max size	perfs	AMSC	SMSC	NO MSC
3D Ising EA	Metropolis	96^3	16 ps	$45\times$	$190\times$	
3D Ising EA	Heat Bath	96^3	16 ps	$60\times$		
$Q = 4$ 3D Glassy Potts	Metropolis	16^3	64 ps	$1250\times$	$1900\times$	
$Q = 4$ 3D disordered Potts	Metropolis	88^3	32 ps	$125\times$		$1800\times$
$Q = 4$, $C_m = 4$ random graph	Metropolis	24000	2.5 ns	$2.4\times$		$10\times$

Costs:

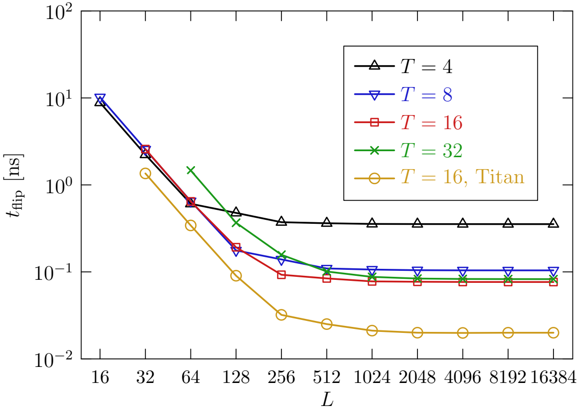
- Janus: 256 units, total cost about $700,000$ Euros
- Same performance with GPU: 64 PCs (2000 Euros) with 2 GTX 295 cards (500 Euros) $\Rightarrow 200,000$ Euros
- Same performance with CPU only (assuming a speedup of ~ 50): 800 blade servers with two dual Quadcore sub-units (3500 Euros) $\Rightarrow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

Outline

Performance

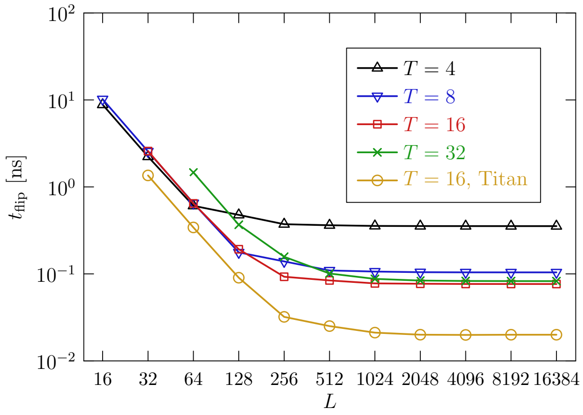


Performance



For sufficiently large lattices, one achieves spin-flip times as low as 20 ps, about 250 times faster than a single CPU core.

Performance



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.

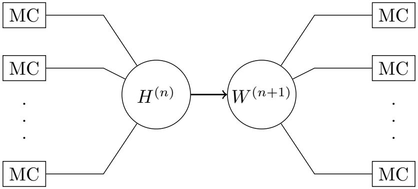
Parallel multicanonical simulations

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

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.

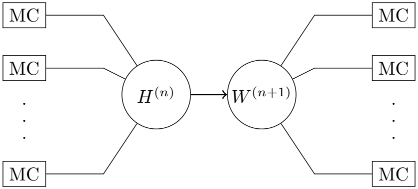
Parallel multicanonical simulations (Zierenberg et al., 2013)



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,

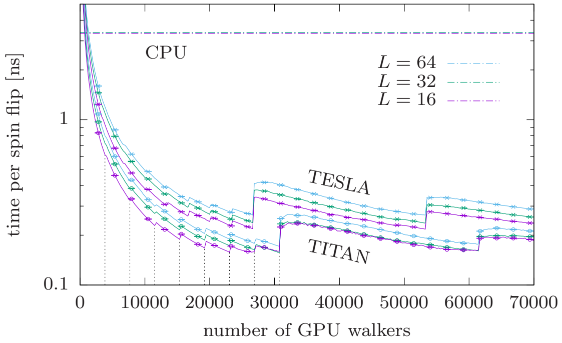
$$H^{(n)}(E) = \sum_i H_i^{(n)}(E).$$

Parallel muca (cont'd)

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

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

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

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.
  $\Rightarrow$ 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.
  $\Rightarrow$ 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.
  $\Rightarrow$ 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.
  $\Rightarrow$ 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.
  $\Rightarrow$ memory coalescence, but long lists
- All threads write directly into one global histogram, using atomic operations.
  $\Rightarrow$ good in case of few collisions, which is the case for not too small systems

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

Combining histograms

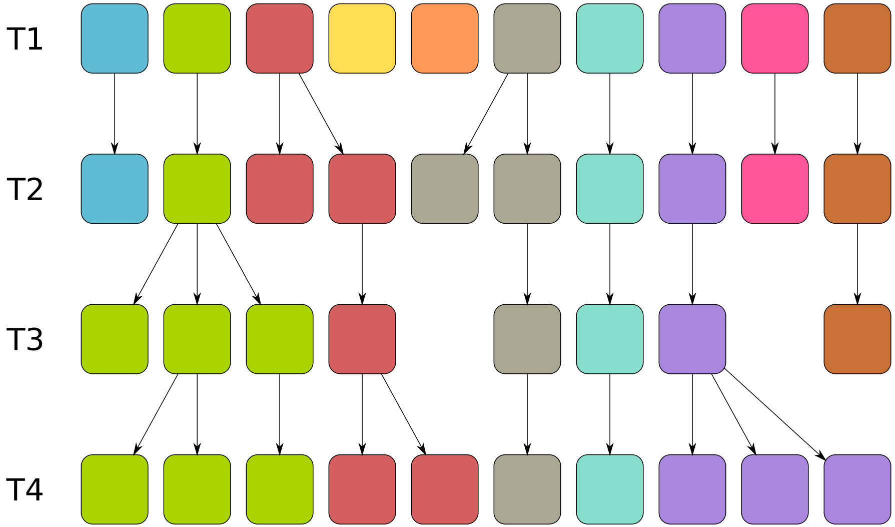
How does one best go about in combining the histograms $H^{(n)}(E)$? (At least) three solutions come to mind

- Each walker keeps its own histogram in global memory. Individual histograms are then combined in a separate kernel.
  $\Rightarrow$ 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.
  $\Rightarrow$ memory coalescence, but long lists
- All threads write directly into one global histogram, using atomic operations.
  $\Rightarrow$ 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.)

Population annealing



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 β_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 β_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 $Q = \sum \exp(-(\beta_i - \beta_{i-1})E_j)$.

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 β_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 $Q = \sum \exp(-(\beta_i - \beta_{i-1})E_j)$.

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

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 β_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 $Q = \sum \exp(-(\beta_i - \beta_{i-1})E_j)$.

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.

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 β_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 $Q = \sum \exp(-(\beta_i - \beta_{i-1})E_j)$.

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').

Population annealing

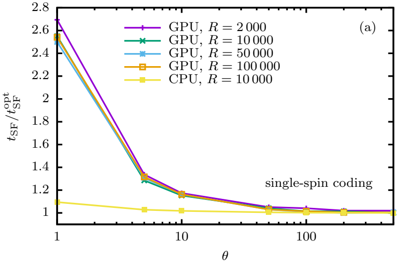
Population annealing algorithm `(Hukushima + Iba, 2003; Machta, 2010)`:

1. Set up an equilibrium ensemble of R independent copies of the system at inverse temperature β_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 θ rounds of an MCMC algorithm at inverse temperature β_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.



L. Barash, MW, M. Borovský, W, Janke, and L. Shchur,
Comput. Phys. Commun. 220, 341 (2017).
Code at `github.com/LevBarash/PAising`.

Massively parallel approach

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

	CPU	GPU			
		SSC		MSC	
L	t_{SF} [ns]	t_{SF} [ns]	speedup	t_{SF} [ns]	speedup
16	23.1	0.092	251	0.0096	2406
32	22.9	0.094	243	0.0095	2410
64	22.6	0.095	238	0.0098	2306
128	22.6	0.098	230	0.0098	2306
256	22.5	0.099	227	0.0098	2295

L. Barash, MW, M. Borovský, W, Janke, and L. Shchur,
Comput. Phys. Commun. 220, 341 (2017).
Code at `github.com/LevBarash/PAising`.

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

$$S = \frac{T_1}{T_p} = \begin{cases} \frac{E+T}{E+T/p} & \stackrel{p\to\infty}{\longrightarrow} \quad 1 + \frac{T}{E} \quad \text{MCMC,} \\ p & \stackrel{p\to\infty}{\longrightarrow} \quad \infty \quad \text{PA} \end{cases}$$

Parallel scaling

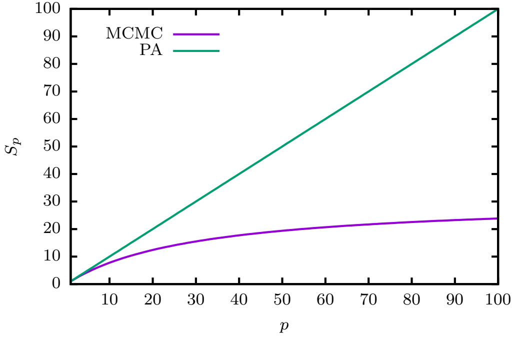
Compare MCMC and PA regarding parallel scaling.

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

$$W \propto R.$$



Performance

Benchmark results for various models considered:

System	Algorithm	L	CPU ns/flip	C1060 ns/flip	GTX 480 ns/flip	speed-up
2D Ising	Metropolis	32	8.3	2.58	1.60	3/5
2D Ising	Metropolis	16 384	8.0	0.077	0.034	103/235
2D Ising	Metropolis, $k = 1$	16 384	8.0	0.292	0.133	28/60
3D Ising	Metropolis	512	14.0	0.13	0.067	107/209
2D Heisenberg	Metro. double	4096	183.7	4.66	1.94	39/95
2D Heisenberg	Metro. single	4096	183.2	0.74	0.50	248/366
2D Heisenberg	Metro. fast math	4096	183.2	0.30	0.18	611/1018
2D spin glass	Metropolis	32	14.6	0.15	0.070	97/209
2D spin glass	Metro. multi-spin	32	0.18	0.0075	0.0023	24/78
2D Ising	Swendsen-Wang	10240	77.4	—	2.97	–/26
2D Ising	multicanonical	64	42.1	—	0.33	–/128
2D Ising	Wang-Landau	64	43.6	—	0.94	–/46

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.

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.

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

- M. Manssen, M. Weigel, and A. K. Hartmann, Eur. Phys. J. Special Topics **210**, 53 (2012).
- L. Yu. Barash, M. Weigel, M. Borovský, W. Janke, and L. N. Shchur, Comput. Phys. Commun. 220, 341 (2017).
- J. Gross, J. Zierenberg, M. Weigel, and W. Janke, Comput. Phys. Commun. 224, 387 (2018).

Exercises

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.