Parallel Multigrid Ising Cluster Detection

Final Presentation

Parker Van Roy

EC526
Brower

Algorithmic Procedure

Major change in parallelization method:

Using only 4 bond array to cover both ½ power and natural powers

Reduced copying increased speed

~15% at N = 64

~25% at N = 256

$$\Delta_{ij}^{(l)} = \begin{cases} M_{ij} & \text{if } i-j \text{ is a distance } \pm 2^l \text{ in a coordinate direction} \\ 0 & \text{otherwise} \end{cases} \qquad (13)$$

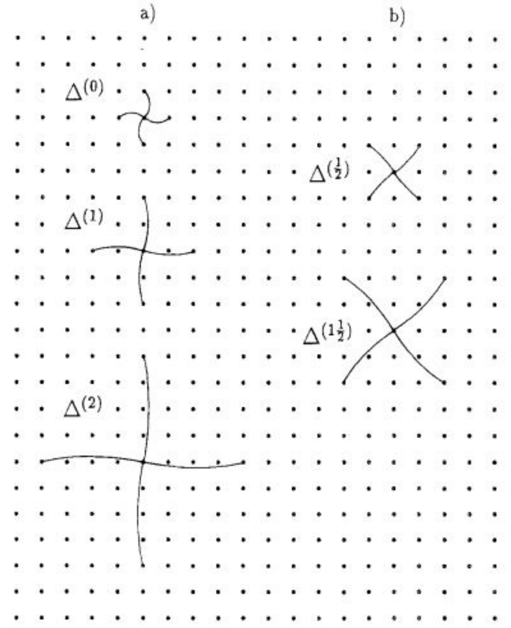


Fig. 1. (a) Multigrid connectivity matrices $\Delta^{(l)}$ connect neighbors at powers-of-two distances along coordinate axes. (b) The connectivity matrices for half-integer levels improve convergence by connecting neighbors along diagonals.

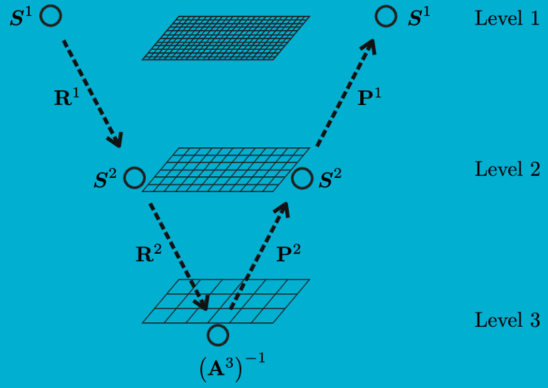
V-Cycle Multigrid

As proposed by Prof. Brower’s paper,

V-cycle multigrid was more effective than simply parallelizing loops at scale.

I found that V-Cycle all the way down to L=2 was most efficient up to L=512 (will test higher)

The bonds at L = 2 were hard coded to increase speed.



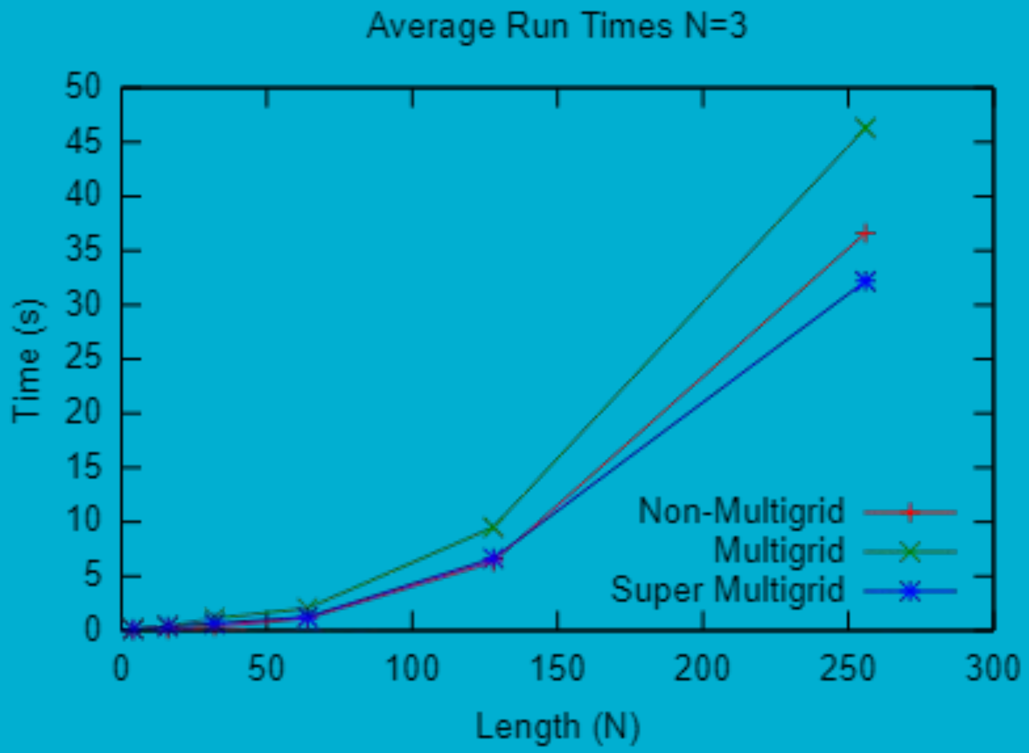
Timing

Base case tested at 256 with no multigrid:

11s O(0) time

Worst case runtime (force stop):

1m12s



Determining Relaxation

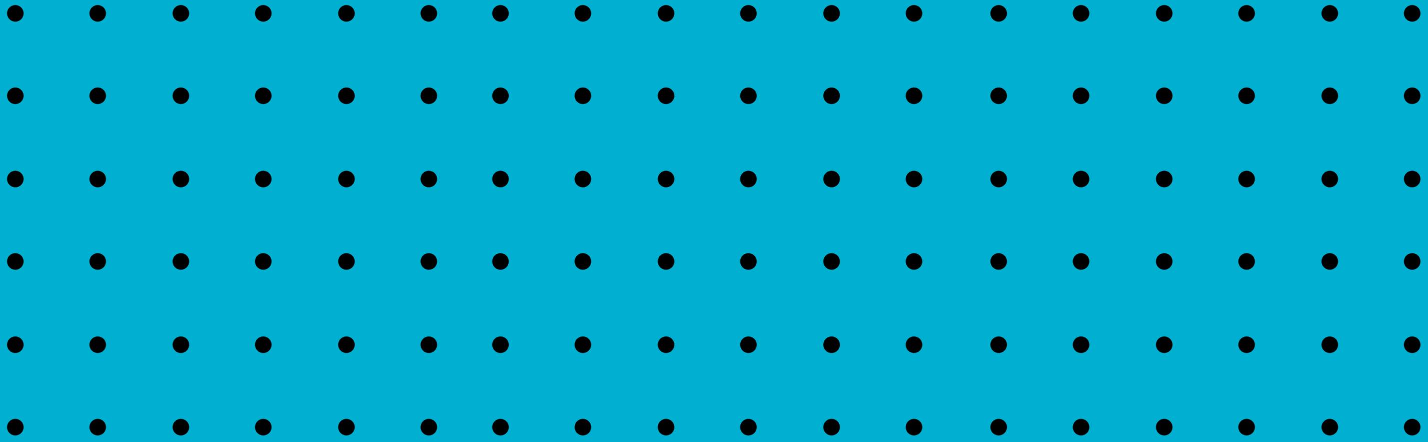
Setting relaxation to 1000 is arbitrary and will not necessarily work on large data sets

Max necessary iterations to relax a grid of L*L with k step is L*L/2k due to longest possible path taking ½ spaces on grid

In large applications, it is possible to reduce this iteration count based on probability AS LONG AS A FINAL STOPPING ITERATION IS RUN FOR GUARANTEE

Since this is a v-cycle, we can take log2(L*L/2(k+1)) or simply L works

Why isn’t more time saved!? Graph paper



Profiling Notes

The profiling shows 22% of cpu time in the ACC multigrid code is spent on the MultigridSW function. This is down heavily from 35% at mid project report and shows a relatively lower amount of the total time being used there.

On the gpu, 99% of time is spent on copy. There are portions that may be unnecessary to put on GPU, but for large applications it will almost always be optimal. Obviously there are always further optimizations to be made

Progress since Mid Project Report

O ½ step / diagonal step implementation

O Move relaxation regime into function

O Try recursive version?

X Move FlipSpins?

O Grid Visualization

X Implement OMP

~ Paper

$$\Delta_{ij}^{(l)} = \begin{cases} M_{ij} & \text{if } i-j \text{ is a distance } \pm 2^l \text{ in a coordinate direction} \\ 0 & \text{otherwise} \end{cases} \qquad (13)$$

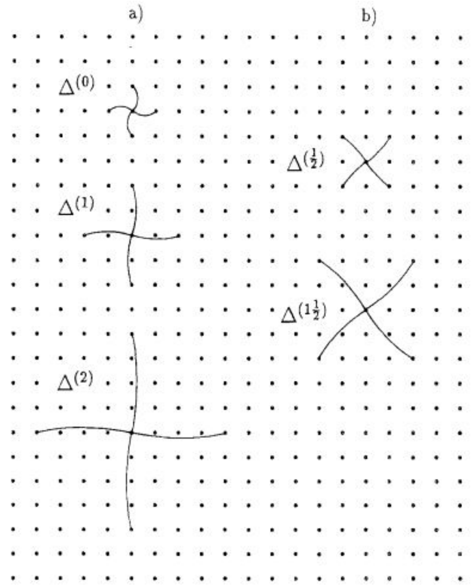
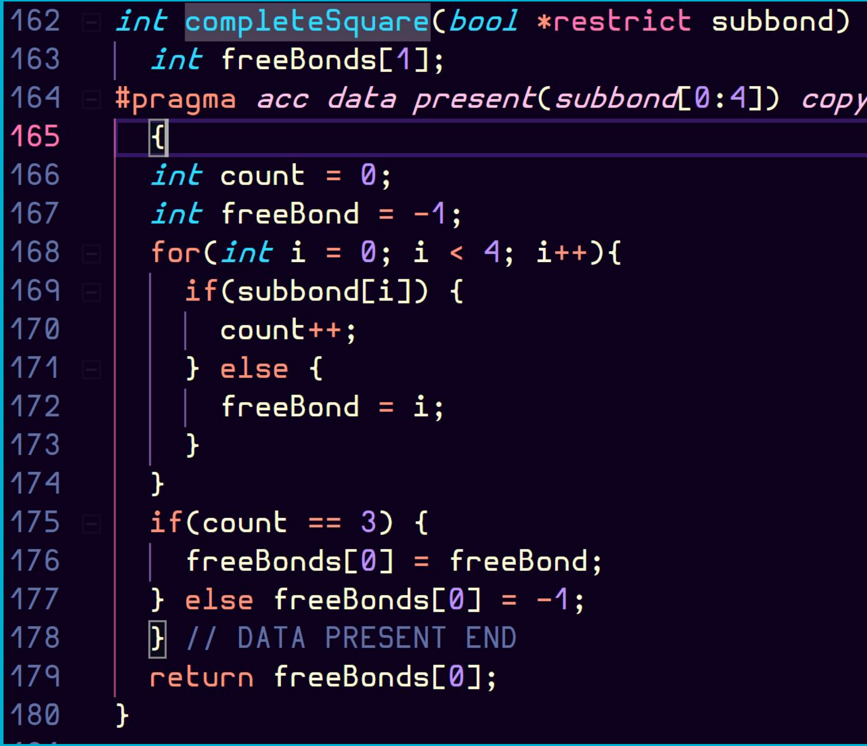


Fig. 1. (a) Multigrid connectivity matrices $\Delta^{(l)}$ connect neighbors at powers-of-two distances along coordinate axes. (b) The connectivity matrices for half-integer levels improve convergence by connecting neighbors along diagonals.

Bonus Optimizations!

One optimization I found to give the base code a 10% speed boost without any multigrid was a function to pathfind exact neighbors. The 'Super Multigrid' idea derived from expanding this after the original was performing slowly.



```c
int completeSquare(bool *restrict subbond)
  int freeBonds[1];
#pragma acc data present(subbond[0:4]) copy
  {
  int count = 0;
  int freeBond = -1;
  for(int i = 0; i < 4; i++){
    if(subbond[i]) {
      count++;
    } else {
      freeBond = i;
    }
  }
  if(count == 3) {
    freeBonds[0] = freeBond;
  } else freeBonds[0] = -1;
  } // DATA PRESENT END
  return freeBonds[0];
}
```

TODO Final Report

- Testing on much higher L (Fix fn)
- Some general optimizations on other parts of code
- Change array initialization to malloc or finish restricting everything
- Try kernels
- Write ups for number derivations
- Implement timer for function as opposed to code
- Play with gangs and vectors