Ising Model

High Performance Parallel Computing 2022

Daniel Hans Munk, Kimi Kreilgaard, Linea Stausbøll Hedemark & Moust Holmes

March 2022

1 Introduction

The Ising model is a mathematical representation of physical systems, where a collection of identical objects that can only occupy one of two possible states interact and influence each other. The interactions and general behaviour of the system depends on the dimensions of the problem.
In its simplest one dimensional form the Ising model is a string of two-state objects, for example magnetic dipoles with either spin up or spin down. Ernst Ising proved in 1925 that no phase transition is possible in this model in one dimension. Phase transition happens when all spins gradually flip and align themselves in the same direction. The spins align themselves according to their nearest neighbours with a preference for lowering the overall energy of the system. The objects in the system are occasionally disturbed by random thermal fluctuations. In one dimension the influence of neighbours is low and thus uniformity over more than a few dipoles is very unlikely and random fluctuations dominate. This results in the conclusion that there is no temperature threshold at which a steady state of full alignment is possible, except for absolute zero, which can not count as a transition temperature as there is per definition no temperature lower from which it can transition from.
The two dimensional square Ising model can be interpreted as a lattice, with each node in the lattice being a magnetic dipole oriented either up or down. The energy of the system is calculated as,

$$E = -J \sum_{<i,j>} \sigma_i \sigma_j - h \sum_i \sigma_i, \tag{1}$$

where J represents the spin-spin interaction and h represents the influence of an external magnetic field. Throughout our project we assume that there is no external magnetic field applied, meaning $h = 0$. Note in the spin-spin interaction, that we only sum over neighbouring pairs.
There exists an exact analytical solution to the two dimensional case, in which it is proven that phase transitions are possible, but its very technical, and often the numerical solution is much simpler to calculate. In a 2D lattice the number of nearest neighbours at a given dipole is of course bigger than that of a 1D string, and the increased influence of neighbours means that stable solutions are possible.
The fact that phase transition have been proven in the higher dimensional Ising models is significant in several areas of physics, most notably perhaps in explaining ferromagnetism. The 2D Ising model can be

used to illustrate how different temperatures influence materials, and how ferromagnetism occurs when all the "spins" in the model align themselves.

2 Implementing a Sequential Program

First we constructed a sequential program for the 2D Ising model which formed the foundation for later parallelising.
Here is an overview of the algorithm:

1. Initialize the Lattice, calculate the initial energy and magnetization and define constants.
2. Monte Carlo step: Pick random site, calculate probability for the site changes energy and change/no change depending on the calculation
3. Compute the new energy and magnetization
4. Repeat (2) and (3) until number of desired iterations has been achieved.

Explanation of Step (1): Firstly we created a 2D lattice which was a square grid of size $N \times N$ where each site has four neighbors and which were saved globally in our program using a public class. The lattice was initialized randomly with -1,+1 values by a random number generator. We then chose a coupling constant of $J = 1$ and $h = 0$ meaning we have zero external field and the system behaves *ferromagnetically*, the energy is minimized and order in the system is expected at low temperatures. Then we set the number of iterations as a global constant: *iter*.
We then created a function which calculates the initial energy and magnetization using (1). Since we our system must obey periodic boundaries we use modular arithmetic to wrap around the grid. Magnetization is just the sum of all the spins $\{-1, +1\}$ on the lattice.
Explanation of Step (2): When doing a Monte Carlo sampling we need detailed balance and ergodicity which means there is a balance of probabilities between any two states and ergodicty means any state can be visited within an infinite time frame. One way to ensure both of these requirements are fulfilled is by the Metropolis algorithm: First choose a random site uniformly on the lattice, then calculate the energy; E at the particular site with (1), then the energy-cost for a flip is: $\Delta E = -2 * E$. Then pick a uniformly random number $r \in [0, 1]$ and the Metropolis algorithm states that if: $\exp(-\Delta E/t) \geq r$, then the spin on the chosen site will flip.
Explanation of Step (3): We then use the function from step (1) which calculates the energy and magnetization and store each value in history vectors which are stored in the same public class as the lattice grid.

3 Parallelising the Program with Open MP

Before we can begin parallelising the program, we need to identify components of code that can run independently of each other. In the Ising model the state of one object is only dependent on its neighbours (four objects for the 2D lattice system). This means that we are able to update different regions of the

lattice simultaneously by using multiple threads. Since we are working with one lattice, we will use Open MP which has shared memory so all threads can update and access the same lattice, although they will each work in their own region. The lattice we are working with has dimensions $N \times N$ and is thus a square. For simplicity we choose to keep the threads regions square as well. We do this by implementing a `segmentation` parameter defining how many threads one side of the lattice will be divided into. The number of threads we run on is thus given by `segmentation`^{2}. This domain decomposition is illustrated in Fig. 1 below for a segmentation of 3 and therefore 9 threads. While most of the individual regions can be updated entirely separated from the other regions, there is an exception. Objects on the border of each region have neighbours from other regions, and we say that these are located in the boundary regions also illustrated in Fig. 1. This implies that we cannot have two neighbouring threads both updating objects in the boundary, since this can lead to updating an object before a neighbour has used the information for its own update. To solve this problem we instate a global signal vector that holds information about whether a given thread is working on a boundary or not. If a neighbour threads *is* working on a boundary, a point sampled on the boundary will simply be resampled until it is not on the boundary. To not undersample the boundary regions we only resample a point if neighbouring threads are working on boundaries. In the illustration, for example, thread number 0 can work independent of thread 8, and they therefore don't react to a signal for that thread. The pseudo-code explanation of this process can be found in Algorithm 1 below, which describes the Markov Chain Monte Carlo Metropolis update for one thread. To actually parallelise the process, the update function is placed inside an Open MP parallel scope.

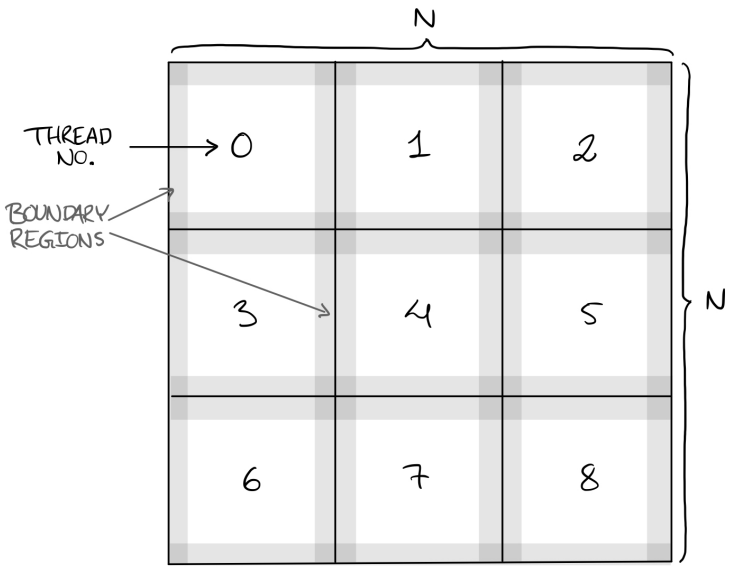

Figure 1: Domain decomposition for a $N \times N$ lattice distributed on 9 threads.

Algorithm 1 One Thread's Monte Carlo Simulation with Metropolis Criterion

```
Initialize signal vector to zeros
for every iteration do
    Sample a point within the workspace (a subsection of the lattice that only the given thread updates)
    if the sampled node is inside the boundary then
        update the thread-specific index in the global signal vector to 1 to notify other threads of work in boundary.
        if neighbouring threads are also in the boundary region then
            while the sampled node is inside the boundary do resample a new point within the workspace
            end while
            Change signal back to 0
        end if
    end if
    Calculate energy
    Calculate probability of acceptance
    if the proposed spin change is accepted then
        Update data in lattice
    end if
end for
```

4 Results

Running our 2D Ising Model program with the following parameters: Reduced Temperature: $t = 2.27$, Iterations: $n = 10 \times 10^5$, Burn in: $n_0 = 10 \times 10^4$ and seed 42 we got four results for different grid sizes $N \times N$ with $N = [5, 10, 15, 20]$ this can be seen in figure 2. For a small grid 5×5, the energy pr spin changes a lot since when a random site is flipped, the total energy changes an lot - meaning the system is very sensitive to change. For $N = 10$ this is still the case, however we already see a more well defined average which is calculated using equation (2) and as N increases the error on the variance decreases. This indicates that if one wants to have better precision with ones results then a bigger grid is better than a smaller grid. At the same time, for the model to converge with a bigger grid it also needs more time to run and the iteration number n therefore needs to be bigger as well. This is exactly why we have parallelised our model.

With the knowledge that bigger grid-size yields more precise results, we changed the parameters to Iterations: $n = 10 \times 10^6$, Burn in: $n_0 = 30 \times 10^4$, grid size: $N = 50$; $2500 = 50 \times 50$. Then we did 30 simulations where we changed the reduced temperature from $t_{start} = 0.1$ to $t_{end} = 5.0$ with even increments between all temperatures. The result is seen in figure 3 where the reduced critical temperature is shown, calculated from Onsager's result: $t_c = \dfrac{2J}{\ln(\sqrt{2}+1}$). The top left figure shows the average energy per spin which is calculated by (2), Top right shows the magnetization which is the absolute value of (3).

The lower left is the heat capacity per spin calculated with (4) and the lower left figure in figure 3 shows magnetic susceptibility per spin using (5).

We chose to do the grid-scan over the critical temperature to show the phase-transition of all the four values. Note there is an outlier: $t = 0.43$ which is clearly seen in the absolute average magnetization plot. We see the divergence of the heat capacity and magnetic susceptibility at the critical temperature which matches the theory: $C \propto \ln|\frac{1}{t-t_c}|$ when we are close to the critical temperature. We also know from theory that as temperature increases the spins disalign yielding absolute average magnetization equal to zero since there is no order in spin direction.

$$\frac{\langle E \rangle}{N \times N} = \frac{1}{N \times N} \frac{1}{n - n_0} \sum_{n > n_0} E_n \tag{2}$$

$$\frac{\langle M \rangle}{N \times N} = \frac{1}{N \times N} \frac{1}{n - n_0} \sum_{n > n_0} M_n \tag{3}$$

$$\frac{C}{N \times N} = \frac{(\langle E^2 \rangle - \langle E \rangle^2) t^{-2}}{N \times N} = \frac{\left(\frac{1}{N \times N} \frac{1}{n - n_0} \sum_{n > n_0} E_n^2 - (\frac{1}{(N \times N)} \frac{1}{n - n_0} \sum_{n > n_0} E_n)^2 \right) t^{-2}}{N \times N} \tag{4}$$

$$\frac{\chi}{N \times N} = \frac{(\langle M^2 \rangle - \langle M \rangle^2) t^{-1}}{N \times N} = \frac{\left(\frac{1}{N \times N} \frac{1}{n - n_0} \sum_{n > n_0} M_n^2 - (\frac{1}{(N \times N)} \frac{1}{n - n_0} \sum_{n > n_0} M_n)^2 \right) t^{-1}}{N \times N} \tag{5}$$

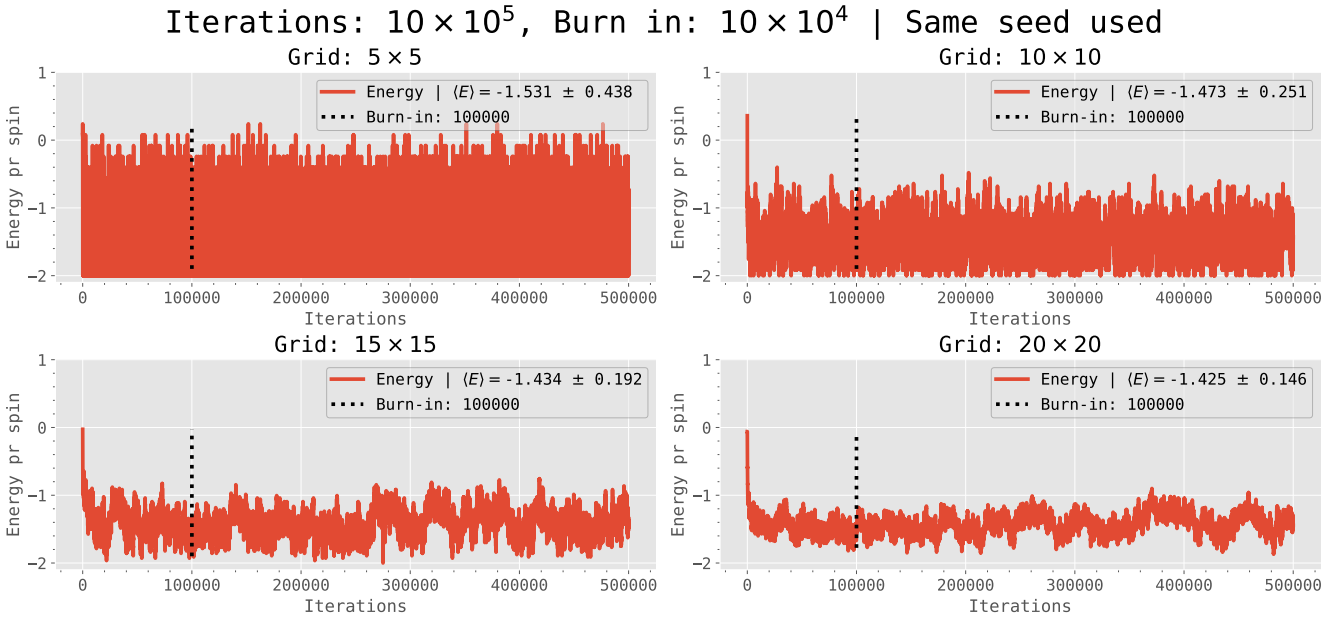

Figure 2: Four plots of different grid-sizes: $N = [5, 10, 15, 20]$ where the uncertainty on the average energy gets smaller when grid size is increased.

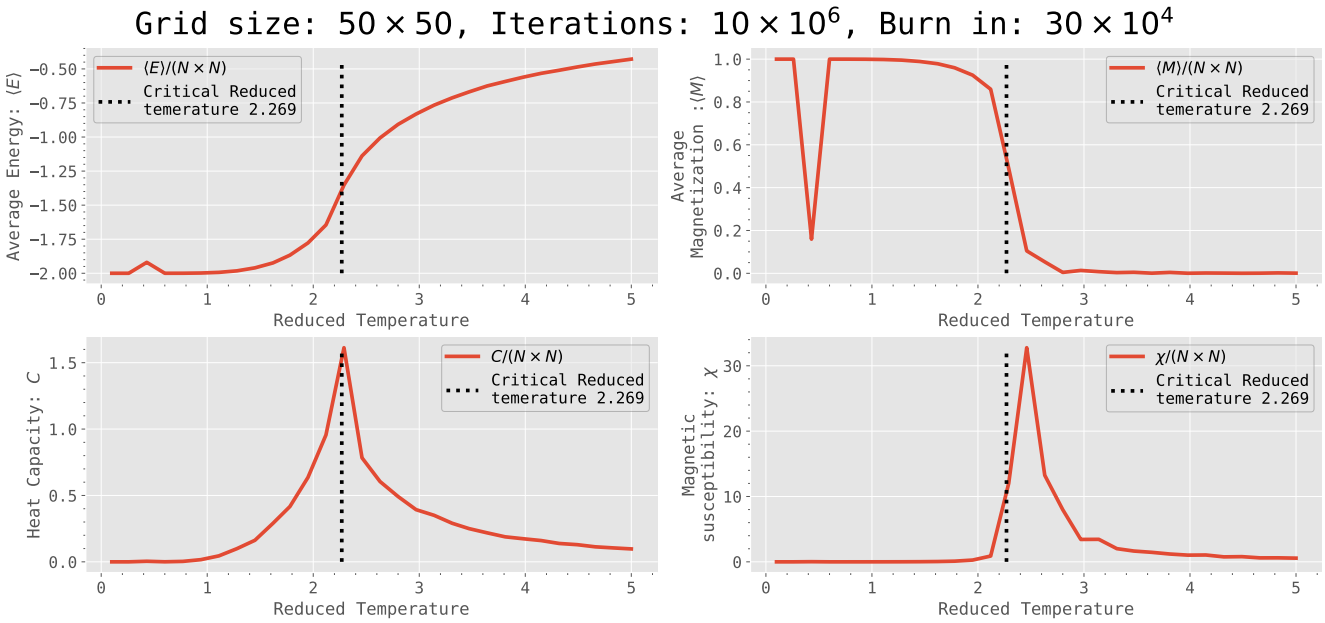

Figure 3: Four plots of average energy, average magnetization, heat capacity and magnetic susceptibility, all per spin. All plotted against reduced temperature to show phase transition at $t \sim 2.27$.

We did a strong scaling test of our program with the following parameters: Reduced Temperature: $t = 3.27$, Iterations: $n = 64 \times 10^4$ and grid size of 320×320. Since we partitioned our square lattice into 4, every number of processors/threads should be dividable by 4. The lattice size and iterations should also be divisible by 4. Also the number of iterations are divided by the number of processors since the program multiplies this number internally when doing the Monte Carlo steps. Bench-marking with the sequential version which took 172.7 seconds the relative speedup is shown in figure 4. There one can see that our code - following Amdahl's law has a parallel portion of $P \sim 92\%$ which is really good.

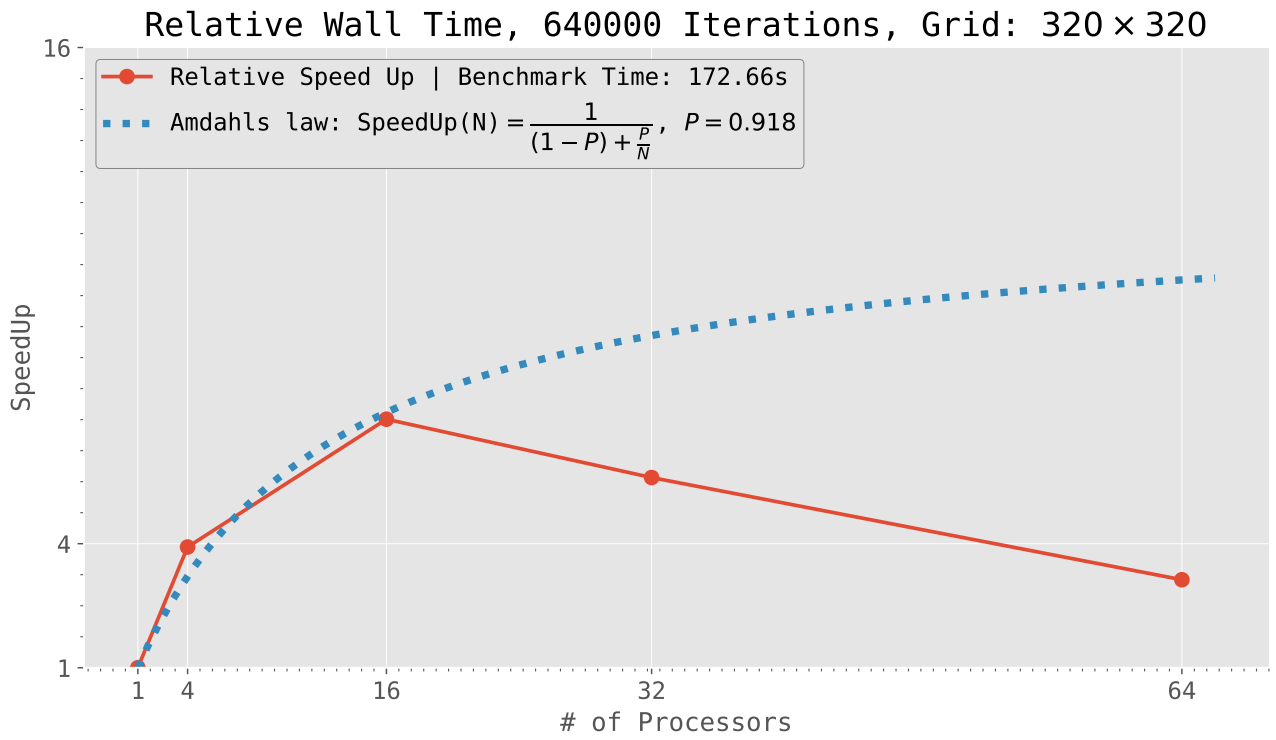

Figure 4: Strong scaling with increasing number of processors, size of problem stays consistent. The reduced temperature used are $t = 3.27$.

5 Discussion

Computational Note: After each Monte Carlo step we calculated the new energy and magnetization by summing over the full lattice. This is unnecessary and takes extra time when doing this approach. Instead one can calculate the new values by: $E_{new} = E_{old} + \Delta E$ and $M_{new} = M_{old} - 2 \times$ "*old spin value before flip*". The reason we did not use this approach was because we had a bug in this step of our code using that method, which was only fixed close to deadline. We kept the original calculations with the summation method instead of changing the new values inside the Monte Carlo step as we did not have time to do it over.

The error was due to how modulo works in c++. Then $i = 0$ and one wants to wrap round the matrix (periodic boundary) you can write "Lattice$[(i-1)\%N]$" however this gives Lattice$[-1]$ and this is not understood in c++ and we want Lattice$[N-1]$. A solution for this is: Lattice$[(N+i-1)\%N]$.

We can see in figure 4 that speedup does not quite follow the curve we would expect for strong scaling for runs where the number of processors exceed 16. A probable cause for this is how we handle the boundary cases. When the number of processors is small the size of each threads domain is large and the number of boundary nodes is small compared to the size of the domain. But as we increase the number of processors the percent of the domain that is boundary nodes increases. The reason why this makes the program slower is that the odds of two threads updating a node in the boundary increases and then

the two treads have to pick a new node to update which of course takes more time. The original method of handling the boundary cases which we were presented with was to use only one flag which kept track of whether ANY thread had chosen to update a node in the boundary. This method suffers from the same problem of the odds of having to resample increasing when the number of processors goes up. We worried that the first methods resample problem is so big that the under-sampling of the boundaries is so high that our program didn't model the Ising model correctly since an under-sampling of the boundary would inhibit the exchange of information between the threads domains which is required for a phase transition to be possible. This under-sampling is still present in our method but as long as the size of the tread domains doesn't get too small we'll argue that the problem isn't that big of a concern but it is something to keep an eye on when changing the number of threads and domain size. A potential method for minimizing this problem is to instead of having a vector keeping track of when a tread is sampling the boundary we could implement a vector that kept track of which side of the treads boundary is being sampled. Since a thread doesn't care if the boundary node is on the other side of its neighbor's domain this should decrease the number of resamples by approximately a factor 4.

The method of segmenting the domain into N equal pieces and then letting one thread update the one node each iteration is technically not equivalent to the sequential model. For the sequential model we would expect that the number of updates performed in a given region would be a binomial distribution with mean number of iterations divided by region size but for the parallel version the variance introduced by the binomial distribution isn't there. This variance of course becomes very small when we have a lot of iterations simply by the law of large numbers but it is worth noting that the sequential and the parallel versions are not equivalent in this regard.

Another very interesting method we found in literature is a so called "checkerboard" method where instead of assigning a domain to each thread you restrict which nodes the treads can update so that there is no nodes next to each other and then just let the treads freely choose a node to update (of course not the same). This restriction can be done in several ways for example by only choosing every third column in a row, every third row. This method circumvents many of the problems that the other methods have and scales much better with many processors since it only has to make sure that the treads doesn't choose the same points. Also the problem with under sampling the boundary is of course also gone since there is no boundary to be mindful of. The problem of the too uniform number of updates still exists but is much less apparent since the regions which have the exact same number of updates are disconnected and not squares like our method.

6 Conclusion

We have implemented a two dimensional square Ising model that runs sequentially and then parallelised the program using OpenMP pragmas and shown that the parallel program exhibits the characteristic behavior we would expect from the 2D Ising model such as phase transitions. Our method for parallelising the code works by segmenting the total domain into equal smaller sections where each thread is assigned to this smaller section and tasked with choosing and updating the nodes in its region. We use a signal vector which helps us avoid collisions when updating the spins so that when a thread has chosen a node

in the boundary region it tells its neighboring threads not to update a node in the boundary. Our method has some possible problems in perhaps under-sampling the boundary and overly uniform distribution of updates.
For further investigation it might be interesting to try to implement different way of handling the boundary cases and compare them to each other as well as their strong scaling behaviours.