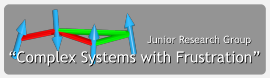
Simulating spin models on GPU

Lecture 4: Advanced Techniques

Martin Weigel

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

IMPRS School 2012: GPU Computing,
Wroclaw, Poland, November 1, 2012



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?

⇒ 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;
  }
```

butterfly sum

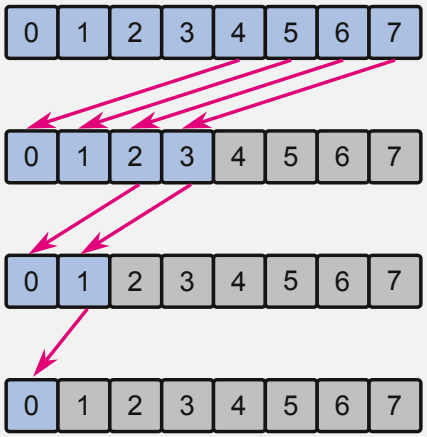
```
  __shared__ int deltaE[THREADS];
  deltaE[n] = ie;

  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

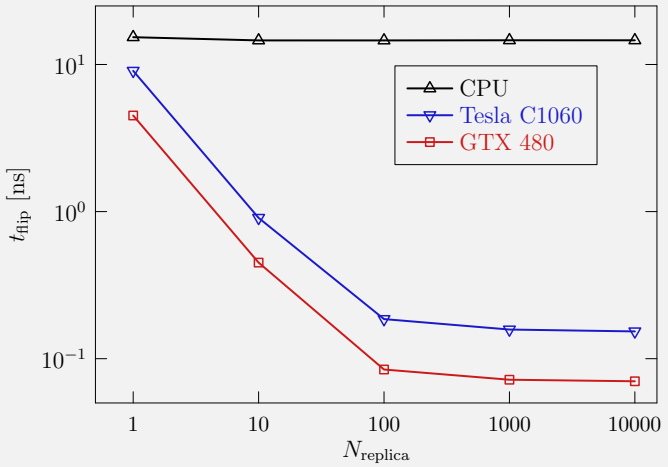
Recall Hamiltonian:

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

where J_{ij} are quenched random variables. 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.

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

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.

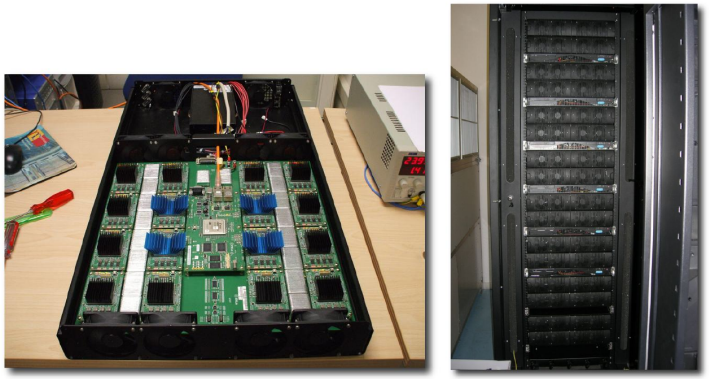
Further improvement: use multi-spin coding

- 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

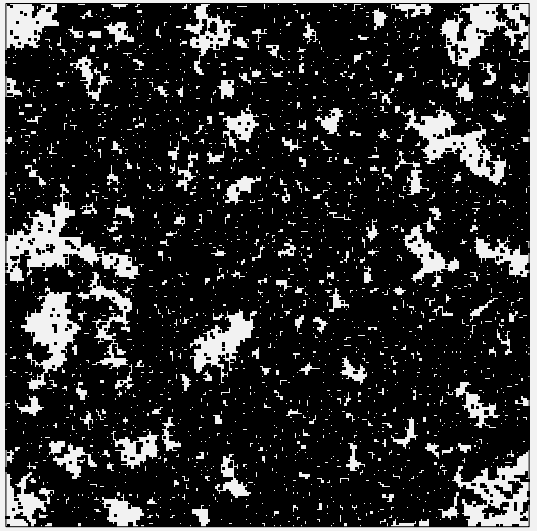
JANUS, a modular massively parallel and reconfigurable FPGA-based computing system.

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

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

Critical configuration



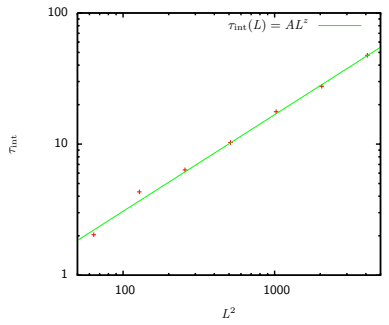
Cluster algorithms

Beat critical slowing down

Spatial correlations close to a continuous phase transition impede decorrelation with local spin flips.

Need to

- update non-local variables
- of characteristic size, i.e., percolating at transition, that
- have the right geometric properties, i.e., fractal dimensions etc.



Cluster algorithms

These are strong requirements only fulfilled for a few algorithms, most notably for the Potts, O(n) and related lattice models.

Cluster algorithms

Would need to use cluster algorithms for efficient equilibrium simulation of spin models at criticality:

1. Activate bonds between like spins with probability $p = 1 - e^{-2\beta J}$.
2. Construct (Swendsen-Wang) spin clusters from domains connected by active bonds.
3. Flip independent clusters with probability $1/2$.
4. Goto 1.

Steps 1 and 3 are local ⇒ Can be efficiently ported to GPU.
What about step 2? ⇒ Domain decomposition into tiles.

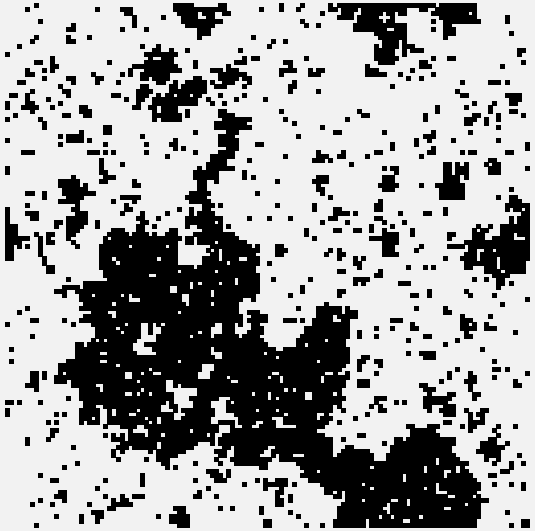
labeling *inside* of domains

- Hoshen-Kopelman
- breadth-first search
- self-labeling
- union-find algorithms

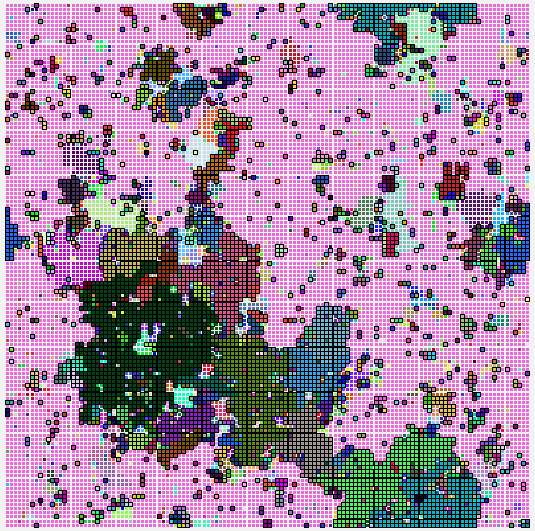
relabeling *across* domains

- self-labeling
- hierarchical approach
- iterative relaxation

Swendsen-Wang update



Swendsen-Wang update



Label activation

label activation

```
__global__ void prepare_bonds_small(spin_t *s, bond_t *bond, int *inclus, int
     *ranvec)
{
  int x = blockIdx.x*(L/gridDim.x)+threadIdx.x;
  int y = blockIdx.y*(L/gridDim.y)+threadIdx.y;
  bond[y*L+x] = (spin(x,y) == spin((x==L-1)?0:x+1,y) && RAN(ranvec[y*L+x]) >
       BOLTZ) ? 1 : 0;
  bond[y*L+x+N] = (spin(x,y) == spin(x,(y==L-1)?0:y+1) && RAN(ranvec[y*L+x])
       > BOLTZ) ? 1 : 0;
  inclus[y*L+x] = y*L+x;
}
```

- one thread per site
- thread updates two bonds
- low load, therefore updating more bonds per thread beneficial

BFS or Ants in the Labyrinth

56	57	58	59	60	61	62	63
48	49	50	51	52	53	54	55
40	41	42	43	36	36	46	47
32	33	34	36	36	36	38	39
24	25	26	36	28	36	30	31
16	17	18	19	20	21	22	23
8	9	10	11	12	13	14	15
0	1	2	3	4	5	6	7

only wave-front vectorization would be possible $\Rightarrow$ many idle threads

BFS code

breadth-first search on tiles

```
__global__ void localBFS(int *inclus, bond_t *bond, int *next)
{
  int clus = 0, clus_leng, tested;

  int xmin = blockIdx.x*(L/gridDim.x), xmax = (blockIdx.x+1)*(L/gridDim.x);
  int ymin = blockIdx.y*(L/gridDim.y), ymax = (blockIdx.y+1)*(L/gridDim.y);
  int offset = ymin*L+xmin;

  for(int y = ymin; y < ymax; ++y) {
    for(int x = xmin; x < xmax; ++x) {
      if(inclus[y*L+x] <= clus) continue;
      clus_leng = tested = 0;
      clus = next[offset] = y*L+x;
      while(tested <= clus_leng) {
        int xx = next[offset+tested] % L;
        int yy = next[offset+tested] / L;
        int k1 = yy*L+(xx+1);
        if(xx < xmax-1 && inclus[k1] > clus && bond[yy*L+xx]) {
          ++clus_leng;
          next[offset+clus_leng] = k1;
          inclus[k1] = clus;
        }
        k1 = yy*L+(xx-1);
        if(xx > xmin && inclus[k1] > clus && bond[k1]) {
          ++clus_leng;
          next[offset+clus_leng] = k1;
          inclus[k1] = clus;
        }
        ... // other directions
        ++tested;
      }
    }
  }
}
```


Self-labeling

56	57	58	59	60	61	62	63
48	49	49	51	52	53	54	55
40	33	42	43	44	45	46	47
32	33	34	35	36	37	38	39
24	25	26	19	28	29	30	31
16	17	18	19	19	21	22	23
8	9	10	11	12	13	14	15
0	1	2	3	4	5	6	7

effort is O(L^3) at the critical point, but can be vectorized with O(L^2) threads

Self-labeling code

self-labeling on tiles

```
do {
  changed = 0;
  if(bnd[me] && in[me] != in[right]) {
    atomicMin(in+me,in[right]);
    atomicMin(in+right,in[me]);
    changed = 1;
  }
  if(bnd[me+BLOCKL*BLOCKL] && in[me] != in[up]) {
    atomicMin(in+me,in[up]);
    atomicMin(in+up,in[me]);
    changed = 1;
  }
} while(__syncthreads_or(changed));
```

- atomic operations:
  - guarantee no interference by other threads
  - fast intrinsic implementation
  - `atomicAdd()`, `atomicMin()`, `atomicInc()`, and others
- `__syncthreads_or()` synchronizes threads and returns true if *any* of the thread expression evaluates to true

Self-labeling code

self-labeling on tiles

```
do {
  changed = 0;
  if(bnd[me] && in[me] != in[right]) {
    in[me] = in[right] = min(in[me], in[right]);
    changed = 1;
  }
  if(bnd[me+BLOCKL*BLOCKL] && in[me] != in[up]) {
    in[me] = in[up] = min(in[me], in[up]);
    changed = 1;
  }
} while(__syncthreads_or(changed));
```

- synchronization not strictly necessary here
- "non-deterministic" version is faster
- might need a couple of more iterations

Union-find

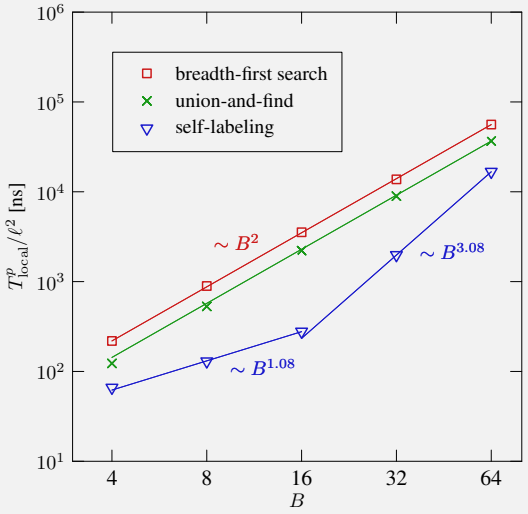
56	57	58	59	60	61	62	63
48	41	41	51	52	53	54	55
40	32	41	41	44	45	46	47
32	32	34	30	30	30	38	39
24	25	26	27	30	30	13	31
16	17	18	19	20	21	13	23
8	9	10	11	12	13	13	15
0	1	2	3	4	5	6	7

tree structure with two optimizations:

- balanced trees
- path compression

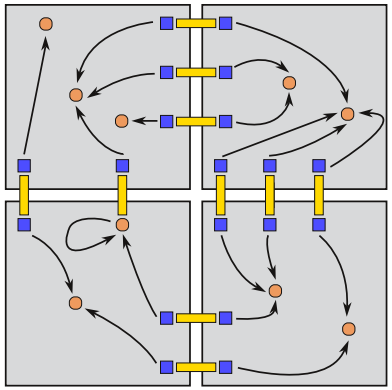
$\Rightarrow$ root finding and cluster union essentially O(1) operations

Comparison



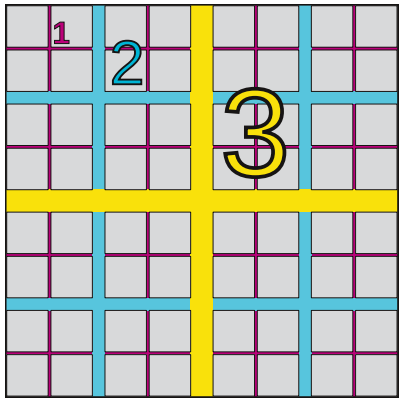
Label consolidation

Label relaxation



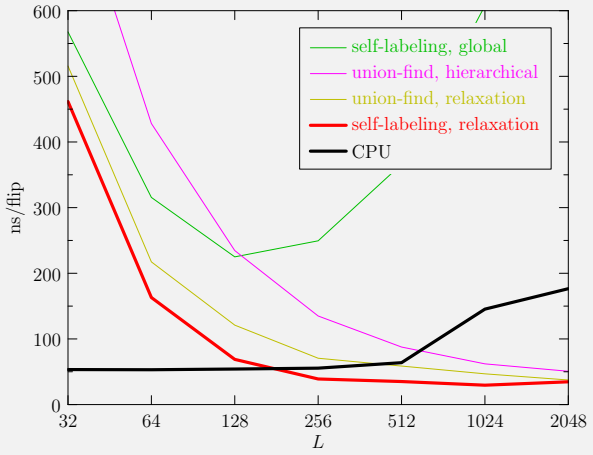
$$T^p_{\text{global}} = C_{\text{rel}} \frac{\ell^2}{\min(\ell^2, n)} \frac{B}{\min(B, m)} \ell^{d_{\min}}$$

Hierarchical sewing

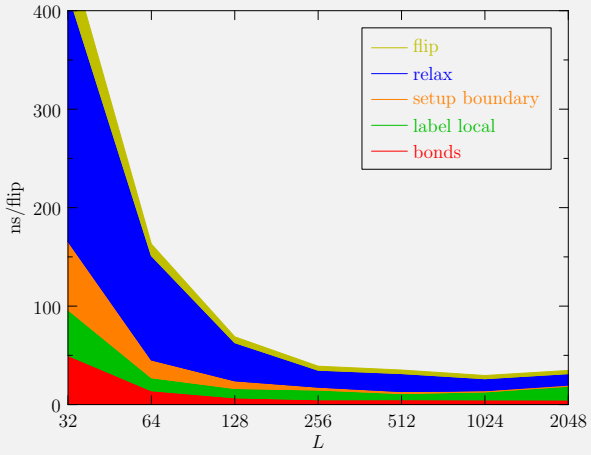


$$T^p_{\text{global}} = C_{\text{sew}} L^2 \left[\frac{1}{nB} + \left(4 - \frac{5}{\sqrt{n}}\right) \frac{1}{L} \right]$$

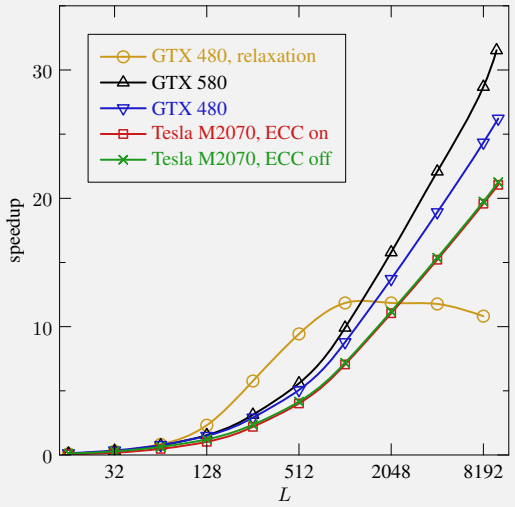
Performance



Performance



Performance



Multicanonical simulations

Generalized ensembles

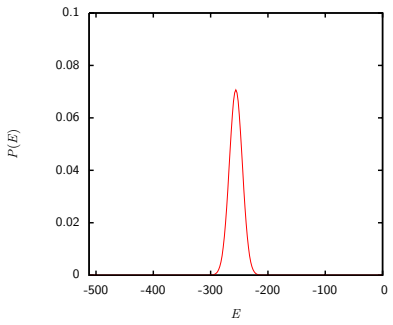
Instead of simulating the canonical distribution,

$$P_K(E) = \frac{1}{Z_K}\Omega(E)e^{-KE},$$

consider using a more general distribution

$$P_{\mathrm{muca}}(E) = \frac{\Omega(E)/W(E)}{Z_{\mathrm{muca}}} = \frac{\Omega(E)e^{-\omega(E)}}{Z_{\mathrm{muca}}},$$

engineered to overcome barriers, improve sampling speed and extend the reweighting range.



Multicanonical simulations

Choice of weights

To overcome barriers, we need to *broaden* $P(E)$, in the extremal case to a *constant* distribution,

$$P_{\mathrm{muca}}(E) = Z_{\mathrm{muca}}^{-1}\,\Omega(E)/W(E) = Z_{\mathrm{muca}}^{-1}\,e^{S(E)-\omega(E)} \stackrel{!}{=} \mathrm{const},$$

where $S(E) = \ln\Omega(E)$ is the microcanonical entropy.

Under these assumptions, $W(E) = \Omega(E)$ is optimal, i.e., we again desire to estimate the density of states. This is not known a priori, so (again) use histogram estimator

$$\hat{\Omega}(E) = Z_{\mathrm{muca}}\,\hat{H}_{\mathrm{muca}}(E)/N \times e^{\omega(E)}.$$

Canonical averages can be recovered at any time by reweighting:

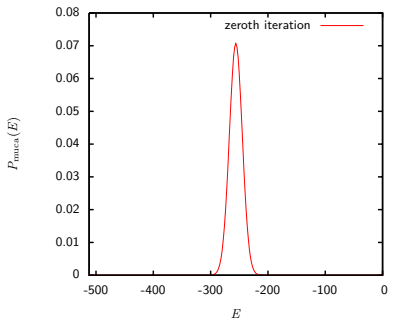
$$\langle A\rangle_K = \frac{\sum_E A(E)P_K(E)/P_{\mathrm{muca}}(E)}{\sum_E P_K(E)/P_{\mathrm{muca}}(E)}$$

Multicanonical simulations

Muca iteration

Determine muca weights/density of states iteratively:

1. Use, e.g., a $K = 0$ canonical simulation to get initial estimate $\hat{S}_0(E) = \ln\hat{\Omega}_0(E)$.
2. Choose multicanonical weights $\omega_1(E) = \hat{S}_0(E)$ for next simulation.
3. Iterate.

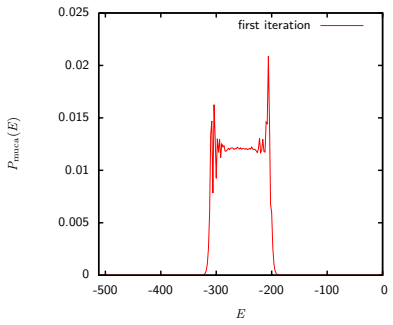


Multicanonical simulations

Muca iteration

Determine muca weights/density of states iteratively:

1. Use, e.g., a $K = 0$ canonical simulation to get initial estimate $\hat{S}_0(E) = \ln\hat{\Omega}_0(E)$.
2. Choose multicanonical weights $\omega_1(E) = \hat{S}_0(E)$ for next simulation.
3. Iterate.

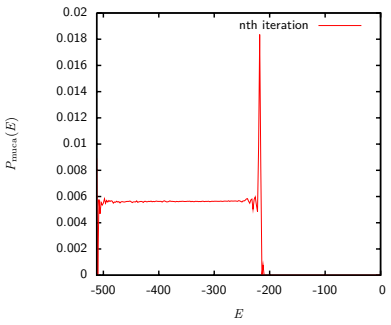


Multicanonical simulations

Muca iteration

Determine muca weights/density of states iteratively:

1. Use, e.g., a $K = 0$ canonical simulation to get initial estimate $\hat{S}_0(E) = \ln \hat{\Omega}_0(E)$.
2. Choose multicanonical weights $\omega_1(E) = \hat{S}_0(E)$ for next simulation.
3. Iterate.



Multicanonical simulations

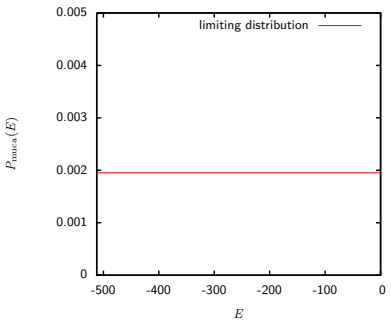
Muca iteration

Determine muca weights/density of states iteratively:

1. Use, e.g., a $K = 0$ canonical simulation to get initial estimate $\hat{S}_0(E) = \ln \hat{\Omega}_0(E)$.
2. Choose multicanonical weights $\omega_1(E) = \hat{S}_0(E)$ for next simulation.
3. Iterate.

Advantages:

- always in equilibrium
- arbitrary distributions possible
- system ideally performs an unbiased random walk in energy space → fast(er) dynamics

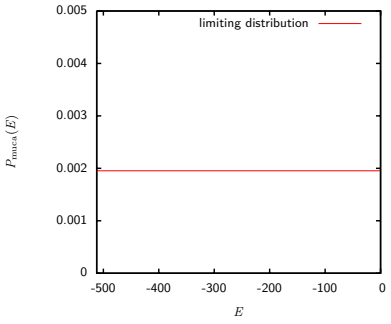


Multicanonical simulations

Multicanonical simulations

Variants:

- umbrella sampling, entropic sampling (identical)
- multiple Gaussian modified ensemble
- (broad histogram method)
- transition-matrix Monte Carlo
- metadynamics
- Wang-Landau sampling
- ...



Multicanonical simulations

Wang-Landau sampling

Muca weights are updated as

$$\omega_{i+1}(E) - \omega_i(E) = \mathrm{const} + \ln \hat{H}_i(E),$$

i.e., if an energy E is visited more often than others, it receives *less* weight in future iterations.

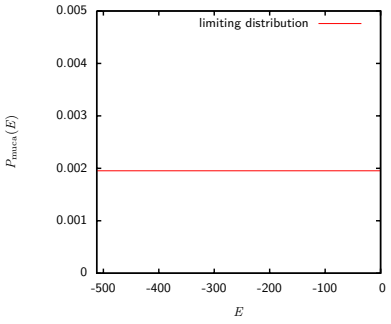
This behavior can be imitated in a one-step iteration: simulate

$$P_{\mathrm{WL}}(E) \propto \Omega(E) e^{-\omega(E)},$$

but update

$$\omega_{i+1}(E) - \omega_i(E) = \phi,$$

each time an energy E is seen.



Multicanonical simulations

Wang-Landau sampling

Muca weights are updated as

$$\omega_{i+1}(E) - \omega_i(E) = \text{const} + \ln \hat{H}_i(E),$$

i.e., if an energy E is visited more often than others, it receives *less* weight in future iterations.

This behavior can be imitated in a one-step iteration: simulate

$$P_{\text{WL}}(E) \propto \Omega(E) e^{-\omega(E)},$$

but update

$$\omega_{i+1}(E) - \omega_i(E) = \phi,$$

each time an energy E is seen.

WL iteration

1. Start with $\omega(E) = 0 \; \forall E$.
2. Simulate "sufficiently long" while continuously updating $\omega(E)$.
3. Reduce modification factor, e.g., $\phi \to \phi/2$
4. Iterate till $\phi < \phi_{\text{thres}}$.

Multicanonical simulations

Parallelization

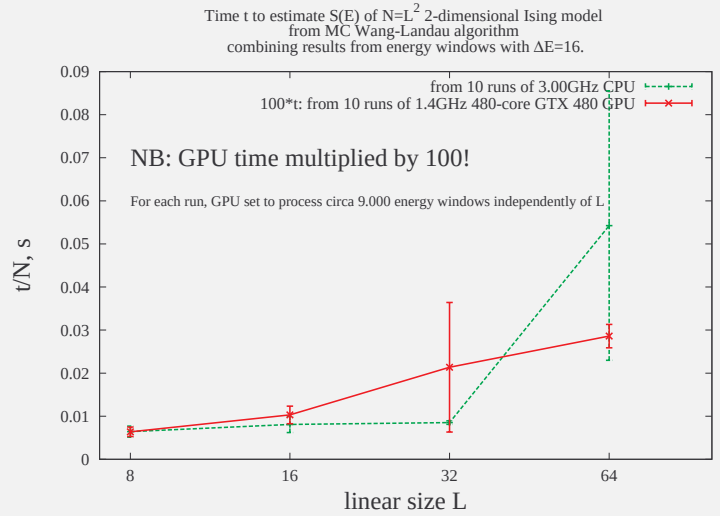
Problem: dependence on *global* quantity E (or M, q, ...) effectively serializes all calculations. But:

- can subdivide range of E in windows
- can increase statistics (and estimate statistical errors) with independent runs

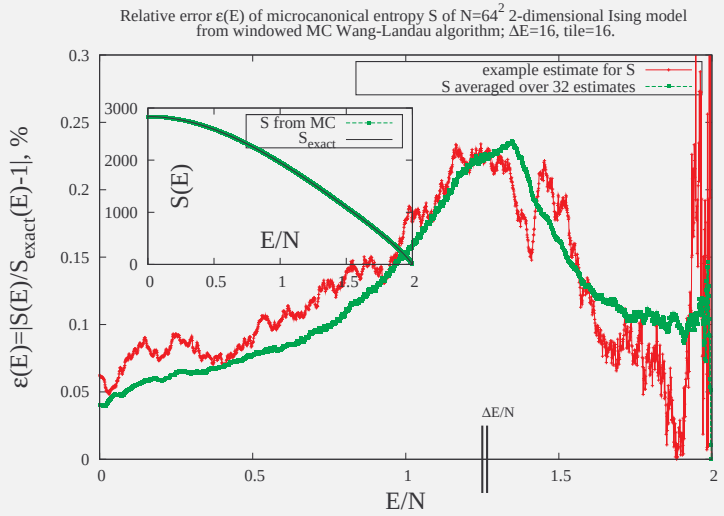
WL iteration

1. Start with $\omega(E) = 0 \; \forall E$.
2. Simulate "sufficiently long" while continuously updating $\omega(E)$.
3. Reduce modification factor, e.g., $\phi \to \phi/2$
4. Iterate till $\phi < \phi_{\text{thres}}$.

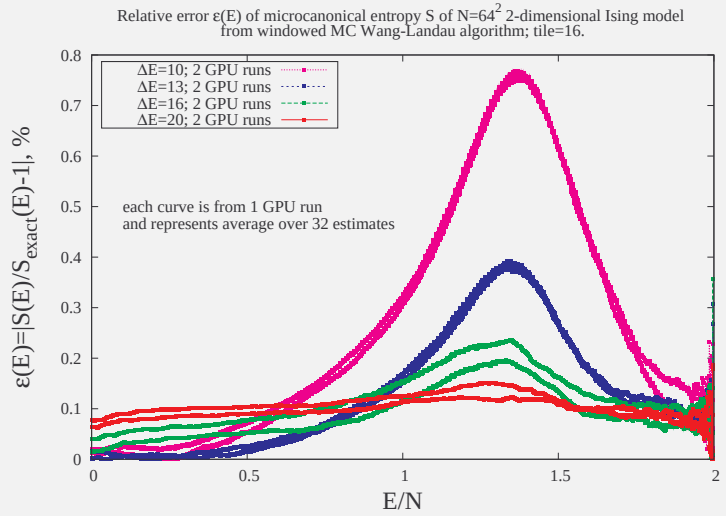
Wang-Landau simulations



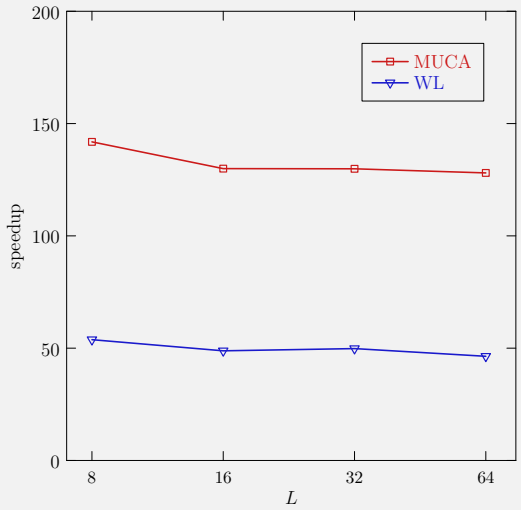
Wang-Landau simulations



Wang-Landau simulations



Wang-Landau simulations



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

All the rest

What I haven't talked about:

- no-copy pinning of system memory
- asynchronous execution and data transfers: streams etc.
- surface memory
- GPU-GPU communication
- C++ language features:
  - C++ new and delete
  - virtual functions
  - inline PTX
- Thrust library (similar to STL):
  - data structures: device_vector, host_vector, ...
  - algorithms: sort, reduce , ...
- libraries: FFT, BLAS,
- other language bindings: pyCUDA, Copperhead, Fortran, ...
- OpenCL

Summary and outlook

This lecture

In this lecture we have covered some more advanced simulation techniques. As expected, highly (data) non-local algorithms give significantly smaller speed-ups. Still, these are sizeable.

All lectures

Scientific computing with graphics processing units:

- tailoring to the hardware needed for really good performance
- special, but general enough to provide significant speed-ups for most problems
- GPU computing might be a fashion, but CPU computing goes the same way

Reading

For this lecture:

- M. Weigel, Phys. Rev. E 84, 036709 (2011) [arXiv:1105.5804].
- M. Weigel and T. Yavors'kii, Physics Procedia 15, 92 (2011) [arXiv:1107.5463].
- `http://www.cond-mat.physik.uni-mainz.de/~weigel/GPU`

Some advertisement: EPJST issue

`http://epjst.epj.org/articles/epjst/abs/2012/10/contents/contents.html`