3-Dimensional Heisenberg Model

Asbjørn Bonefeld Preuss Daniel Lomholt Christensen Elie Cueto
Frederik Aaboe Andersen Emilie Berg Nete Wen Yu O. Lyndrup

March 2024

Contents

Section 1 Introduction

This project seeks to implement the three-dimensional Heisenberg model into C++, and attempts to optimize the implementation, to allow simulations of larger systems, faster.

The project therefore starts with an Introduction to the Heisenberg model, followed by an overview of a sequential implementation, and then the parallelization that will be used to optimize the sequential implementation. Finally, a chapter showing the weak and strong scaling of the different versions, as well as some further ideas we did not have time to implement in this project.

1.1 The Heisenberg Model

The goal of the Heisenberg model is to predict the behavior of condensed matter systems. The Heisenberg model assumes an N×N×N cube, occupied by N×N×N spins. These spins are vectors that can point in any direction in 3D space. Each of these spins are generated as a random spin vector initially.

Then the Metropolis-Hastings Monte Carlo algorithm is used [1]. This algorithm selects a random site and flips it in a new, random direction. If the flip minimizes the energy of the system, it is accepted. If not, a flip probability is calculated. This probability is $P = \exp\left(-\frac{\Delta E}{k_B T}\right)$ according to Boltzmann statistic [2]. A random number is then generated. If the random number is smaller than the probability, the flip is accepted. Otherwise, the flip is rejected, and the system remains the same.

The energy of the system is calculated by the Hamiltonian considering the 6 nearest neighbours of the spin as well as the strength and direction of the magnetic field [2], cf. eq. (1).

$$H = -\frac{J}{2}\sum_{n,\lambda} \mathbf{S}_n \cdot \mathbf{S}_{n+\lambda} + g\mu_B \sum_n \mathbf{S}_n \cdot \mathbf{B}. \tag{1}$$

Here $g\mu_B$ are the gyromagnetic ratio and the Bohr magneton, which for the purpose of these calculations are set to unity. At each flip, the energy of the old spin and new spin must be calculated according to eq. (1).

Section 2 Sequential Implementation

The initial implementation first creates a class `spin_system`, most importantly containing a (standard) vector of vectors of positions and a vector of vectors of spins, which are both initially empty. Two generator functions are then called that create the positions of the spins and the spin vectors. After this, another generator finds the indices of each spin's neighbours in all directions.

Then the simulation begins. A random site is chosen (seeded by index of the currently running iteration), and the energy of the current system, as well as the old spin state, is recorded. The new spin direction is calculated, as well as put directly into the system of spins, and its energy is found - each spin is set to have six neighbours that influence the mentioned. The two energies are compared. If the new energy is lower, the spin-flip is accepted. If the new energy is higher, the

critical probability of acceptance is calculated. A random number (seeded by twice the iteration it is) is found. If the random number is larger than the critical probability, the change is not accepted and the old state of the spin is restored. This is done `flip` times, as requested in the command line. Our goal is to perform the simulation until convergence of the system (for temperatures below the critical temperature). We have applied a weak external B-field to the system that points in the z-direction. So we can check if the system has converged by checking that the magnetization of the system is parallel to the external B-field.

We expect the sequential version to be very slow for large system sizes since the single processor has to run through every single spin in the system for each time step in the simulation. Therefore, it is beneficial to exploit the potential parallelism in the problem and thereby speeding up the simulation which is the topic of the next sections.

Section 3 Parallelization

There are two versions of the parallelization of the program. In both versions, 3-dimensional domain decomposition with distributed memory is used, and the Message Passing Interface [3], `MPI`, is used for ghost cell communications between neighbouring domains. The 3D global system is divided into 3D local subsystems, and each rank is assigned to a local system. There is given a set number of flips to simulate the global system, where each local system has to run $\frac{flips}{\#ranks}$ number of iterations. All ranks start by exchanging the edge of their domain with their neighbours (these are the ghost cells) with a `MPI_Neighbor_alltoallw`. The exchange of ghost cells is necessary since the calculation of the energy difference for two different states uses the nearest neighbours of the spins. So when a spin is flipped at the edge of a local system, the globally adjacent spins which belong to another local system needs to be known.

In version 1, with each iteration, the ranks do the flipping in their local systems as in section 2 and regardless of whether something flipped or not, and regardless of whether the flipped spin is at the local domain's edge or not, the ranks will send an "entire wall" of ghost-cells to its neighbours with `MPI_Neighbor_alltoallw`. This works, but it is not the most efficient way to exchange ghost cells. This is because it is only when a spin somewhere on the edge of a local domain is flipped that the neighbour's ghost cells need updating. If a spin is flipped in the interior of a local domain or if a proposed flip at the edge is rejected, then the neighbouring domain will receive the same ghost cell values that it already has.

In version 2 we have implemented a different way to exchange ghost cells that does not have the inefficiency of version 1. With each iteration, a rank wants first to check if its neighbour has sent it anything, here `MPI_Iprobe` is called. If the flag is output from `MPI_Iprobe`, the rank shall use `MPI_Recv`, which is a blocking receive, such that it can update its ghost-cells. `MPI_Iprobe` is used to avoid putting a lot of non-blocking receives that may not need to receive as the edges of the neighbouring ranks do not flip often for bigger systems.

A rank can now choose a random spin and do the flipping as explained in section 2. If a flip is performed at the edge, the rank sends non-blocking – by `MPI_Isend` – *only* the information of the given changed spin to the neighbours' ghost cells - not the whole "wall" as in version 1, and only to the neighbours that need it. An `MPI_Barrier` is placed to ensure that the iteration is done before continuing to the next iteration. This is to prevent race conditions. Another barrier is also added

to ensure that all ranks are done with their job for the current time step before starting the next. This is done in both versions and is to prevent some ranks coming ahead of other ranks.

Lastly, for both versions when the simulation is done, all local systems are gathered on rank 0 with `MPI_Gather`. This way we end up with the global system.

Using this approach, it is possible to do as many flips at a time as there are ranks, in theory converging much faster even though an extra overhead is added.

Section 4 Results from simulations

We have run the simulation for multiple temperatures and two different system sizes ($\#spins = 729$ and $\#spins = 27000$). For this, we used version 2 of the parallel implementation with 27 processors. From this we obtain data for energy, magnetization, etc. This data is investigated in this section. As a cross check we have also plotted data from the sequential version of the code for the small system. It should be noted that the temperature is in all cases in units of the Boltzmann's constant, k_B.

4.1 Energy

We can find the total energy per spin from

$$E = \frac{\sum E_j}{\#spin}, \tag{2}$$

where j runs over each spin. In fig. 1, we have visualized the total energy per spin as a function of temperature. Our program outputs the energy of each spin for the last iteration where the program has reached convergence for temperatures below the critical temperature. As expected, we see the lowest energies for the lowest temperatures, and the energy gradually rises as the temperature rises. For higher temperatures, the energy is close to 0. This is expected since these temperatures are above the critical temperature meaning that the system cannot converge so here the spins point in random directions in the system which sets the energy to zero. Furthermore, we see that the data from the sequential version of the program follows the same trend as the parallel version within any fluctuations that is expected due to the stochastic nature of the spin flipping. This supports the correctness of our parallel implementation.

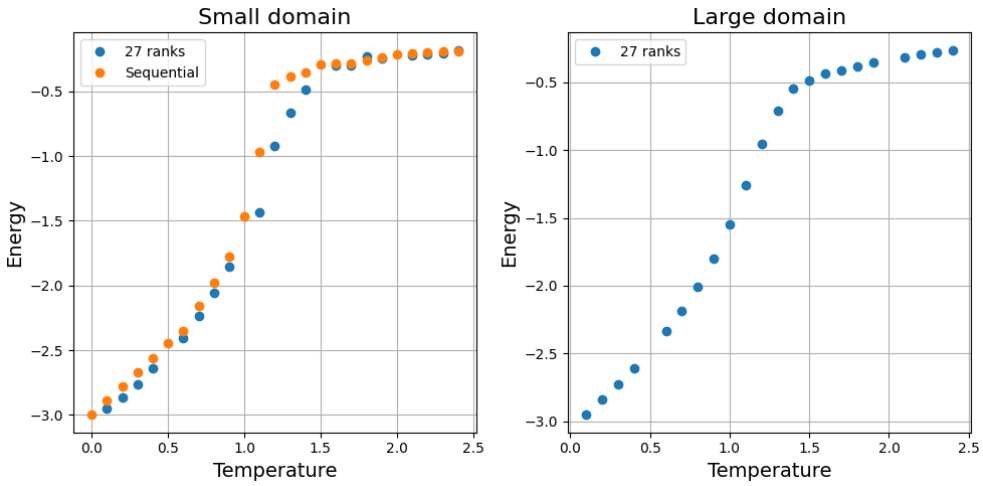

Figure 1: The total energy per spin in the last iteration of the simulation as a function of temperature.

4.2 Magnetization

The components of the magnetization vector for the whole system is found from the individual spins through

$$M^i = \frac{\sum S_j^i}{\#spin}, \tag{3}$$

where $i = x, y, z$ and j runs over each spin. We have also visualized the absolute value of the magnetization in the z-direction of the system as a function of temperature, which can be seen in fig. 2. For low temperatures, the magnetization in the z-direction is 1 corresponding to essentially all spins in the system pointing along the z-axis. For higher temperatures, this rises until it becomes zero meaning that all the spins point in random directions. Again, we see that the data from the sequential version follows the same trend as the data from the parallel version of the program as expected.

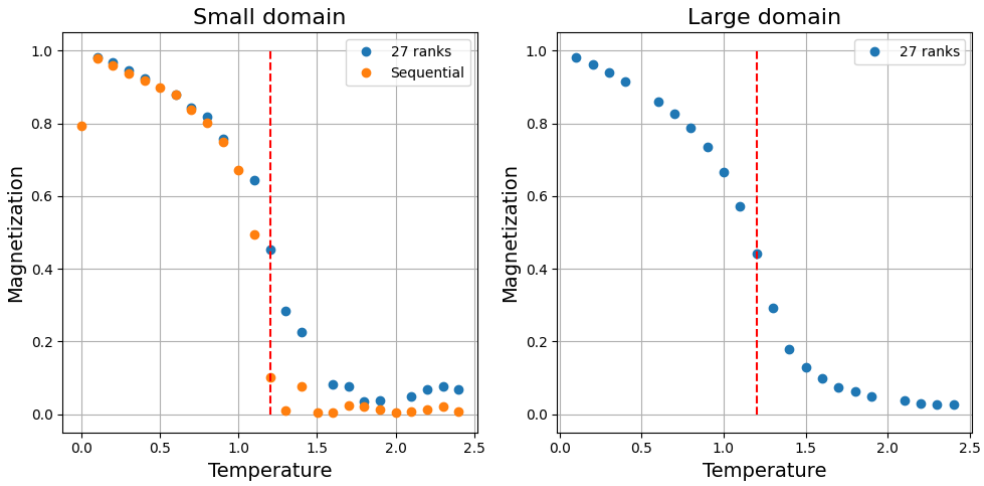

Figure 2: The absolute value of the total magnetization of the system along the z-axis in the last iteration of the simulation as a function of temperature.

4.3 Critical temperature

The critical temperature of the system is the temperature where the system no longer converges, where order is no longer observed. Since the external B-field applied to the system points in the z-direction, we can find the critical temperature by plotting the z-component of the magnetization of the system as has been done in fig. 2 and find the temperature at which the system changes from being ordered along the z-axis to being random. The approximate critical temperatures has been read off as $T_{crit} \approx 1.2$, and this is also visualized as a red dotted line in fig. 2. The critical temperature appears to be the same for the two system sizes even though the size difference between them ($\#spins = 729$ vs. $\#spins = 27000$) is quite large. This suggests that the critical temperature is independent of system size.

4.4 Heat capacity

From the energy of the individual spins, it is possible to obtain the heat capacity of the system given by

$$C_v \propto \frac{\langle E^2 \rangle - \langle E \rangle^2}{\#spin}. \tag{4}$$

Hence, we simply need to compute the mean of the squared energy of each spin and subtract the square of the mean energy of the spins and divide by the total number of spins in the system. We have done this for the different temperatures, and the result is plotted in fig. 3. It is evident that the heat capacity reaches a maximum just below the critical temperature for both system sizes. And also here, we see that the data from the sequential version of the program follows the same trend as the data from the parallel implementation as expected.

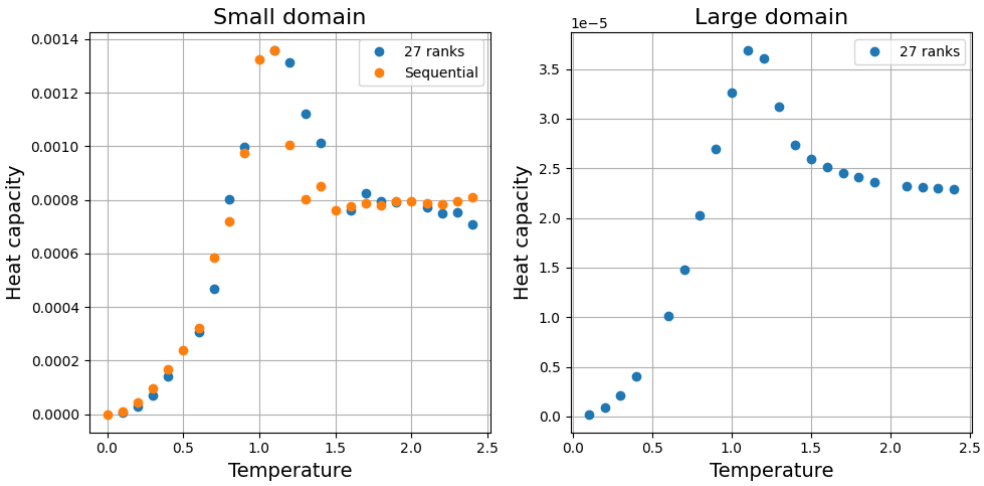

Figure 3: The heat capacity of the system in the last iteration of the simulation as a function of temperature.

Section 5 Scaling

We have investigated the strong and weak scaling of our parallel implementation. Hence, we have performed benchmarking of both versions of our parallel code and our sequential version in order to compute the speedup of each version of the parallel code compared to the sequential as a function of the number of processors used in the parallel versions. Hence, the speedup is given by

$$\text{Speedup}(N) = \frac{T_{sequential}(N)}{T_{parallel}}. \tag{5}$$

In strong scaling, the workload of the problem is kept fixed as the number of processors is increased. In weak scaling, the workload is scaled linearly to the number of processors.

For both strong and weak scaling we could only obtain four data points as our the sides of the world should be a cubic number and thereby so should the ranks, and we decided to limit ourselves to 1 node, both to be nice to our fellow students with whom we were sharing the 8 available nodes, and to avoid having to deal with how the rather large latency introduced by node-to-node communication would affect our scaling results, as we would only be able to have a single data point in 2, 4, 6, and 8 nodes with 5^3, 6^3, 7^3, and 8^3 ranks, which would make it a lot more difficult to interpret the results.

5.1 Strong scaling

To examine the strong scaling of our implementation, the code in appendix B was run for 1728000 flips and a system size of $\#spins = 1259712 = 108^3$ to ensure meaningful results. Since our code

runs in cubic blocks, the code was tested for 1, 8, 27 and 64 ranks. The resulting speedups are plotted in fig. 4.

To obtain the theoretical parallel fraction of the code according to strong scaling, we have fitted our data to Amdahl's law which is given by [4]

$$\text{Speedup}(N) = \frac{1}{S + P/N} = \frac{1}{1 - P + P/N}, \tag{6}$$

where N is the number of processors, P is the parallel fraction of the code and S is the sequential fraction of the parallel code. This too was displayed in fig. 4.

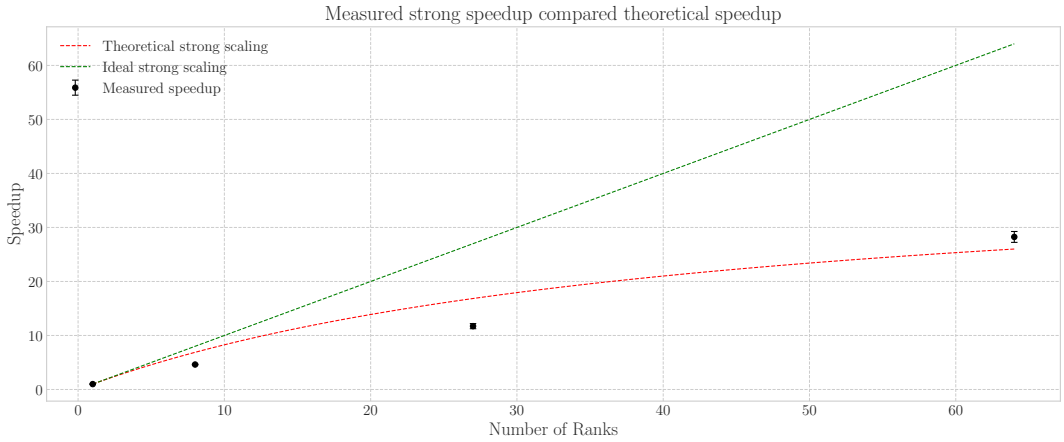

Figure 4: Plot of strong scaling with rank = {1, 8, 27, 64}. Rank = 1 is the sequential program. Only version 2 was examined for this strong scaling plot. Additionally, an ideal strong scaling was plotted for reference.

In fig. 4 we have plotted only version 2 of the parallelized program, due to the fact that version 1 took too long to run on MODI with everybody else. As version 1 ran for more than 5 minutes, it had to be placed in the `modi_short` partition, where it kept being removed in the middle of running in favour of jobs in the `modi_HPPC` partition, which made it nigh impossible to run version 1 at all, let alone get reliable timing results. This is thereby also a clear indication that version 1 achieves poor strong scaling, which aligns with the way the parallelization is done. As version 1 sends and receives a whole wall of ghost-cells at every iteration, a huge overhead is created when increasing the numbers of ranks. Version 1 of the parallelization does not lend well to strong scaling. Version 2 as seen in fig. 4 also is far from ideal strong scaling speedup, however does fair better for a high number of ranks - at least according it the theoretical strong scaling. Also, from the fit of eq. (6) we're able to determine the parallelized fraction of our version 2 code to be

$$P = 97.7\% \pm 0.5\%\,, \tag{7}$$

meaning that over 97% of the code benefits from increasing the computing resources while keeping the workload constant. Note, however, that the data points in fig. 4 do not very well align with the

fit to Amdahls law, suggesting that different kinds of overhead introduced by the parallelization play a significant role in how well the code speeds up as well.

To further cement just how inefficient version 1 of the code was, it ran for at least 10 minutes without finishing in the `modi_short` partition before it was interrupted, while version 2 of the code completed in about $2.5s$, $0.9s$ and $0.4s$ for 8, 27 and 64 cores respectively, and the sequential version completed in around $12s$.

5.2 Weak scaling

To examine the weak scaling of our implementation, the code in appendix B was tested for a constant system size of $\#spins = 1259712 = 108^3$ to ensure meaningful results. However, the number of flips was varied throughout. The flips were chosen such that running the code for 1, 8, 27 and 64 ranks lead to a linear scaling between workload and computing resources. Thereby, the numbers of flips chosen were: $\#flips = 10^7$, $8 \cdot 10^7$, $27 \cdot 10^7$ and $64 \cdot 10^7$. The resulting speedups are plotted in fig. 5.

To obtain the theoretical parallel fraction of our code according to weak scaling, we have fitted our data to Gustafson's law given by [4]

$$\text{Speedup}(N) = N - S(N-1) = P(N-1) + 1\,, \tag{8}$$

where N is the number of processors and S is the sequential fraction of the parallel code. Likewise, this was displayed in fig. 5. Hence, we expect the speedup to scale linearly with the number of processors for weak scaling.

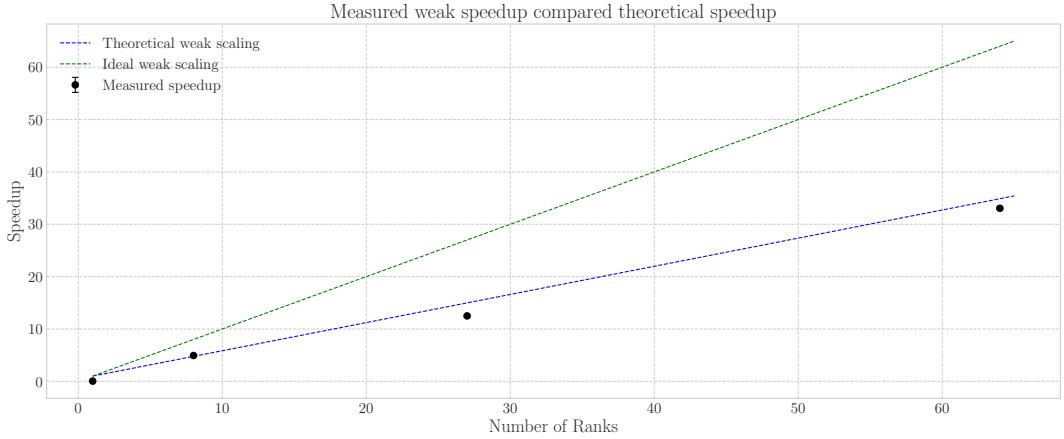

Figure 5: Plot of weak scaling, with rank {1, 8, 27, 64} with flips = $\{10^7, 8 \cdot 10^7, 27 \cdot 10^7$ and $64 \cdot 10^7\}$. Here we varied the number of flips such that it scaled with the number of ranks. Only version 2 was examined for this weak scaling plot. Additionally, an ideal weak scaling was plotted for reference.

In fig. 5 we see that at more flips and higher number of ranks obtain a better speedup for version 2 of the implementation. This aligns with our expectations, as in version 2, as a rank sends one single spin to its neighbours and it only does so when a spin is flipped, not with every iteration, which should lower the overhead that comes with sending and receiving compared to version 1. It should be noted that version 1 of the parallelization fares worse than the sequential program and is therefore not plotted. Version 1 being slower might stem from the immense overhead that is created in sending ghost-cells in a relatively small system.

However, we are able to determine the parallelized fraction of our code from fitting eq. (8) to the speedups measured for version 2 to be:

$$P = 54\% \pm 4\% \tag{9}$$

This shows us how about 50% of the code benefits from increasing the workload proportionally to the computing resources.

Section 6 Further ideas

Due to time constraints not all thinkable optimizations were implemented into the code, and we have had a few ideas which could have helped speed up our implementations even further.

One of these includes using openMP to do the sending and receiving of ghost cells between neighbouring ranks in our second version. By using hybrid MPI & openMP, and using the `MPI_THREAD_MULTIPLE` thread level, we could check for messages from, and send messages to, multiple neighbouring ranks at once. This would significantly reduce the overhead introduced by having to check for incoming messages in each iteration, potentially by up to a factor of six if given enough threads. It would also to a smaller degree reduce the overhead introduced by having to send messages to other ranks, as up to three messages potentially have to be sent at once if the flipped spin is in the corner of its domain.

OpenMP would also allow for doing some calculations asynchronously in each iteration. After finding the spin to be flipped, multiple different operations could be carried out at once: Determining the current energy in the cell, generating a new direction to be flipped to, and checking if the spin is on an edge so it would potentially need to be sent to neighbours could all be carried out simultaneously with multithreading.

For the initial setting up of the systems, like generating initial spins and deriving indices of neighbouring cells, vectorization with `omp simd` pragmas might also help, but this would be a relatively minor gain.

For the first version of the code, which sends all ghost cells between all neighbours after every iteration, it could also grant a significant boost in performance to successfully make the `MPI_Neighbor_alltoallw` send all the spins in all three directions in one communication instead of the current splitting into three which is extremely inefficient. Our attempts at this were unfortunately unsuccessful, and in any case the second version of the code is going to be more efficient, so we decided not to spend too much time trying to get this to work *(lie, we spent way too much time trying in vain to get it working)*.

It might also be imagined that updating only a single spin in each iteration is not the most effective way to go about things. If multiple spins could be updated per iteration, the amount of neighbour

communication could be reduced significantly. If multiple spins could be updated in parallel with multithreading, the amount of time spent per iteration could further be reduced. We could figure out a way of updating multiple spins at once, such that two neighbouring spins in the same domain could not be picked at once, as this would lead to race conditions when updating them. The number of simultaneous update would have to depend on domain sizes, as smaller domains don't have as much space for updating multiple independent spins at once, which might improve the weak scaling capabilities of the code. If we could update five spins in each iteration in this way, we could also decrease the number of communications with neighbours by a factor of five. We have found however, that hybrid openMP & MPI is not a beast one can easily conquer in a week, so we leave it to the reader to imagine how much speedup we could potentially obtain this way.

Section 7 Conclusion

In this project we have implemented a model of the three dimensional Heisenberg model into C++, and we have used MPI to optimize the model by splitting the system into regions that can run in parallel. We have found that a typical ghost-cell exchange between neighbouring regions at the end of each iteration is a very inefficient way to parallelize this model. Instead we find that sending individual ghost-cells between neighbouring ranks only when they are actually updated makes for a much more efficient code, which scales very well with both weak and strong scaling. Even so, we still find that the overhead introduced by the communication between ranks is significant, so finding a way to use openMP to parallelize the sending and receiving of messages, or to decrease the amount of needed communication by flipping several spins per iteration, would likely be an effective next optimization step.

References

[1] K. Sneppen and J. O. Haerter, *Complex Physics*. Sep. 4, 2023, 316 pp.

[2] S. H. Simon, *The Oxford Solid State Basics*, 1st ed. Oxford: Oxford University Press, 2013, 290 pp., ISBN: 978-0-19-968077-1.

[3] Message Passing Interface Forum, *MPI: A message-passing interface standard version 4.1*, Nov. 2023. [Online]. Available: `https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf`.

[4] R. Robey and Y. Zamora, *Parallel and High Performance Computing*. Shelter Island: Manning, 2021, 667 pp., ISBN: 978-1-61729-646-8.

Section A Source Code parallel Neighbor_alltoallw

```
1  #include <vector>
2
3  #include <iostream>
4  #include <iomanip>
5
6  #include <fstream>
7  #include <chrono>
8  #include <cmath>
9  #include <fstream>
10 #include <cstdlib>
11 #include <mpi.h>
12
13 int mpi_size;
14 int mpi_rank;
15 int nproc_x = 4, nproc_y = 4,nproc_z=4;
16 enum {ghost_cell_request, ghost_cell_answer};
17
18 bool verbose = false;
19 class spin_system {
20     public:
21     int flips = 100; // Number of flips the system will simulate.
22     int n_spins = 64; // Number of spins in The system.
23     int n_dims = 3; // Number of dimensions the spins are placed in.
24     float N_spins_row; // Number of rows in the square/cube
25
26     int x_offsets[6] = {-1,0,0,1,0,0}; // For calculating neighbor
           indices in energy calculation
27     int y_offsets[6] = {0,-1,1,0,0,0};
28     int z_offsets[6] = {0,0,0,0,-1,1};
29
30     int x_dist = 1;
```

```
31      int y_dist = 1;
32      int z_dist = 1;
33
34      int nearest_neighbours = 1; // Number of nearest neighbour
            interactions to be calculated
35      int xlen, ylen, zlen;
36      double J = 1; // Magnetization parameter, used in calculating
            energy.
37      double H; // Total energy of the system;
38      double B = 0; // Magnetic field in z direction
39      double Temperature = 1; // Temperature of the system.
40      std::string filename = "parallel_out.txt"; // Output file.
41      int write = 1;
42      std::vector<std::vector<double>> position; // Three by n_spins
            matrix, defining the spin's 3d position.
43      std::vector<std::vector<double>> spin; // Three by n_spins matrix,
            defining the spin vector for each spin.
44      std::vector<std::vector<int>>    neighbours; // 2*n_dims by
            n_spins matrix, defining the neighbour indices of each cell, so
            they need only be calculated once.
45      std::vector<double>              energy; // Energy in each cell,
            derivaed at the end.
46      spin_system(std::vector <std::string> argument){
47      for (long unsigned int i = 1; i<argument.size() ; i += 2){
48              std::string arg = argument.at(i);
49              if(arg=="-h"){ // Write help
50                  std::cout << "Heisenberg simulation\n --flips <number
                        of flips performed>\n --nspins <number of spins
                        simulated>\n --ndims <number of dimensions to
                        simulate> \n"
51                            << " --ofile <filename>\n --magnet <strength
                                of external magnetic field in z
                                direction>\n"
52                            << " --temp <temperature> \n" << "
                                --writeout <write to data file (1 for
                                true, 0 for false)>\n";
53
54                  exit(0);
55                  break;
56              } else if(arg=="--flips"){
57                  flips = std::stoi(argument[i+1]);
58              } else if(arg=="--nspins"){
59                  n_spins = std::stoi(argument[i+1]);
60              } else if(arg=="--ndims"){
61                  n_dims = std::stoi(argument[i+1]);
62              } else if(arg=="--ofile"){
```

```
63                  filename = argument[i+1];
64              } else if(arg=="--magnet"){
65                  B = std::stoi(argument[i+1]);
66              } else if(arg=="--temp"){
67                  Temperature = std::stod(argument[i+1]);
68              } else if(arg=="--writeout"){
69                  write = std::stoi(argument[i+1]);
70              } else{
71                  std::cout << "-----> error: the argument type is not
                        recognized \n";
72              }
73          }
74      N_spins_row = cbrt(double(n_spins)); //Equal size in all dimensions
75      int n_spins_row = round(N_spins_row);
76      xlen = n_spins_row;
77      ylen = n_spins_row;
78      zlen = n_spins_row;
79      if(verbose) std::cout << "Nspins row " << n_spins_row << std::endl;
80      }
81  };
82
83  class local_spins{
84      public:
85      int x_offsets[6] = {-1,0,0,1,0,0}; // For calculating neighbor
            indices in energy calculation
86      int y_offsets[6] = {0,-1,1,0,0,0};
87      int z_offsets[6] = {0,0,0,0,-1,1};
88
89      int x_dist,y_dist,z_dist;
90
91      int nearest_neighbours; // Number of nearest neighbour
            interactions to be calculated
92      int xlen, ylen, zlen;
93      int pad_xlen, pad_ylen, pad_zlen;
94      int offset_x, offset_y, offset_z;
95      int n_spins;
96      double J; // Magnetization parameter, used in calculating energy.
97      double H; // Total energy of the system;
98      double B; // Magnetic field in z direction
99      double Temperature; // Temperature of the system.
100     std::string filename; // Output file.
101
102     int no_in_padded_layer, no_in_layer;
103
104     local_spins(spin_system &sys,
105             int local_xlen, int local_ylen, int local_zlen,
```

```
106             int offx, int offy, int offz){
107         x_dist = sys.x_dist;
108         y_dist = sys.y_dist;
109         z_dist = sys.z_dist;
110 
111         nearest_neighbours = sys.nearest_neighbours;
112         xlen = local_xlen;
113         ylen = local_ylen;
114         zlen = local_zlen;
115 
116         pad_xlen = xlen+2;
117         pad_ylen = ylen+2;
118         pad_zlen = zlen+2;
119 
120         n_spins = xlen*ylen*zlen;
121         no_in_layer = xlen*ylen;
122         no_in_padded_layer = (xlen+2)*(ylen+2);
123 
124         offset_x = offx;
125         offset_y = offy;
126         offset_z = offz;
127         J = sys.J;
128         H = sys.H;
129         B = sys.B;
130         Temperature = sys.Temperature;
131         filename = sys.filename;
132     };
133     std::vector<std::vector<double>> position; // Three by n_spins
        matrix, defining the spin's 3d position.
134     std::vector<std::vector<double>> spin; // Three by n_spins matrix,
        defining the spin vector for each spin.
135     std::vector<std::vector<int>>    neighbours; // 2*n_dims by
        n_spins matrix, defining the neighbour indices of each cell, so
        they need only be calculated once.
136 
137 
138     int index_to_padded_index(int index){
139 
140         int x = index%(no_in_layer)%xlen + 1;
141         int y = (index%(no_in_layer))/xlen + 1;
142         int z = index/(no_in_layer) + 1;
143 
144         return z*no_in_padded_layer + y*pad_xlen + x;
145     }
146 
147     int padded_index_to_index(int index){
```

```
148         int x,y,z;
149         padded_index_to_padded_coordinates(index,x,y,z);
150         x -= 1;
151         y -= 1;
152         z -= 1;
153         return z * xlen * ylen + y * xlen + x;
154     }
155     void padded_index_to_padded_coordinates(int index, int& x, int& y,
        int& z){
156         x = index % pad_xlen; // Which row the spin is in
157         y = (index/pad_ylen)%pad_ylen; // Which column the spin is in
158         z = index / no_in_padded_layer;
159     }
160
161     void index_to_coordinates(int index, int& x, int& y, int& z){
162         x = index % xlen;
163         y = (index/ylen)%ylen;
164         z = index / (ylen * xlen);
165     }
166     void padded_coordinates_to_padded_index(int &index,int x, int y,
        int z){
167         index = x%pad_xlen + (y%pad_ylen) * pad_xlen + (z%pad_zlen) *
            no_in_padded_layer;
168     }
169
170     void padded_index_to_global_index(int p_index, int &g_index,int
        global_x, int global_y, int global_z){
171         int local_index = padded_index_to_index(p_index);
172         int x,y,z;
173         index_to_coordinates(local_index,x,y,z);
174         x += offset_x+1;
175         y += offset_y+1;
176         z += offset_z+1;
177         g_index = x%global_x + (y%global_y) * global_x + (z%global_z)
            * global_x * global_y;
178         //if(mpi_rank==3)std::cout <<mpi_rank<<" "<< p_index<<"
            "<<g_index<<" " << x<<" "<<y<<" "<<z<<"\n";
179
180     }
181
182     void global_index_to_padded_index(int g_index, int &p_index, int
        global_x, int global_y, int global_z){
183         // Find global coordinaetes
184         int g_x = g_index % global_x;
185         int g_y = (g_index/global_y)%global_y;
186         int g_z = g_index / (global_y * global_x);
```

```
187
188         // Define padded coords
189         int x,y,z;
190
191         if(g_x==0&&offset_x==global_x-xlen-1){x=pad_xlen-1;} //If on
                the left / bottom / back edge, of global, and local is on
                the right / top / front edge, set to be on the right / top
                / front edge
192         else{
193         x = (g_x -(offset_x)); // Convert from global to padded index
194         if(x==global_x) x=0; // if on the right / top / front edge,
                set to be on the left / bottom / back edge.
195         x = (x + pad_xlen)%pad_xlen; // Makes sure is non-negative
196         }
197         if(g_y==0&&offset_y==global_y-ylen-1){y=pad_ylen-1;}
198         else{
199         y = (g_y -(offset_y));
200         if(y==global_y) y=0;
201         y = (y + pad_ylen)%pad_ylen;
202         }
203         if(g_z==0&&offset_z==global_z-zlen-1){z=pad_zlen-1;}
204         else{
205         z = (g_z -(offset_z));
206         if(z==global_z) z=0;
207         z = (z + pad_zlen)%pad_zlen;
208         }
209         padded_coordinates_to_padded_index(p_index,x,y,z);
210         //if(mpi_rank==1)std::cout <<mpi_rank<<" "<<std::setw(3)<<
                g_index<<" "<<std::setw(3)<<p_index<<" " << x<<" "<<y<<"
                "<<z<<" " << offset_x<<" "<<offset_y<<" "<<offset_z<<"\n";
211     }
212
213     void recv_index_to_padded_index(int r_index, int dir, int p_index){
214         int x,y,z;
215         if(dir == 4) { x = 1; y = r_index%pad_ylen; z =
                r_index/pad_ylen;}
216         if(dir == 5) { x = xlen; y = r_index%pad_ylen; z =
                r_index/pad_ylen;}
217         if(dir == 2){  y = 1; x = r_index%pad_xlen; z =
                r_index/pad_xlen;}
218         if(dir == 3){  y = ylen; x = r_index%pad_xlen; z =
                r_index/pad_xlen;}
219         if(dir == 0){  z = 1; x = r_index%pad_xlen; y =
                r_index/pad_xlen;}
220         if(dir == 1){  z = zlen; x = r_index%pad_xlen; y =
                r_index/pad_xlen;}
```

```
221             padded_coordinates_to_padded_index(p_index,x,y,z);
222         }
223         void padded_index_to_send_index(int p_index, int dir, int s_index){
224             int x,y,z;
225             padded_index_to_padded_coordinates(p_index, x, y, z);
226             if(dir == 4 || dir == 5) {s_index = y + z * pad_ylen;}
227             if(dir == 2 || dir == 3) {s_index = x + z * pad_xlen;}
228             if(dir == 0 || dir == 1) {s_index = z + y * pad_xlen;}
229         }
230     };
231
232
233     // Function that generates rectangular positions for alle the spins in
        the system,
234     void generate_positions_box(local_spins &sys){
235             for (double k=0; k<sys.pad_zlen; k++)
236             for (double j=0; j<sys.pad_ylen; j++)
237             for (double i=0; i<sys.pad_xlen; i++)
238                 sys.position.push_back({double(i*sys.x_dist),
                        double(j*sys.y_dist), double(k*sys.z_dist)});
239     };
240
241     // Function that generates random directions for all the spins in the
        system
242     void generate_spin_directions(local_spins &sys){
243
244         for (int i = 0; i<sys.pad_zlen*sys.no_in_padded_layer; i++){
245             srand(i); // Seed is here to make it perform nicely when
                comparing to parallel
246             double spin_azimuthal = (double) rand()/RAND_MAX * M_PI;
247             srand(i*rand()); // Seed is here to make it perform nicely
                when comparing to parallel
248             double spin_polar = (double) rand()/RAND_MAX * 2. * M_PI;
249
250             sys.spin.push_back({sin(spin_azimuthal)*cos(spin_polar),
251                                 sin(spin_azimuthal)*sin(spin_polar),
252                                 cos(spin_azimuthal)});
253         }
254     };
255
256     void generate_neighbours(local_spins &sys){
257
258         for (int spin = 0; spin< sys.pad_zlen*sys.no_in_padded_layer;
            spin++){
259             // Find position in square / cube
260             int spin_x, spin_y, spin_z;
```

```
261         sys.padded_index_to_padded_coordinates(spin, spin_x, spin_y,
                spin_z);
262
263         // Find indices of neighbours
264         std::vector<int> spin_interactions;
265         for(int i = 0; i < 6; i++){
266             spin_interactions.push_back((spin_x +
                    sys.x_offsets[i])%sys.pad_xlen +
267                                         (spin_y +
                                            sys.y_offsets[i])%sys.pad_ylen
                                            * sys.pad_xlen +
268                                         (spin_z +
                                            sys.z_offsets[i])%sys.pad_zlen
                                            * sys.no_in_padded_layer);
269             //std::cout<< "RANK: " << mpi_rank << ". neighbour " << i
                    << " Of padded local index " << spin << " is " <<
                    spin_interactions[i] << std::endl;
270         }
271         sys.neighbours.push_back(spin_interactions);
272     }
273 }
274 // Function that calculates the energy of a single spin in 2d
275 double energy_calculation_nd(local_spins &sys, int spin, MPI_Comm&
        cart_comm){
276     double energy = 0;
277     double dot_product;
278
279     for (int i=0; i<6; i++){
280         // Calculate the energy with the nearest neighbour with no
                corners
281         dot_product =
                sys.spin[spin][0]*sys.spin[sys.neighbours[spin][i]][0]
282                         + sys.spin[spin][1]*
                              sys.spin[sys.neighbours[spin][i]][1]
283                         + sys.spin[spin][2]*
                              sys.spin[sys.neighbours[spin][i]][2];
284         energy -= sys.J/2*dot_product;
285     }
286     energy += sys.B*sys.spin[spin][2];
287     return energy;
288 };
289
290 // Calculate the total energy of the system
291 void Calculate_h(local_spins& sys, MPI_Comm cart_comm){
292     sys.H = 0; // Set H to zero before recalculating it
293     double mag_energy = 0;
```

```
294     for (int i=0; i<sys.n_spins; i++){
295         int pad_i = sys.index_to_padded_index(i);
296         sys.H += energy_calculation_nd(sys, pad_i, cart_comm)*0.5; //
                Half the energy, because we calculate on all the spins
297         mag_energy += sys.spin[pad_i][2];
298     }
299     sys.H += sys.B*mag_energy * 0.5; // Half of the magnetization
            energy is removed above
300 };
301
302 // Write the spin configurations in the output file.
303 void Writeoutput(spin_system& sys, std::ofstream& file, MPI_Comm
        cart_comm){
304     // Loop over all spins, and write out position and spin direction<
305     file << "Position_x-" << "Position_y-" << "Position_z-" << "Spin_x-
            " << "Spin_y-" << "Spin_z-" << "Spin_energy-" <<
            "Temperature-" << "n_spins" << std::endl;
306     for (int i = 0; i<sys.n_spins; i++){
307         if (i == 0) {
308             file << sys.position[i][0] << "-" << sys.position[i][1] <<
                    "-" << sys.position[i][2] << "-"
309             << sys.spin[i][0] << "-" << sys.spin[i][1] << "-" <<
                    sys.spin[i][2] << "-" << sys.energy[i]
310             << "-" << sys.Temperature << "-" << sys.n_spins
311             << std::endl;
312         } else {
313             file << sys.position[i][0] << "-" << sys.position[i][1] <<
                    "-" << sys.position[i][2] << "-"
314             << sys.spin[i][0] << "-" << sys.spin[i][1] << "-" <<
                    sys.spin[i][2] << "-" << sys.energy[i]
315             << std::endl;
316         }
317     }
318 };
319
320
321 void exchange_ghost_cells(local_spins &local_sys,
322                           MPI_Aint &sdispls, MPI_Aint &rdispls,
323                           MPI_Datatype &sendtypes, MPI_Datatype
                              &recvtypes,
324                           MPI_Comm cart_comm){
325     int counts[6] = {1,1,1,1,1,1};
326
327     // Define arrays for sending.
328     std::vector<double> sx;
329     std::vector<double> sy;
```

```
330     std::vector<double> sz;
331
332     // Fill arrays with the spins in each direction
333     for (uint64_t i=0; i<local_sys.spin.size(); i++){
334         sx.push_back(local_sys.spin[i][0]);
335         sy.push_back(local_sys.spin[i][1]);
336         sz.push_back(local_sys.spin[i][2]);
337     }
338     // Send ghostcells of spin in each direction
339     if(verbose) std::cout << "Rank-" << mpi_rank << "-Starting-
            exchange-Size-of-spins-is-" << sx.size() << std::endl;
340     MPI_Neighbor_alltoallw (sx.data(), counts,  &sdispls, &sendtypes,
341                             sx.data(), counts,  &rdispls, &recvtypes,
                                cart_comm);
342     MPI_Neighbor_alltoallw (sy.data(), counts,  &sdispls, &sendtypes,
343                             sy.data(), counts,  &rdispls, &recvtypes,
                                cart_comm);
344     MPI_Neighbor_alltoallw (sz.data(), counts,  &sdispls, &sendtypes,
345                             sz.data(), counts,  &rdispls, &recvtypes,
                                cart_comm);
346     // Put the spin back in the system
347     for(uint64_t i=0; i<local_sys.spin.size();i++){
348         local_sys.spin[i] = {sx[i],sy[i],sz[i]};
349     }
350     if(verbose) std::cout << "Rank-" << mpi_rank << "-Exchanged-Ghost-
            Cells" << "-Size-of-temp-is" << sx.size() <<  std::endl;
351
352
353 };
354
355 // This function checks if the index is on the edge of the block, in 3
        dimensions.
356 void check_if_we_are_on_edge(local_spins &local_sys, int
        &check_index,std::vector<int> &edges){
357     int x,y,z;
358     local_sys.padded_index_to_padded_coordinates(check_index, x, y, z);
359     if(x==1) edges[0]=1;
360     if(x==local_sys.zlen) edges[1]=1;
361     if(y==1) edges[2]=1;
362     if(y==local_sys.ylen) edges[3]=1;
363     if(z==1) edges[4]=1;
364     if(z==local_sys.xlen) edges[5]=1;
365 };
366
367 void Simulate(spin_system& sys, local_spins& localsys,MPI_Aint
        &sdispls, MPI_Aint &rdispls,
```

```
368                         MPI_Datatype &sendtypes, MPI_Datatype
                                &recvtypes,
369                         MPI_Comm cart_comm,
370                         int neighbors[6],spin_system& global_sys){
371
372     double old_energy, new_energy, spin_azimuthal, spin_polar,
            probability_of_change;
373     std::vector<double> old_state(3);
374     int not_flipped = 0;
375     int flipped = 0;
376     int local_iterations = sys.flips/mpi_size;
377     int request_ghost_index;
378     exchange_ghost_cells(localsys,sdispls, rdispls,
379                         sendtypes, recvtypes,
380                         cart_comm);
381     //std::cout << "Rank " << mpi_rank << " Exchanged Ghost Cells" <<
            " Size of spins is" << localsys.spin.size() <<  std::endl;
382     //std::cout << "Rank " << mpi_rank << " Neighbors: " <<
            neighbors[0] << " " << neighbors[1] << " " << neighbors[2] << "
            " << neighbors[3] << " " <<
383         //neighbors[4] << " " << neighbors[5] << std::endl;
384
385     for (int iteration=0; iteration<local_iterations; iteration++){
386
387         if(verbose) std::cout << "Rank-" << mpi_rank << "-off-x-" <<
                localsys.offset_x << "-off-y-" <<localsys.offset_y<< "-off-
                z-" <<localsys.offset_z << std::endl;
388
389         // First we check each neighbor to see it we have received
                updates
390         for (int i=0; i<6;i++){
391             int index = -1;
392             int thisFlag = 0;
393             MPI_Status status;
394             // Iprobe checks for incoming messages
395             MPI_Iprobe(neighbors[i],MPI_ANY_TAG,cart_comm,&thisFlag,&status);
396             if(thisFlag){
397                 //if there is a message, we receive it and put it in
                        the appropriate ghost cell
398                 double received[3];
399                 int index_received;
400                 MPI_Status status2;
401                 MPI_Recv(&received, 3, MPI_DOUBLE, neighbors[i],
                        MPI_ANY_TAG, cart_comm, &status2);
402                 index_received = status2.MPI_TAG;
```

```
403                 //if(mpi_rank==0)std::cout<<"Rank " << mpi_rank << "
                        Receiving update from " << neighbors[i]<<"\n";
404
405                 localsys.global_index_to_padded_index(index_received,index,global_sys.x
                        global_sys.ylen, global_sys.zlen);
406                 localsys.spin[index][0] = received[0];
407                 localsys.spin[index][1] = received[1];
408                 localsys.spin[index][2] = received[2];
409                 //std::cout<<"Finished update on " <<index_received<<"
                        "<<index<< " "<<mpi_rank<<"\n";
410             }
411         }
412
413         bool flip = false;
414
415         // Choose a random spin site
416         srand(iteration+mpi_rank);
417         int rand_site = rand()%(localsys.n_spins);
418         rand_site = localsys.index_to_padded_index(rand_site);
419
420         // Calculate its old energy
421
422         old_energy = energy_calculation_nd(localsys, rand_site,
                cart_comm);
423
424         // Store its old state.
425         old_state[0] = localsys.spin[rand_site][0];
426         old_state[1] = localsys.spin[rand_site][1];
427         old_state[2] = localsys.spin[rand_site][2];
428
429         // Generate new state
430         spin_azimuthal = (double) rand()/RAND_MAX * M_PI;
431         srand(mpi_rank*iteration + iteration);
432         spin_polar = (double) rand()/RAND_MAX * 2. * M_PI;
433         localsys.spin[rand_site] =
                {sin(spin_azimuthal)*cos(spin_polar),
434                                 sin(spin_azimuthal)*sin(spin_polar),
435                                 cos(spin_azimuthal)};
436         flipped++;
437         flip = true;
438         // Calculate if it lowers energy
439         new_energy = energy_calculation_nd(localsys, rand_site,
                cart_comm);
440
441         if (new_energy > old_energy){
442             // If not, see if it should be randomised in direction
```

```
443             if (verbose) std::cout << "New Energy:" << new_energy <<
                    " Old energy:" << old_energy << std::endl;
444             probability_of_change =
                    exp(-(new_energy-old_energy)/localsys.Temperature); //
                    FIgure out probability of change
445             //std::cout << "Change prob: " << probability_of_change <<
                    " Exp factor :" <<
                    -(new_energy-old_energy)/(Boltzmann*sys.Temperature) <<
                    std::endl;
446             srand((mpi_rank+1)*(iteration+1)*2);
447             if (probability_of_change < (double) rand()/RAND_MAX){
448                 // If not, revert to old state
449                 localsys.spin[rand_site] = {
450                     old_state[0], old_state[1], old_state[2]
451                 };
452                 not_flipped++;
453                 flipped--;
454                 flip = false;
455                 new_energy = old_energy;
456             }
457         }
458
459
460         if(flip){
461             std::vector<int> edges = {0,0,0,0,0,0};
462             check_if_we_are_on_edge(localsys,rand_site,edges);
463             //std::cout << "Edges " << edges[0] << " " << edges[1] <<
                    " " << edges[2] << " " << edges[3] << " " << edges[4]
                    << " " << edges[5] << " " <<std::endl;
464             for(int i=0;i<6;i++){
465                 if(edges[i]==1){
466                     int send_index;
467                     MPI_Request request;
468                     double send_spin[3] =
                            {localsys.spin[rand_site][0],localsys.spin[rand_site][1],localsy
469
470                     localsys.padded_index_to_global_index(rand_site,
                            send_index, global_sys.xlen, global_sys.ylen,
                            global_sys.zlen);
471                     MPI_Isend(&send_spin,3,MPI_DOUBLE,neighbors[i],send_index,cart_comm
472                 }
473             }
474         }
475
476         MPI_Barrier(MPI_COMM_WORLD);
477         //exchange_ghost_cells(localsys,sdispls, rdispls,
```

```
478         //                  sendtypes, recvtypes,
479         //                  cart_comm);
480     }
481     if(verbose) std::cout <<"Finished my jobs /"<<mpi_rank<<"\n";
482     MPI_Barrier(cart_comm);
483     if(verbose){
484         std::cout << "Not flipped no. is " << not_flipped << std::endl;
485         std::cout << "Flipped no. is " << flipped << std::endl;
486         std::cout << "Total energy: " << localsys.H << std::endl;
487     }
488     Calculate_h(localsys, cart_comm);
489 }
490 //=============================================================================
491 //============================   MAIN FUNCTION   ==============================
492 //=============================================================================
493 int main(int argc, char* argv[]){
494     if (verbose) std::cout << "Hello Heisenberg!" << std::endl;
495 
496     //MPI
497     MPI_Init(&argc, &argv);
498     MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);
499     MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);
500     if (verbose) {
501     // Get the name of the processor
502     char processor_name[MPI_MAX_PROCESSOR_NAME];
503     int name_len;
504     MPI_Get_processor_name(processor_name, &name_len);
505 
506     // Print off a hello world message
507     if(verbose) std::cout << "Heisenberg running on " << processor_name
508               << ", rank " << mpi_rank << " out of " << mpi_size <<
509                   std::endl;
510 
511     }
512 
513     //Initialise and load config
514     spin_system global_sys({argv, argv+argc});
515 
516     //Setup MPI
517     int dims[3] = {nproc_z, nproc_y, nproc_x};
518     int periods[3] = {1,1,1};
519     int coords[3];
520     MPI_Dims_create(mpi_size, 3, dims);
521     MPI_Comm cart_comm;
    MPI_Cart_create(MPI_COMM_WORLD, 3, dims, periods,
```

```
522             0, &cart_comm);
523     MPI_Cart_coords(cart_comm, mpi_rank, 3, coords);
524
525     int nleft, nright, nbottom, ntop, nfront, nback;
526     MPI_Cart_shift(cart_comm, 2,1,&nleft,&nright);
527     MPI_Cart_shift(cart_comm, 1,1,&nbottom,&ntop);
528     MPI_Cart_shift(cart_comm, 0,1,&nfront,&nback);
529     int neigbors[6] = {nleft, nright, nbottom, ntop, nfront, nback};
530
531     const long int offset_x = global_sys.xlen * coords[2] / nproc_x -1;
532     const long int offset_y = global_sys.ylen * coords[1] / nproc_y -1;
533     const long int offset_z = global_sys.zlen * coords[0] / nproc_z -1;
534
535     const long int end_x = global_sys.xlen * (coords[2]+1) / nproc_x
            +1;
536     const long int end_y = global_sys.ylen * (coords[1]+1) / nproc_y
            +1;
537     const long int end_z = global_sys.zlen * (coords[0]+1) / nproc_z
            +1;
538
539     if(verbose) std::cout << mpi_rank << "-" << end_x << "-" <<
            offset_x << "-"<< end_y << "-" << offset_y <<"-"<< end_z<<"-"
            << offset_z <<std::endl;
540     local_spins local_sys(global_sys,
541             end_x-offset_x-2, end_y-offset_y-2, end_z-offset_z-2,
542             offset_x, offset_y, offset_z);
543     //=============================================================================
544     //========================= START OF GHOST CELL COMMUNICATION
            SETUP ========================
545     //=============================================================================
546
547     // Define subarray types for ghost cell exchanges
548     /* The following send blocks are defined as follows:
549      * x_type sends to the nearest MPI block in the x direction
550      * y_type sends to the nearest MPI block in the y direction
551      * z_type sends to the nearest MPI block in the z direction
552      *
553      * The blocks define the parts of data that will be sent in
554      * MPI_Neigbor_alltoallw.
555      *
556      * HEAVILY INSPIRED BY
              https://github.com/essentialsofparallelcomputing/Chapter8/blob/master/GhostExch
557      * LINE 110 AND FORWARD.
558     */
559
560     // send subarrays
```

```
561     /*int subarray_sizes_x[] = { 1,local_sys.ylen,local_sys.zlen};
562     int subarray_x_start[] = {0,1,1};
563     MPI_Datatype x_type;
564     MPI_Type_create_subarray (3, array_sizes, subarray_sizes_x,
            subarray_x_start,
565                               MPI_ORDER_C, MPI_DOUBLE, &x_type);
566     MPI_Type_commit(&x_type);
567
568     int subarray_sizes_y[] = {local_sys.xlen, 1, local_sys.zlen};
569     int subarray_y_start[] = {1,0,1};
570     MPI_Datatype y_type;
571     MPI_Type_create_subarray (3, array_sizes, subarray_sizes_y,
            subarray_y_start,
572                               MPI_ORDER_C, MPI_DOUBLE, &y_type);
573     MPI_Type_commit(&y_type);
574
575     int subarray_sizes_z[] = { local_sys.xlen, local_sys.ylen,1};
576     int subarray_z_start[] = {1,1,0};
577     MPI_Datatype z_type;
578     MPI_Type_create_subarray (3, array_sizes, subarray_sizes_z,
            subarray_z_start,
579                               MPI_ORDER_C, MPI_DOUBLE, &z_type);
580     MPI_Type_commit(&z_type);
581     int xyplane_mult = local_sys.pad_ylen*local_sys.pad_xlen*8; //8
            because datatype is 3 doubles,
582     int xstride_mult = local_sys.pad_xlen*8;
583     // Define displacements of send and receive in bottom top left
            right.
584     MPI_Aint sdispls[6] = { 8,
585                             local_sys.xlen*8 ,
586                             xstride_mult,
587                             local_sys.ylen*xstride_mult,
588                             xyplane_mult,
589                             local_sys.zlen*xyplane_mult
590                             };
591     MPI_Aint rdispls[6] = {0,
592                             (local_sys.xlen+1)*8,
593                             0,
594                             (local_sys.ylen+1)*xstride_mult,
595                             0,
596                             (local_sys.zlen+1)*xyplane_mult,
597                             };
598
599     MPI_Datatype sendtypes[6] = { x_type, x_type, y_type, y_type,
            z_type, z_type};
```

```
600     MPI_Datatype recvtypes[6] = { x_type, x_type, y_type, y_type,
            z_type, z_type};
601     */
602     MPI_Datatype Vector_type;
603 //    MPI_Type_vector(1,3,0,MPI_DOUBLE,&Vector_type);
604     MPI_Type_contiguous(3,MPI_DOUBLE,&Vector_type);
605     MPI_Type_commit(&Vector_type);
606
607
608     const int esize = 1;
609     const int fsize = 1;
610     const int array_sizes[] =
            {local_sys.pad_xlen*esize,local_sys.pad_ylen,
            local_sys.pad_zlen*fsize};
611     int subarray_sizes_h[] = {
            local_sys.zlen*esize,local_sys.ylen,1*fsize};
612     int subarray_h_start[] = {1*esize,1,0};
613     MPI_Datatype h_type;
614     MPI_Type_create_subarray (3, array_sizes, subarray_sizes_h,
            subarray_h_start,
615                               MPI_ORDER_C, MPI_DOUBLE, &h_type);
616     MPI_Type_commit(&h_type);
617
618     int subarray_sizes_v[] = {local_sys.zlen*esize, 1,
            local_sys.xlen*fsize};
619     int subarray_v_start[] = {1*esize,0,1*fsize};
620     MPI_Datatype v_type;
621     MPI_Type_create_subarray (3, array_sizes, subarray_sizes_v,
            subarray_v_start,
622                               MPI_ORDER_C, MPI_DOUBLE, &v_type);
623     MPI_Type_commit(&v_type);
624
625     int subarray_sizes_d[] = { 1*esize,
            local_sys.zlen,local_sys.xlen*fsize};
626     int subarray_d_start[] = {0,1,1*fsize};
627     MPI_Datatype d_type;
628     MPI_Type_create_subarray (3, array_sizes, subarray_sizes_d,
            subarray_d_start,
629                               MPI_ORDER_C, MPI_DOUBLE, &d_type);
630     MPI_Type_commit(&d_type);
631
632
633     int element_size = sizeof(double);
634     //std::cout << element_size << std::endl;
635     int nhalo = 1;
```

```
636     int xyplane_mult =
            local_sys.pad_ylen*local_sys.pad_xlen*element_size; //8 because
            datatype is 3 doubles,
637     int xstride_mult = local_sys.pad_xlen*element_size;
638     // Define displacements of send and receive in bottom top left
            right.
639     MPI_Aint sdispls[6] = { nhalo           * xyplane_mult ,
640                             local_sys.zlen * xyplane_mult ,
641                             nhalo           * xstride_mult,
642                             local_sys.ylen * xstride_mult,
643                             nhalo           * element_size,
644                             local_sys.xlen * element_size
645                             };
646     MPI_Aint rdispls[6] = {0,
647                            (local_sys.zlen+1) * xyplane_mult ,
648                            0,
649                            (local_sys.ylen+1) * xstride_mult,
650                            0,
651                            (local_sys.xlen+1) * element_size,
652                            };
653     for(int i=0;i<6;i++){
654     //    std::cout << "Rank " << mpi_rank << " Send " << i << " is
           "<< sdispls[i] << " Recv is " << rdispls[i] << std::endl;
655     }
656     MPI_Datatype sendtypes[6] = { d_type, d_type, v_type, v_type,
            h_type, h_type};
657     MPI_Datatype recvtypes[6] = { d_type, d_type, v_type, v_type,
            h_type, h_type};
658
659     //=============================================================================
660     //============================ END OF GHOST CELL COMMUNICATION SETUP
            ============================
661     //=============================================================================
662     //Generate system TODO: Done in parallel
663     generate_positions_box(local_sys);
664     generate_spin_directions(local_sys);
665     generate_neighbours(local_sys);
666     //Magic TODO h as reduction
667     Calculate_h(local_sys, cart_comm);
668
669     auto begin = std::chrono::steady_clock::now();
670     Simulate(global_sys,local_sys,*sdispls, *rdispls,
671                          *sendtypes, *recvtypes,
672                          cart_comm,
673                          neigbors, global_sys);
674     auto end = std::chrono::steady_clock::now();
```

```
675
676
677     MPI_Barrier(cart_comm);
678     MPI_Reduce(&local_sys.H, &global_sys.H, 1, MPI_DOUBLE, MPI_SUM, 0,
            cart_comm);
679     if (mpi_rank == 0){ std::cout << "Final_energy:-" <<
            "Elapsed_time" << "Temperature-" << "B_field-" << "System_size-
            " << "No_of_ranks-" << "No_of_flips-" << "Version-" <<std::endl;
680                         std::cout << global_sys.H << "-" <<
                                (end-begin).count() / 1000000000.0 << "-"
                                << global_sys.Temperature << "-" <<
                                global_sys.B <<
681                                    "-" << global_sys.n_spins << "-" <<
                                        mpi_size << "-" << global_sys.flips
                                        << "-" << 2 << std::endl;
682                             }
683     if(global_sys.write==1){
684     std::vector<double> px, py, pz;
685     std::vector<double> sx, sy, sz;
686     std::vector<double> energy;
687     std::vector<double> globpx, globpy, globpz;
688     std::vector<double> globsx, globsy, globsz;
689     std::vector<double> globenergy;
690     if (mpi_rank ==0) {
691         globpx.reserve(global_sys.n_spins);
692         globpy.reserve(global_sys.n_spins);
693         globpz.reserve(global_sys.n_spins);
694         globsx.reserve(global_sys.n_spins);
695         globsy.reserve(global_sys.n_spins);
696         globsz.reserve(global_sys.n_spins);
697         globenergy.reserve(global_sys.n_spins);
698
699     }
700
701     int temp;
702     for (int i = 0; i<local_sys.n_spins; i++){
703                 temp = local_sys.index_to_padded_index(i);
704                 px.push_back(local_sys.position[temp][0]+local_sys.offset_x);
705                 py.push_back(local_sys.position[temp][1]+local_sys.offset_y);
706                 pz.push_back(local_sys.position[temp][2]+local_sys.offset_z);
707                 sx.push_back(local_sys.spin[temp][0]);
708                 sy.push_back(local_sys.spin[temp][1]);
709                 sz.push_back(local_sys.spin[temp][2]);
710                 energy.push_back(energy_calculation_nd(local_sys,temp,cart_comm));
711         }
712     MPI_Barrier(cart_comm);
```

```
713
714     MPI_Gather(px.data(), px.size(), MPI_DOUBLE,
715                globpx.data(), local_sys.n_spins, MPI_DOUBLE,
716                0, cart_comm);
717
718     MPI_Gather(py.data(), py.size(), MPI_DOUBLE,
719                globpy.data(), local_sys.n_spins, MPI_DOUBLE,
720                0, cart_comm);
721     MPI_Gather(pz.data(), pz.size(), MPI_DOUBLE,
722                globpz.data(), local_sys.n_spins, MPI_DOUBLE,
723                0, cart_comm);
724     MPI_Gather(sx.data(), sx.size(), MPI_DOUBLE,
725                globsx.data(), local_sys.n_spins, MPI_DOUBLE,
726                0, cart_comm);
727     MPI_Gather(sy.data(), sy.size(), MPI_DOUBLE,
728                globsy.data(), local_sys.n_spins, MPI_DOUBLE,
729                0, cart_comm);
730     MPI_Gather(sz.data(), sz.size(), MPI_DOUBLE,
731                globsz.data(), local_sys.n_spins, MPI_DOUBLE,
732                0, cart_comm);
733     MPI_Gather(energy.data(), energy.size(), MPI_DOUBLE,
734                globenergy.data(), local_sys.n_spins, MPI_DOUBLE,
735                0, cart_comm);
736
737     if (mpi_rank == 0){
738         for (int i=0; i<global_sys.n_spins; i++){
739             global_sys.spin.push_back({globsx[i], globsy[i],
                    globsz[i]});
740             global_sys.position.push_back({globpx[i], globpy[i],
                    globpz[i]});
741             global_sys.energy.push_back(globenergy[i]);
742         }
743         std::ofstream file(global_sys.filename); // open file
744         Writeoutput(global_sys, file, cart_comm);
745     }
746     }
747     MPI_Type_free(&h_type);
748     MPI_Type_free(&v_type);
749     MPI_Type_free(&d_type);
750     MPI_Type_free(&Vector_type);
751     MPI_Barrier(cart_comm);
752
753     MPI_Finalize();
754     return 0;
755 }
```

Section B Source Code parallel Send and receive as necessary

```
1  #include <vector>
2
3  #include <iostream>
4  #include <iomanip>
5
6  #include <fstream>
7  #include <chrono>
8  #include <cmath>
9  #include <fstream>
10 #include <cstdlib>
11 #include <mpi.h>
12
13 int mpi_size;
14 int mpi_rank;
15 int nproc_x = 2, nproc_y = 2,nproc_z=2;
16 enum {ghost_cell_request, ghost_cell_answer};
17
18 bool verbose = false;
19 class spin_system {
20     public:
21     int flips = 100; // Number of flips the system will simulate.
22     int n_spins = 64; // Number of spins in The system.
23     int n_dims = 3; // Number of dimensions the spins are placed in.
24     float N_spins_row; // Number of rows in the square/cube
25
26     int x_offsets[6] = {-1,0,0,1,0,0}; // For calculating neighbor
           indices in energy calculation
27     int y_offsets[6] = {0,-1,1,0,0,0};
28     int z_offsets[6] = {0,0,0,0,-1,1};
29
30     int x_dist = 1;
31     int y_dist = 1;
32     int z_dist = 1;
33
34     int nearest_neighbours = 1; // Number of nearest neighbour
           interactions to be calculated
35     int xlen, ylen, zlen;
36     double J = 1; // Magnetization parameter, used in calculating
           energy.
37     double H; // Total energy of the system;
38     double B = 0; // Magnetic field in z direction
39     double Temperature = 1; // Temperature of the system.
```

```
40      std::string filename = "parallel_out.txt"; // Output file.
41      int write = 1;
42      std::vector<std::vector<double>> position; // Three by n_spins
            matrix, defining the spin's 3d position.
43      std::vector<std::vector<double>> spin; // Three by n_spins matrix,
            defining the spin vector for each spin.
44      std::vector<std::vector<int>>    neighbours; // 2*n_dims by
            n_spins matrix, defining the neighbour indices of each cell, so
            they need only be calculated once.
45      std::vector<double>              energy; // Energy in each cell,
            derivaed at the end.
46      spin_system(std::vector <std::string> argument){
47      for (long unsigned int i = 1; i<argument.size() ; i += 2){
48              std::string arg = argument.at(i);
49              if(arg=="-h"){ // Write help
50                  std::cout << "Heisenberg_simulation\n--flips <number
                        of flips performed>\n--nspins <number of spins
                        simulated>\n--ndims <number of dimensions to
                        simulate> \n"
51                            << "--ofile <filename>\n--magnet <strength
                                of external magnetic field in z
                                direction>\n"
52                            << "--temp <temperature> \n" << "
                                --writeout <write to data file (1 for
                                true, 0 for false)>\n";
53
54                  exit(0);
55                  break;
56              } else if(arg=="--flips"){
57                  flips = std::stoi(argument[i+1]);
58              } else if(arg=="--nspins"){
59                  n_spins = std::stoi(argument[i+1]);
60              } else if(arg=="--ndims"){
61                  n_dims = std::stoi(argument[i+1]);
62              } else if(arg=="--ofile"){
63                  filename = argument[i+1];
64              } else if(arg=="--magnet"){
65                  B = std::stoi(argument[i+1]);
66              } else if(arg=="--temp"){
67                  Temperature = std::stod(argument[i+1]);
68              } else if(arg=="--writeout"){
69                  write = std::stoi(argument[i+1]);
70              } else{
71                  std::cout << "--> error: the argument type is not
                        recognized \n";
72              }
```

```
73          }
74      N_spins_row = cbrt(double(n_spins)); //Equal size in all dimensions
75      int n_spins_row = round(N_spins_row);
76      xlen = n_spins_row;
77      ylen = n_spins_row;
78      zlen = n_spins_row;
79      if(verbose) std::cout << "Nspins row " << n_spins_row << std::endl;
80      }
81  };
82
83  class local_spins{
84      public:
85      int x_offsets[6] = {-1,0,0,1,0,0}; // For calculating neighbor
            indices in energy calculation
86      int y_offsets[6] = {0,-1,1,0,0,0};
87      int z_offsets[6] = {0,0,0,0,-1,1};
88
89      int x_dist,y_dist,z_dist;
90
91      int nearest_neighbours; // Number of nearest neighbour
            interactions to be calculated
92      int xlen, ylen, zlen;
93      int pad_xlen, pad_ylen, pad_zlen;
94      int offset_x, offset_y, offset_z;
95      int n_spins;
96      double J; // Magnetization parameter, used in calculating energy.
97      double H; // Total energy of the system;
98      double B; // Magnetic field in z direction
99      double Temperature; // Temperature of the system.
100     std::string filename; // Output file.
101
102     int no_in_padded_layer, no_in_layer;
103
104     local_spins(spin_system &sys,
105             int local_xlen, int local_ylen, int local_zlen,
106             int offx, int offy, int offz){
107         x_dist = sys.x_dist;
108         y_dist = sys.y_dist;
109         z_dist = sys.z_dist;
110
111         nearest_neighbours = sys.nearest_neighbours;
112         xlen = local_xlen;
113         ylen = local_ylen;
114         zlen = local_zlen;
115
116         pad_xlen = xlen+2;
```

```
117         pad_ylen = ylen+2;
118         pad_zlen = zlen+2;
119 
120         n_spins = xlen*ylen*zlen;
121         no_in_layer = xlen*ylen;
122         no_in_padded_layer = (xlen+2)*(ylen+2);
123 
124         offset_x = offx;
125         offset_y = offy;
126         offset_z = offz;
127         J = sys.J;
128         H = sys.H;
129         B = sys.B;
130         Temperature = sys.Temperature;
131         filename = sys.filename;
132     };
133     std::vector<std::vector<double>> position; // Three by n_spins
        matrix, defining the spin's 3d position.
134     std::vector<std::vector<double>> spin; // Three by n_spins matrix,
        defining the spin vector for each spin.
135     std::vector<std::vector<int>>    neighbours; // 2*n_dims by
        n_spins matrix, defining the neighbour indices of each cell, so
        they need only be calculated once.
136 
137 
138     int index_to_padded_index(int index){
139 
140         int x = index%(no_in_layer)%xlen + 1;
141         int y = (index%(no_in_layer))/xlen + 1;
142         int z = index/(no_in_layer) + 1;
143 
144         return z*no_in_padded_layer + y*pad_xlen + x;
145     }
146 
147     int padded_index_to_index(int index){
148         int x,y,z;
149         padded_index_to_padded_coordinates(index,x,y,z);
150         x -= 1;
151         y -= 1;
152         z -= 1;
153         return z * xlen * ylen + y * xlen + x;
154     }
155     void padded_index_to_padded_coordinates(int index, int& x, int& y,
        int& z){
156         x = index % pad_xlen; // Which row the spin is in
157         y = (index/pad_ylen)%pad_ylen; // Which column the spin is in
```

```
158         z = index / no_in_padded_layer;
159     }
160
161     void index_to_coordinates(int index, int& x, int& y, int& z){
162         x = index % xlen;
163         y = (index/ylen)%ylen;
164         z = index / (ylen * xlen);
165     }
166     void padded_coordinates_to_padded_index(int &index,int x, int y,
            int z){
167         index = x%pad_xlen + (y%pad_ylen) * pad_xlen + (z%pad_zlen) *
                no_in_padded_layer;
168     }
169
170     void padded_index_to_global_index(int p_index, int &g_index,int
            global_x, int global_y, int global_z){
171         int local_index = padded_index_to_index(p_index);
172         int x,y,z;
173         index_to_coordinates(local_index,x,y,z);
174         x += offset_x+1;
175         y += offset_y+1;
176         z += offset_z+1;
177         g_index = x%global_x + (y%global_y) * global_x + (z%global_z)
                * global_x * global_y;
178     }
179
180     void global_index_to_padded_index(int g_index, int &p_index, int
            global_x, int global_y, int global_z){
181         // Find global coordinaetes
182         int g_x = g_index % global_x;
183         int g_y = (g_index/global_y)%global_y;
184         int g_z = g_index / (global_y * global_x);
185
186         // Define padded coords
187         int x,y,z;
188
189         if(g_x==0&&offset_x==global_x-xlen-1){x=pad_xlen-1;} //If on
                the left / bottom / back edge, of global, and local is on
                the right / top / front edge, set to be on the right / top
                / front edge
190         else{
191         x = (g_x -(offset_x)); // Convert from global to padded index
192         if(x==global_x) x=0; // if on the right / top / front edge,
                set to be on the left / bottom / back edge.
193         x = (x + pad_xlen)%pad_xlen; // Makes sure is non-negative
194         }
```

```
195         if(g_y==0&&offset_y==global_y-ylen-1){y=pad_ylen-1;}
196         else{
197         y = (g_y -(offset_y));
198         if(y==global_y) y=0;
199         y = (y + pad_ylen)%pad_ylen;
200         }
201         if(g_z==0&&offset_z==global_z-zlen-1){z=pad_zlen-1;}
202         else{
203         z = (g_z -(offset_z));
204         if(z==global_z) z=0;
205         z = (z + pad_zlen)%pad_zlen;
206         }
207         padded_coordinates_to_padded_index(p_index,x,y,z);
208     }
209 };
210
211
212 // Function that generates rectangular positions for alle the spins in
        the system,
213 void generate_positions_box(local_spins &sys){
214         for (double k=0; k<sys.pad_zlen; k++)
215         for (double j=0; j<sys.pad_ylen; j++)
216         for (double i=0; i<sys.pad_xlen; i++)
217             sys.position.push_back({double(i*sys.x_dist),
                    double(j*sys.y_dist), double(k*sys.z_dist)});
218 };
219
220 // Function that generates random directions for all the spins in the
        system
221 void generate_spin_directions(local_spins &sys){
222
223     for (int i = 0; i<sys.pad_zlen*sys.no_in_padded_layer; i++){
224         srand(i); // Seed is here to make it perform nicely when
                comparing to parallel
225         double spin_azimuthal = (double) rand()/RAND_MAX * M_PI;
226         srand(i*rand()); // Seed is here to make it perform nicely
                when comparing to parallel
227         double spin_polar = (double) rand()/RAND_MAX * 2. * M_PI;
228
229         sys.spin.push_back({sin(spin_azimuthal)*cos(spin_polar),
230                             sin(spin_azimuthal)*sin(spin_polar),
231                             cos(spin_azimuthal)});
232     }
233 };
234
235 void generate_neighbours(local_spins &sys){
```

```
236
237     for (int spin = 0; spin< sys.pad_zlen*sys.no_in_padded_layer;
            spin++){
238         // Find position in square / cube
239         int spin_x, spin_y, spin_z;
240         sys.padded_index_to_padded_coordinates(spin, spin_x, spin_y,
                spin_z);
241
242         // Find indices of neighbours
243         std::vector<int> spin_interactions;
244         for(int i = 0; i < 6; i++){
245             spin_interactions.push_back((spin_x +
                    sys.x_offsets[i])%sys.pad_xlen +
246                                         (spin_y +
                                            sys.y_offsets[i])%sys.pad_ylen
                                            * sys.pad_xlen +
247                                         (spin_z +
                                            sys.z_offsets[i])%sys.pad_zlen
                                            * sys.no_in_padded_layer);
248         }
249         sys.neighbours.push_back(spin_interactions);
250     }
251 }
252 // Function that calculates the energy of a single spin in 2d
253 double energy_calculation_nd(local_spins &sys, int spin, MPI_Comm&
        cart_comm){
254     double energy = 0;
255     double dot_product;
256
257     for (int i=0; i<6; i++){
258         // Calculate the energy with the nearest neighbour with no
                corners
259         dot_product =
                sys.spin[spin][0]*sys.spin[sys.neighbours[spin][i]][0]
260                     + sys.spin[spin][1]*
                            sys.spin[sys.neighbours[spin][i]][1]
261                     + sys.spin[spin][2]*
                            sys.spin[sys.neighbours[spin][i]][2];
262         energy -= sys.J/2*dot_product;
263     }
264     energy += sys.B*sys.spin[spin][2];
265     return energy;
266 };
267
268 // Calculate the total energy of the system
269 void Calculate_h(local_spins& sys, MPI_Comm cart_comm){
```

```
270     sys.H = 0; // Set H to zero before recalculating it
271     double mag_energy = 0;
272     for (int i=0; i<sys.n_spins; i++){
273         int pad_i = sys.index_to_padded_index(i);
274         sys.H += energy_calculation_nd(sys, pad_i, cart_comm)*0.5; //
              Half the energy, because we calculate on all the spins
275         mag_energy += sys.spin[pad_i][2];
276     }
277     sys.H += sys.B*mag_energy * 0.5; // Half of the magnetization
          energy is removed above
278 };
279
280 // Write the spin configurations in the output file.
281 void Writeoutput(spin_system& sys, std::ofstream& file, MPI_Comm
        cart_comm){
282     // Loop over all spins, and write out position and spin direction<
283     file << "Position_x" << "Position_y" << "Position_z" << "Spin_x
          " << "Spin_y" << "Spin_z" << "Spin_energy" <<
          "Temperature" << "n_spins" << std::endl;
284     for (int i = 0; i<sys.n_spins; i++){
285         if (i == 0) {
286             file << sys.position[i][0] << "," << sys.position[i][1] <<
                  "," << sys.position[i][2] << ","
287             << sys.spin[i][0] << "," << sys.spin[i][1] << "," <<
                  sys.spin[i][2] << "," << sys.energy[i]
288             << "," << sys.Temperature << "," << sys.n_spins
289             << std::endl;
290         } else {
291             file << sys.position[i][0] << "," << sys.position[i][1] <<
                  "," << sys.position[i][2] << ","
292             << sys.spin[i][0] << "," << sys.spin[i][1] << "," <<
                  sys.spin[i][2] << "," << sys.energy[i]
293             << std::endl;
294         }
295     }
296 };
297
298
299 void exchange_ghost_cells(local_spins &local_sys,
300                           MPI_Aint &sdispls, MPI_Aint &rdispls,
301                           MPI_Datatype &sendtypes, MPI_Datatype
                              &recvtypes,
302                           MPI_Comm cart_comm){
303     int counts[6] = {1,1,1,1,1,1};
304
305     // Define arrays for sending.
```

```
306       std::vector<double> sx;
307       std::vector<double> sy;
308       std::vector<double> sz;
309
310       // Fill arrays with the spins in each direction
311       for (uint64_t i=0; i<local_sys.spin.size(); i++){
312           sx.push_back(local_sys.spin[i][0]);
313           sy.push_back(local_sys.spin[i][1]);
314           sz.push_back(local_sys.spin[i][2]);
315       }
316       // Send ghostcells of spin in each direction
317       if(verbose) std::cout << "Rank-" << mpi_rank << "-Starting-
              exchange-Size-of-spins-is-" << sx.size() << std::endl;
318       MPI_Neighbor_alltoallw (sx.data(), counts,  &sdispls, &sendtypes,
319                               sx.data(), counts,  &rdispls, &recvtypes,
                                    cart_comm);
320       MPI_Neighbor_alltoallw (sy.data(), counts,  &sdispls, &sendtypes,
321                               sy.data(), counts,  &rdispls, &recvtypes,
                                    cart_comm);
322       MPI_Neighbor_alltoallw (sz.data(), counts,  &sdispls, &sendtypes,
323                               sz.data(), counts,  &rdispls, &recvtypes,
                                    cart_comm);
324       // Put the spin back in the system
325       for(uint64_t i=0; i<local_sys.spin.size();i++){
326           local_sys.spin[i] = {sx[i],sy[i],sz[i]};
327       }
328       if(verbose) std::cout << "Rank-" << mpi_rank << "-Exchanged-Ghost-
              Cells" << "-Size-of-temp-is" << sx.size() <<  std::endl;
329
330
331   };
332
333   // This function checks if the index is on the edge of the block, in 3
          dimensions.
334   void check_if_we_are_on_edge(local_spins &local_sys, int
          &check_index,std::vector<int> &edges){
335       int x,y,z;
336       local_sys.padded_index_to_padded_coordinates(check_index, x, y, z);
337       if(x==1) edges[0]=1;
338       if(x==local_sys.zlen) edges[1]=1;
339       if(y==1) edges[2]=1;
340       if(y==local_sys.ylen) edges[3]=1;
341       if(z==1) edges[4]=1;
342       if(z==local_sys.xlen) edges[5]=1;
343   };
344
```

```
345 void Simulate(spin_system& sys, local_spins& localsys,MPI_Aint
      &sdispls, MPI_Aint &rdispls,
346                         MPI_Datatype &sendtypes, MPI_Datatype
                               &recvtypes,
347                         MPI_Comm cart_comm,
348                         int neighbors[6],spin_system& global_sys){
349
350     double old_energy, new_energy, spin_azimuthal, spin_polar,
           probability_of_change;
351     std::vector<double> old_state(3);
352     int not_flipped = 0;
353     int flipped = 0;
354     int local_iterations = sys.flips/mpi_size;
355     exchange_ghost_cells(localsys,sdispls, rdispls,
356                         sendtypes, recvtypes,
357                         cart_comm);
358
359     for (int iteration=0; iteration<local_iterations; iteration++){
360
361         if(verbose) std::cout << "Rank-" << mpi_rank << "-off-x-" <<
               localsys.offset_x << "-off-y-" <<localsys.offset_y<< "-off-
               z-" <<localsys.offset_z << std::endl;
362
363         // First we check each neighbor to see it we have received
               updates
364         for (int i=0; i<6;i++){
365             int index = -1;
366             int thisFlag = 0;
367             MPI_Status status;
368             // Iprobe checks for incoming messages
369             MPI_Iprobe(neighbors[i],MPI_ANY_TAG,cart_comm,&thisFlag,&status);
370             if(thisFlag){
371                 //if there is a message, we receive it and put it in
                       the appropriate ghost cell
372                 double received[3];
373                 int index_received;
374                 MPI_Status status2;
375                 MPI_Recv(&received, 3, MPI_DOUBLE, neighbors[i],
                       MPI_ANY_TAG, cart_comm, &status2);
376                 index_received = status2.MPI_TAG;
377
378                 localsys.global_index_to_padded_index(index_received,index,global_sys.x
                       global_sys.ylen, global_sys.zlen);
379                 localsys.spin[index][0] = received[0];
380                 localsys.spin[index][1] = received[1];
381                 localsys.spin[index][2] = received[2];
```

```
382             }
383         }
384 
385         bool flip = false;
386 
387         // Choose a random spin site
388         srand(iteration+mpi_rank);
389         int rand_site = rand()%(localsys.n_spins);
390         rand_site = localsys.index_to_padded_index(rand_site);
391 
392         // Calculate its old energy
393 
394         old_energy = energy_calculation_nd(localsys, rand_site,
                cart_comm);
395 
396         // Store its old state.
397         old_state[0] = localsys.spin[rand_site][0];
398         old_state[1] = localsys.spin[rand_site][1];
399         old_state[2] = localsys.spin[rand_site][2];
400 
401         // Generate new state
402         spin_azimuthal = (double) rand()/RAND_MAX * M_PI;
403         srand(mpi_rank*iteration + iteration);
404         spin_polar = (double) rand()/RAND_MAX * 2. * M_PI;
405         localsys.spin[rand_site] =
                {sin(spin_azimuthal)*cos(spin_polar),
406                                 sin(spin_azimuthal)*sin(spin_polar),
407                                 cos(spin_azimuthal)};
408         flipped++;
409         flip = true;
410         // Calculate if it lowers energy
411         new_energy = energy_calculation_nd(localsys, rand_site,
                cart_comm);
412 
413         if (new_energy > old_energy){
414             // If not, see if it should be randomised in direction
415             if (verbose) std::cout << "New Energy:" << new_energy <<
                    "Old energy:" << old_energy << std::endl;
416             probability_of_change =
                    exp(-(new_energy-old_energy)/localsys.Temperature); //
                    FIgure out probability of change
417 
418                     srand((mpi_rank+1)*(iteration+1)*2);
419             if (probability_of_change < (double) rand()/RAND_MAX){
420                 // If not, revert to old state
421                 localsys.spin[rand_site] = {
```

```
422                     old_state[0], old_state[1], old_state[2]
423                 };
424                 not_flipped++;
425                 flipped--;
426                 flip = false;
427                 new_energy = old_energy;
428             }
429         }
430 
431 
432         if(flip){
433             std::vector<int> edges = {0,0,0,0,0,0};
434             check_if_we_are_on_edge(localsys,rand_site,edges);
435             for(int i=0;i<6;i++){
436                 if(edges[i]==1){
437                     int send_index;
438                     MPI_Request request;
439                     double send_spin[3] =
                            {localsys.spin[rand_site][0],localsys.spin[rand_site][1],localsy
440 
441                     localsys.padded_index_to_global_index(rand_site,
                            send_index, global_sys.xlen, global_sys.ylen,
                            global_sys.zlen);
442                     MPI_Isend(&send_spin,3,MPI_DOUBLE,neighbors[i],send_index,cart_comm
443                 }
444             }
445         }
446 
447         MPI_Barrier(MPI_COMM_WORLD);
448 
449     }
450     if(verbose) std::cout <<"Finished my jobs /"<<mpi_rank<<"\n";
451     MPI_Barrier(cart_comm);
452     if(verbose){
453         std::cout << "Not flipped no. is " << not_flipped << std::endl;
454         std::cout << "Flipped no. is " << flipped << std::endl;
455         std::cout << "Total energy: " << localsys.H << std::endl;
456     }
457     Calculate_h(localsys, cart_comm);
458 }
459 //=============================================================================
460 //============================    MAIN FUNCTION
      ==========================================
461 //=============================================================================
462 int main(int argc, char* argv[]){
463     if (verbose) std::cout << "Hello Heisenberg!" << std::endl;
```

```
464
465     //MPI
466     MPI_Init(&argc, &argv);
467     MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);
468     MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);
469     if (verbose) {
470     // Get the name of the processor
471     char processor_name[MPI_MAX_PROCESSOR_NAME];
472     int name_len;
473     MPI_Get_processor_name(processor_name, &name_len);
474
475     // Print off a hello world message
476     if(verbose) std::cout << "Heisenberg running on " << processor_name
477               << ", rank " << mpi_rank << " out of " << mpi_size <<
                      std::endl;
478     }
479
480     //Initialise and load config
481     spin_system global_sys({argv, argv+argc});
482
483     //Setup MPI
484     int dims[3] = {nproc_z, nproc_y, nproc_x};
485     int periods[3] = {1,1,1};
486     int coords[3];
487     MPI_Dims_create(mpi_size, 3, dims);
488     MPI_Comm cart_comm;
489     MPI_Cart_create(MPI_COMM_WORLD, 3, dims, periods,
490             0, &cart_comm);
491     MPI_Cart_coords(cart_comm, mpi_rank, 3, coords);
492
493     int nleft, nright, nbottom, ntop, nfront, nback;
494     MPI_Cart_shift(cart_comm, 2,1,&nleft,&nright);
495     MPI_Cart_shift(cart_comm, 1,1,&nbottom,&ntop);
496     MPI_Cart_shift(cart_comm, 0,1,&nfront,&nback);
497     int neigbors[6] = {nleft, nright, nbottom, ntop, nfront, nback};
498
499     const long int offset_x = global_sys.xlen * coords[2] / nproc_x -1;
500     const long int offset_y = global_sys.ylen * coords[1] / nproc_y -1;
501     const long int offset_z = global_sys.zlen * coords[0] / nproc_z -1;
502
503     const long int end_x = global_sys.xlen * (coords[2]+1) / nproc_x
          +1;
504     const long int end_y = global_sys.ylen * (coords[1]+1) / nproc_y
          +1;
505     const long int end_z = global_sys.zlen * (coords[0]+1) / nproc_z
          +1;
```

```
506
507     if(verbose) std::cout << mpi_rank << "-" << end_x << "-" <<
            offset_x << "-"<< end_y << "-" << offset_y <<"-"<< end_z<<"-"
            << offset_z <<std::endl;
508     local_spins local_sys(global_sys,
509             end_x-offset_x-2, end_y-offset_y-2, end_z-offset_z-2,
510             offset_x, offset_y, offset_z);
511     //=========================================================================
512     //========================= START OF GHOST CELL COMMUNICATION
            SETUP ========================
513     //=========================================================================
514
515     /* The following send blocks are defined as follows:
516      * h_type sends to the nearest MPI block in the horizontal
            direction
517      * v_type sends to the nearest MPI block in the vertical direction
518      * d_type sends to the nearest MPI block in the depth direction
519      *
520      * The blocks define the parts of data that will be sent in
521      * MPI_Neigbor_alltoallw.
522      *
523      * HEAVILY INSPIRED BY
            https://github.com/essentialsofparallelcomputing/Chapter8/blob/master/GhostExch
524      * LINE 110 AND FORWARD.
525      */
526
527     const int esize = 1;
528     const int fsize = 1;
529     const int array_sizes[] =
            {local_sys.pad_xlen*esize,local_sys.pad_ylen,
            local_sys.pad_zlen*fsize};
530     int subarray_sizes_h[] = {
            local_sys.zlen*esize,local_sys.ylen,1*fsize};
531     int subarray_h_start[] = {1*esize,1,0};
532     MPI_Datatype h_type;
533     MPI_Type_create_subarray (3, array_sizes, subarray_sizes_h,
            subarray_h_start,
534                               MPI_ORDER_C, MPI_DOUBLE, &h_type);
535     MPI_Type_commit(&h_type);
536
537     int subarray_sizes_v[] = {local_sys.zlen*esize, 1,
            local_sys.xlen*fsize};
538     int subarray_v_start[] = {1*esize,0,1*fsize};
539     MPI_Datatype v_type;
540     MPI_Type_create_subarray (3, array_sizes, subarray_sizes_v,
            subarray_v_start,
```

```
541                                 MPI_ORDER_C, MPI_DOUBLE, &v_type);
542     MPI_Type_commit(&v_type);
543
544     int subarray_sizes_d[] = { 1*esize,
            local_sys.zlen,local_sys.xlen*fsize};
545     int subarray_d_start[] = {0,1,1*fsize};
546     MPI_Datatype d_type;
547     MPI_Type_create_subarray (3, array_sizes, subarray_sizes_d,
            subarray_d_start,
548                                 MPI_ORDER_C, MPI_DOUBLE, &d_type);
549     MPI_Type_commit(&d_type);
550
551
552     int element_size = sizeof(double);
553     //std::cout << element_size << std::endl;
554     int nhalo = 1;
555     int xyplane_mult =
            local_sys.pad_ylen*local_sys.pad_xlen*element_size; //8 because
            datatype is 3 doubles,
556     int xstride_mult = local_sys.pad_xlen*element_size;
557     // Define displacements of send and receive in bottom top left
            right.
558     MPI_Aint sdispls[6] = { nhalo          * xyplane_mult ,
559                             local_sys.zlen * xyplane_mult ,
560                             nhalo          * xstride_mult,
561                             local_sys.ylen * xstride_mult,
562                             nhalo          * element_size,
563                             local_sys.xlen * element_size
564                             };
565     MPI_Aint rdispls[6] = {0,
566                            (local_sys.zlen+1) * xyplane_mult ,
567                            0,
568                            (local_sys.ylen+1) * xstride_mult,
569                            0,
570                            (local_sys.xlen+1) * element_size,
571                            };
572     for(int i=0;i<6;i++){
573     //    std::cout << "Rank " << mpi_rank << " Send " << i << " is
            "<< sdispls[i] << " Recv is " << rdispls[i] << std::endl;
574     }
575     MPI_Datatype sendtypes[6] = { d_type, d_type, v_type, v_type,
            h_type, h_type};
576     MPI_Datatype recvtypes[6] = { d_type, d_type, v_type, v_type,
            h_type, h_type};
577
578     //=============================================================================
```

```
579     //============================ END OF GHOST CELL COMMUNICATION SETUP
            ============================
580     //=============================================================================
581     //Generate system TODO: Done in parallel
582     generate_positions_box(local_sys);
583     generate_spin_directions(local_sys);
584     generate_neighbours(local_sys);
585     //Magic TODO h as reduction
586     Calculate_h(local_sys, cart_comm);
587
588     auto begin = std::chrono::steady_clock::now();
589     Simulate(global_sys,local_sys,*sdispls, *rdispls,
590                         *sendtypes, *recvtypes,
591                         cart_comm,
592                         neigbors, global_sys);
593     auto end = std::chrono::steady_clock::now();
594
595
596     MPI_Barrier(cart_comm);
597     MPI_Reduce(&local_sys.H, &global_sys.H, 1, MPI_DOUBLE, MPI_SUM, 0,
            cart_comm);
598     if (mpi_rank == 0){ std::cout << "Final_energy:-" <<
            "Elapsed_time" << "Temperature-" << "B_field-" << "System_size-
            " << "No_of_ranks-" << "No_of_flips-" << "Version-" <<std::endl;
599                         std::cout << global_sys.H << "-" <<
                                (end-begin).count() / 1000000000.0 << "-"
                                << global_sys.Temperature << "-" <<
                                global_sys.B <<
600                                 "-" << global_sys.n_spins << "-" <<
                                        mpi_size << "-" << global_sys.flips
                                        << "-" << 2 << std::endl;
601                             }
602     if(global_sys.write==1){
603     std::vector<double> px, py, pz;
604     std::vector<double> sx, sy, sz;
605     std::vector<double> energy;
606     std::vector<double> globpx, globpy, globpz;
607     std::vector<double> globsx, globsy, globsz;
608     std::vector<double> globenergy;
609     if (mpi_rank ==0) {
610         globpx.reserve(global_sys.n_spins);
611         globpy.reserve(global_sys.n_spins);
612         globpz.reserve(global_sys.n_spins);
613         globsx.reserve(global_sys.n_spins);
614         globsy.reserve(global_sys.n_spins);
615         globsz.reserve(global_sys.n_spins);
```

```
616         globenergy.reserve(global_sys.n_spins);
617
618     }
619
620     int temp;
621     for (int i = 0; i<local_sys.n_spins; i++){
622                 temp = local_sys.index_to_padded_index(i);
623                 px.push_back(local_sys.position[temp][0]+local_sys.offset_x);
624                 py.push_back(local_sys.position[temp][1]+local_sys.offset_y);
625                 pz.push_back(local_sys.position[temp][2]+local_sys.offset_z);
626                 sx.push_back(local_sys.spin[temp][0]);
627                 sy.push_back(local_sys.spin[temp][1]);
628                 sz.push_back(local_sys.spin[temp][2]);
629                 energy.push_back(energy_calculation_nd(local_sys,temp,cart_comm));
630         }
631     MPI_Barrier(cart_comm);
632
633     MPI_Gather(px.data(), px.size(), MPI_DOUBLE,
634                 globpx.data(), local_sys.n_spins, MPI_DOUBLE,
635                 0, cart_comm);
636
637     MPI_Gather(py.data(), py.size(), MPI_DOUBLE,
638                 globpy.data(), local_sys.n_spins, MPI_DOUBLE,
639                 0, cart_comm);
640     MPI_Gather(pz.data(), pz.size(), MPI_DOUBLE,
641                 globpz.data(), local_sys.n_spins, MPI_DOUBLE,
642                 0, cart_comm);
643     MPI_Gather(sx.data(), sx.size(), MPI_DOUBLE,
644                 globsx.data(), local_sys.n_spins, MPI_DOUBLE,
645                 0, cart_comm);
646     MPI_Gather(sy.data(), sy.size(), MPI_DOUBLE,
647                 globsy.data(), local_sys.n_spins, MPI_DOUBLE,
648                 0, cart_comm);
649     MPI_Gather(sz.data(), sz.size(), MPI_DOUBLE,
650                 globsz.data(), local_sys.n_spins, MPI_DOUBLE,
651                 0, cart_comm);
652     MPI_Gather(energy.data(), energy.size(), MPI_DOUBLE,
653                 globenergy.data(), local_sys.n_spins, MPI_DOUBLE,
654                 0, cart_comm);
655
656     if (mpi_rank == 0){
657         for (int i=0; i<global_sys.n_spins; i++){
658             global_sys.spin.push_back({globsx[i], globsy[i],
                    globsz[i]});
659             global_sys.position.push_back({globpx[i], globpy[i],
                    globpz[i]});
```

```
660                 global_sys.energy.push_back(globenergy[i]);
661             }
662             std::ofstream file(global_sys.filename); // open file
663             Writeoutput(global_sys, file, cart_comm);
664         }
665         }
666         MPI_Type_free(&h_type);
667         MPI_Type_free(&v_type);
668         MPI_Type_free(&d_type);
669         MPI_Barrier(cart_comm);
670 
671         MPI_Finalize();
672         return 0;
673     }
```