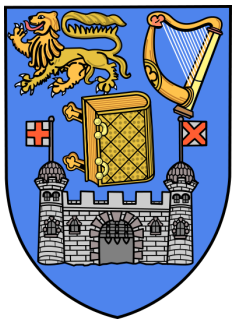

Trinity College Dublin
School of Mathematics

Spin models on random bipartite graphs

Author:

Shane Harding

Supervisor:

Prof. Mike Peardon

Abstract

This is my abstract.

Contents

1 Introduction

This project is centered around doing Ising Model simulations on random bipartite graphs. As such, it is useful to know more the Ising model and graph theory before we get started.

1.1 The Ising Model

1.1.1 What is the Ising Model?

The Ising Model is a mathematical model that was invented by Wilhelm Lenz, but developed by Ernst Ising in 1925. It is used to to explore ferromagnetism in statistical physics.

1.1.2 Analytical tools used for studying the Ising Model

1.1.3 How it is normally studied in serial and parallel simulations

The two dimenstional Ising model is usually represented as a rectangular lattice, sometimes with periodic boundary conditions and sometimes not, depending on the case in question. Each point on the lattice is assigned a spin at the start of the simulation, usually at random. Each point only 'feels' the interaction of its nearest neighbours (each point has four neighbours). These interactions allow a Hamiltonian to be calculated for each point.

FILL ABOUT METROPOLIS HASTINGS AND UPDATING POINTS

In parallel we divide the grid up into smaller subgrids. In the figure, the grid is divided into four subgrids but obviously more subdivisions are possible. Normally we have one subgrid per processor that we intend to run the program on. The need for *Message Passing Interface* (MPI) function calls arises from the fact that on the edges of our subgrids there are points needed for the update step whose values are stored on another processor. These values need to be identified and passed to the relevant processor. MPI is used for this task.

1.2 Graph Theory

Graph theory refers to the mathematical study of *graphs*. A graph is a visual representation of set of objects, known as *vertices*. Some pairs of these obects are then connected by links, known as *edges*. If the edges are said to have orientation (if edge $(a, b) \neq (b, a)$, where a, b are vertices), then we call the graph a directed graph. If the

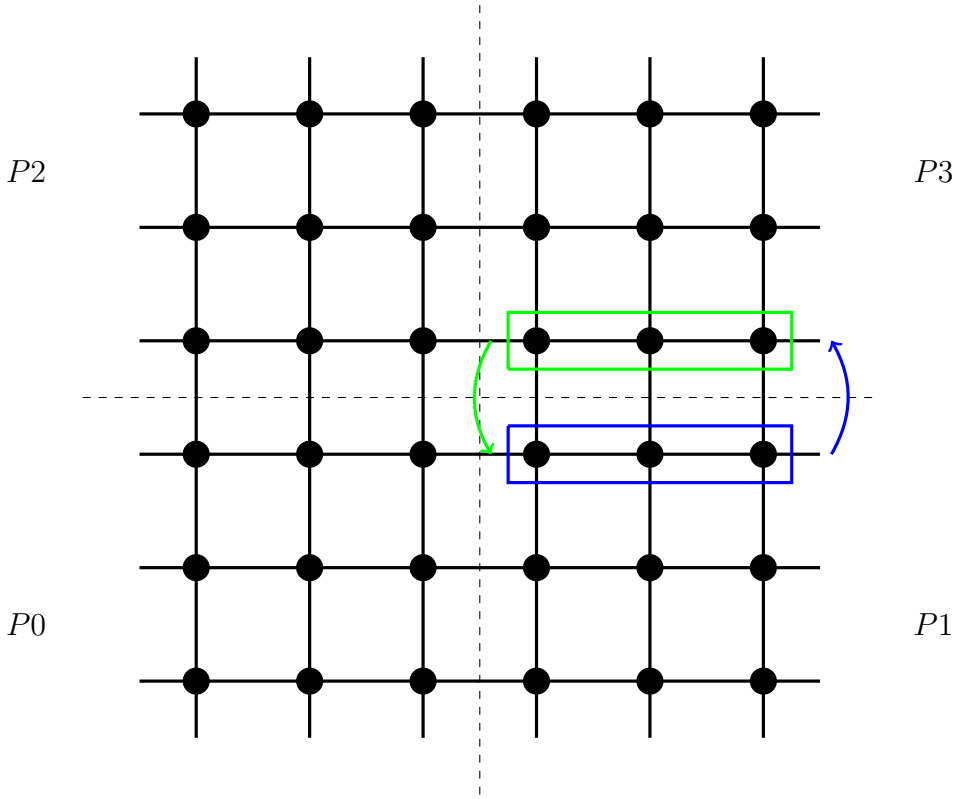

Figure 1: Regular square lattice Ising model.

edges don't have orientation (if $(a, b) = (b, a)$) then we call the graph an undirected graph. We will deal only with undirected graphs in this report.

For this project we're not going to consider disconnected graphs. That is, graphs where there are no nodes connecting a vertex, or set of connected vertices, to the rest of the graph. Only connected graphs are considered.

All graphs considered will be random, *bipartite* graphs. A bipartite graph is a graph in which we can divide its vertices into two disjoint sets, A and B, such that vertices in A are only connected to vertices in B, and vice versa. Disjoint means that the two sets have no element in common.

A random graph is a graph where the edges connect vertices at random; there is no pattern or order to how vertices are connected.

Trivalent graphs are another class of the graphs we will be dealing with a lot. For a graph to be trivalent it means that ever vertex has exactly three edges connecting it to three other distinct vertices.

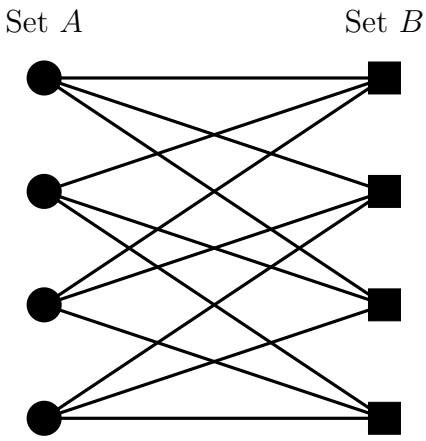

Figure 2: A random bipartite trivalent graph.

1.3 Motivation

1.3.1 Why this is an interesting MPI problem

1.3.2 Physical uses of these simulations

1.4 Aims

2 Software Architecture

In this section I will discuss the various techniques used in creating the simulations that were run in the duration of this project. The main areas these fall under are: graph generating - in both serial and parallel, as well as associated swap algorithms; MPI communication structure; and, finally, how the update step worked.

2.1 Random graph generation

Over the course of this project numerous different methods were implemented and compared for the 'best' way to create a random bipartite graph. These methods were written in both serial and parallel.

2.1.1 Generating random graphs in serial

There were two methods employed to generate random graphs over the course of this project. The first method (which we will call *simple generation*) was not used as often as it could not be generalised as easily to parallel code and is only applicable for generating bipartite, trivalent graphs. The second method is the swap algorithm which is discussed in detail in the following subsection.

The simple generation method works by building the set of neighbours of one of the bipartite sets and then using that to reverse engineer the other. The first set, set A, is generated using the simple rules:

1. Each node in set A must have exactly three unique neighbours.
2. In the set of all neighbours of A each node index of nodes in B must appear exactly three time.

The first rule ensure that the graph is trivalent, from A's point of view, with three unique nodes in B as neighbours. The second rule ensures that each point in B has three neighbours, as it only allows three points in A to connect to each point in B.

This method is works quite well in serial, as it starts at node 0 and fills in its neighbours and then continues all the way down to node $n-1$. The only complication is in filling in the final three nodes where some fairly simple checks must be preformed to ensure that the last three 'slots' to fill in aren't 'illegal' moves. By this, it is meant that filling the slots with the remaining available values would break one of the rules above.

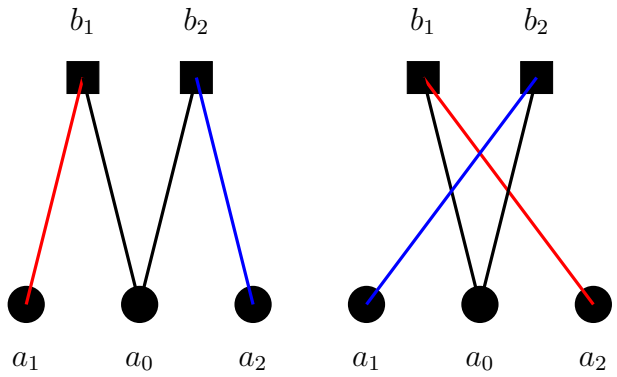

Figure 3: Swap alg.

2.1.2 Swap algorithms in serial and parallel

The swap aglorithm is used for two purposes in this project. The main use is in the generation of random graphs, and a secondary use is as a proposal step during an update (instead of proposing a spin flip a swap is proposed instead) MIGHT DO THIS COME BACK TO.

In the simplest terms the swap algorithm takes two points in one bipartite set and chooses a neighbour of each of the chosed points and then swaps the connectivity, so that the first point is now a neighbour to the point who was neighbouring the second point, and vice versa. The main advantage of using this swap algorithm is that it preserves the properties of the graph. If we start with a bipartite, trivalent graph and do 1000 swaps, say, then the graph will still remain bipartite and trivalent.

The algorithm preserves this because each swap ensures that points in set A remain only connected to points in set B. It also does not change the number of edges coming out of any vertices, this is because it simply two edges are directly swapped with each other between pairs of points so all points involved retain the number of connections they have.

The algorithm

The algorithm works by first choosing a node in set A, at random, which we will call a_0 . From this node two of its neighbours in set B are chosen (again, at random), we call these b_1 and b_2. The next step is to choose one neighbour each of b_1 and b_2, ensuring not to choose a_0 again. Label the neighbour of b_1 as a_1 and the neighbour of b_2 as a_2. At this point we have the nodes selected that we wish to swap. We want to have b_1 not connected to a_1 anymore, but instead have it connected to a_2, and similarly

no longer have b_2 connected to a_2, but connected to a_1.

Before we can do this however we must preform some checks. We must ensure that a_1 is not connected to b_2 by any other edge that we have not considered so far, and similarly we must must ensure that a_2 is in no way connected to b_1. The reason for this check is that we may already have the nodes we wish to swap connected to each other by another edge, so if the swap is performed then it would mean having two edges connecting the same two points to each other, which is not desirable.

Serial Implementation

Implementing the algorithm in serial is a rather straightforward. It follows the description given above and there is no complication involving which point is hosted on which process. The hardest part of writing the code for this algorithm was doing the checks at the end to ensure that the points you were swapping weern't already neighbours through another edge that hadn't yet been considered.

Algorithm 1 Serial swap algorithm

Pick $a_0 = \texttt{rand} \in A$
while $b_1 \neq b_2$ **do**
    $b_1 = \texttt{rand} \in N(a_0)$
    $b_2 = \texttt{rand} \in N(a_0)$
end while
while $flag = 1$ **do**
    while $a_1 \neq a_2$, $a_1 \neq a_0$, $a_2 \neq a_0$ **do**
        $a_1 = \texttt{rand} \in N(b_1)$
        $a_2 = \texttt{rand} \in N(b_2)$
    end while
    if $a_1 \in N(b_2)$ **then**
        $flag = 1$
    end if
    if $a_2 \in N(b_1)$ **then**
        $flag = 1$
    end if
    Do swap:
    $a_2 \in N(b_1)$ and $a_1 \in N(b_2)$
    $b_2 \in N(a_1)$ and $b_1 \in N(a_2)$
end while

Parallel Implementation

The parallel implementation of this algorithm is, naturally, a little more tricky. It is also a problem that does not parallise well at all. In fact, the serial algorithm is much faster and more efficient. The main reason for this is that in the parallel version nodes, and information about what connections they have, are stored on different processors. This means that every time it is needed to pick another node or check a node's neighbours every other processor must wait while one processor does the work. These tasks can't be done simultaneously during the algorithm since each step is directly dependent on the previous step. The problem also can't be avoided by running the algorithm twice (or more) at the same time. This is because, again the whole geometry of the graph must be know in order for the swap to happen, and if one version changes the graph then the other versions don't know about this change yet and so could make a swap that no longer exists on the graph.

Algorithm 2 Parallel swap algorithm

if $rank = 0$ **then**
  Pick $a_0 = \texttt{rand} \in A$
  Determine rank of proc a_0 is hosted on, $rank(a_0)$
end if
`MPI_Bcast(a_0,...,0,...)`
`MPI_Bcast(rank(a_0),...,0,...)`
if $rank = rank(a_0)$ **then**
  while $b_1 \neq b_2$ **do**
    $b_1 = \texttt{rand} \in N(a_0)$
    $b_2 = \texttt{rand} \in N(a_0)$
  end while
end if
`MPI_Bcast(b_1,...,rank(a_0),...)`
`MPI_Bcast(b_2,...,rank(a_0),...)`
Determine rank of proc b_1 is hosted on, $rank(b_1)$
Determine rank of proc b_2 is hosted on, $rank(b_2)$
while $flag = 1$ **do**
  while $a_1 \neq a_2$, $a_1 \neq a_0$, $a_2 \neq a_0$ **do**
    if $rank = rank(b_1)$ **then**
      $a_1 = \texttt{rand} \in N(b_1)$
    end if

if $rank = rank(b_2)$ **then**

$a_2 =$ `rand` $\in N(b_2)$

end if

end while

if $a_1 \in N(b_2)$ **then**

$flag = 1$

end if

if $a_2 \in N(b_1)$ **then**

$flag = 1$

end if

end while

`MPI_Bcast(a_1,...,rank(b_1),...)`

`MPI_Bcast(a_2,...,rank(b_2),...)`

Determine rank of proc a_1 is hosted on, $rank(a_1)$

Determine rank of proc a_2 is hosted on, $rank(a_2)$

Do swap:

if $rank = rank(a_1)$ **then**

$b_2 \in N(a_1)$

end if

if $rank = rank(a_2)$ **then**

$b_1 \in N(a_2)$

end if

if $rank = rank(b_1)$ **then**

$a_2 \in N(b_1)$

end if

if $rank = rank(b_2)$ **then**

$a_1 \in N(b_2)$

end if

2.2 General Overview of Parallel Simulation

In this section a general overview of how the code written for this project works. In the following sections some of the more complicated steps will be explained in greater detail.

The program is first initialised with MPI on the required number of processors, `num_proc`. The number of nodes, `num_nodes` in the graph to be solved is determined. Each bipartite set gets half of these points.

Two typedef'd struct called `Array` is initialised on each CPU, one is for the nodes in set A, the other for the nodes in set B. This structure contains the total number of nodes in the set, the local number of nodes, the offset of the array and a list of all the neighbours of the nodes held localy.

```
typedef struct
{
  int x; // global number of nodes in set
  int x_local; // local number of nodes in set
  int x_offset; // Array offsets
  int **neighbour; // array of neighbouring points
}
Array;
```

The element `**neighbour` is a two dimensional array. Its size is determined at run-time by the size of the graph, number of processors and the number of neighbours each point has. The first index, i, is the local index of a node in the array, the second index, j, is the index of i's j^{th} neighbour. So if we have an array for set A denoted simply by `a` then `a->neighbour[i][j]` gives the global index of the j^{th} neighbour of i. i's global index is given by: `i_global = a->x_local*a->x_offset + i`.

The array is initialised into a double ring configuration by default. The reason for this is that it is very simple to set up. Each node's neighbours are simply the points with global indices $n-1$, n and $n+1$ all in set B, if we are considering the node with global index n in set A.

The next step is to run the swap algorithm, as described in the previous section, a sufficient number of times in order to randomise the graph.

A similar struct to `Array`, called `Field` is then initialised. Like the array struct it is initialised on ever processor twice, once for nodes in A and once for B. This struct has three elements. The first is the value array. This is an array of length `a->x_local`. It holds the value of the spin each node has. Spin is denoted as $\uparrow = +1$ and $\downarrow = -1$.

```
typedef struct
{
  int *value; // vals for points on local process
  int *halo_count; // no of points recived from each proc
  int **halo; // vals of data recieved from each proc
}
Field;
```

The `*halo_count` array is the same size of the number of processors that the code is running on. It is a counter that measures how many data points will be sent from each other process to the host process. For example `if(host.rank == 0) f_a->halo_count[2] = 7`

means that if the host process is rank 0, then it expects to be sent 7 data points from process 2.

Finally, the two dimensional array `**halo` is the array that actually holds the values that are sent to the host process from the other processors. The first index of this array denotes the process from which the data is to be received from. The second index is the counter for the recieved data. Continuing with the example from the last paragraph, if we have `if(host.rank == 0) f_a->halo_count[2] = 7` then `if(host.rank == 0) f_a->halo[2][j]` has 7 elements, where j determines which value is which, when the data is received.

The number of data points that need to be sent from each processor to every other processor is then computed (more on this method later). The next step is to malloc space for the points needed from other procs to be stored locally on the host proc.

At this point it is known on each processor how many nodes are needed from each of the other processes in order to complete an up date step. Before each update step is started, all the values that must be sent from one process to another must be completed. This sending of data requires loading the relevant values into a send buffer, sending them to their correct destination with an MPI call. These values are received into a receive buffer and are then copied into the the halo array of the field struct (all of this is explained in more detail further on). The update step then calls the values from this halo when they are needed.

2.3 MPI communications

2.3.1 Data division and load balancing

The problem of how to split the data most efficiently between computing cores will be handled in this section. When we want to solve the system in parallel we have to give each processor a certain number of points on the graph. The method used to divide the data was to simply divide the nodes equally among the processors. This may seem, at first glance, as though it isn't the most efficient way to distribute the data in order to minimise the MPI communication overhead of the calculation. However, it is actually the fairest and easiest way to balance each processor's load and have a, relatively, uniform spread of MPI communications between processors.

This is the fairest method because if the graph is truly random it is expect that each processor needs to communicate with, on average, the same number of nodes on each of the other processors. This division then ensures that each processor sends the same (on average) data points from itself to each other processor.

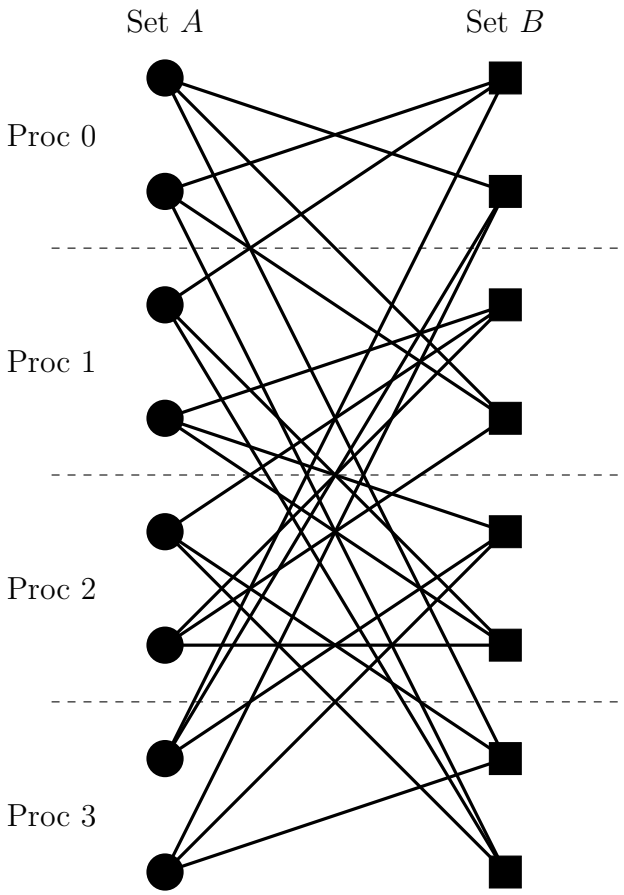

Figure 4: Data division for a random trivalent, bipartite graph.

The alternative to this method is to do some sort of profiling and then preconditioning of the graph and connections. The purpose of this would be to minimise the amount of data that needs to be sent between nodes. This would be a very complicated algorithm that would most likely require a lot of computation time. This large addtional overhead would greatly overshadow any of the benefits gained in the actual computation.

2.3.2 Setting up Halo Regions

To understand why the halo region is needed and how it works, it is easiest to use a picture. Consider the graph shown in figure FILL. It shows two bipartite sets each with six nodes. The graph is divided over three processors. We are going to consider performing an update on the nodes in set A hosted on processor 1. The figure only shows the edges that connect the nodes in set A on process 1 to nodes that are not on process 1. All other nodes are not shown as they aren't necessary in understanding the method.

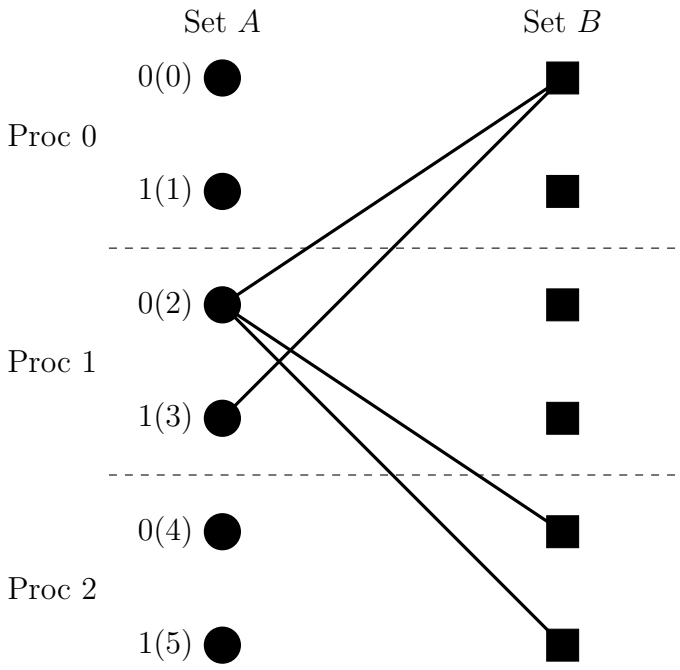

Figure 5: FILL

We start by looking at the nodes in set A, on process 1, and where this data is needed. First we looping over the array of neighbours of nodes in set A. First the neighbours of point 0 (global index 2) are examined. It has neighbours 0, 4 and 5 (global indices). Point 0 is on process 0, and points 4 and 5 are on process 2. So at this stage, on process 1 we know that `send_count[0]=1` and `send_count[2]=1`.

Since point 0 is connected to both 4 and 5 in set B we have a double count which has to be ignored. Once we know that we need to send a point's data to another proc it is marked that it is needed and will not be sent again. To keep track of double counts like this we need a two dimensional array `**double count`. The first index is for the local index of a node and the second is for the process that the data is to be sent to. So for node 0 on process 1 we have `double_count[0][2]=1`. Initially this is value is zero to say that point is not needed on process 1, but when we find that point 4 needs that value we set the double count entry to 1 to mark the value as counted for. So then when the program realises that point 5 also needs point 0 from process 1 we do not need to copy it again because we have marked it as copied already.

Then point 1 is looked at. Its only connection to another process is to point 0 (global index). Point zero is on process 0 and we have that `double_count[1][0]=0` so we are free to increase the send count to process 0: `send_count[0]=2`.

That is all the neighbours looped over and we know how many points process 0 has to send to each other process. Importantly, there has been no double counting, while

it may not have been a big deal in this small example, but for a bigger graph it could mean allocating space for thousands more data points and then performing MPI calls for all of the double counted values. This would add up very quickly over the course of a big simulation and lead to an unnecessarily large MPI communication overhead that would have a big performance impact.

Below is the code snippet for this procedure.

```
for(i=0;i<a->x_local;i++){
  for(j=0;j<3;j++){
  proc = a->neighbour[i][j]/a->x_local;
    if(double_count[i][proc] == 0){
      // determine if neighbour is on a different proc
      if(a->neighbour[i][j] < host.rank*a->x_local || a->
          neighbour[i][j] >= (host.rank+1)*a->x_local){
        send_count[proc]++; // increase count if on diff proc
        double_count[i][proc] = 1;
      }
    }
  }
}
```

At this stage it is known on process 1 how many of its local data points need to be sent to other processors. However, the other processors do not yet know how much data they will be receiving from every other process. For example process 0 does not yet know that it will be 2 data points from process 1. The recevie count array for each process tell it how many data points will be received from each other process. For example, on process 0 `recv_count[1]` holds the number of data points that are to be received from process 1. These receive counts should match the send counts on the sending process with the destination process as the index. So on process 1 `send_count[0]=2`, therefore we must have `recv_count[1]=2` on process 0.

Instead of doing some sort of similar counting algorithm as was done to find the send count, it is much easier to use the `MPI_Alltoall` command. This command will send take an array in the send buffer and a send count, x. It then sends the first x elements of the array to process 0 the second x to process 1 and so on. A receive count is also specified with a receive buffer telling the program what to do with the received data.

```
MPI_Alltoall(send_count, 1, MPI_INT, recv_count, 1, MPI_INT,
    MPI_COMM_WORLD);
```

Now that each processor knows how many data points it will receive from every other process the memory can be allocated to store this incoming data:

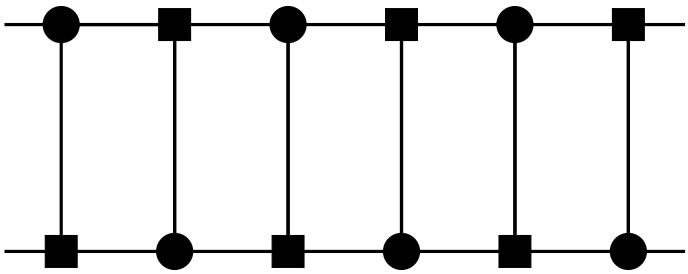

Figure 6: Double ring bipartite graph.

```
for(i=0;i<num_proc;i++)
  f_a->halo_count[i] = recv_count[i];

f_a->halo = (int**)malloc(sizeof(int *)*host.np);

for(i=0;i<num_proc;i++){
  if(f_a->halo_count[i] != 0){
    f_a->halo[i] = (int*)malloc(sizeof(int)*f_a->halo_count[i]);
  } else {
    f_a->halo[i] = NULL;
  }
}
```

2.3.3 Sending Data Between Processors

3 Results

3.1 Serial code

3.1.1 Double Ring Graph

3.1.2 Random Bipartite Trivalent graph

3.2 Swap Algorithm

In this section the performance of the swap algorithm is analysed. The serial version of the algorithm will be compared to the parallel algorithm for a varying number of cores. The number of swaps required to make a graph random will also be studied.

3.2.1 How Many Swaps makes a Random Graph?

Before a graph is randomised for a simulation it is first initialised in an order that is fast and easy to compute. This is normally just the double ring cofiguration, where ever node has neighbours of the node with the same index, one ahead and one behind in the other bipartite set. From this initialisation we have to randomise the graph using the swap algorithm.

The problem here is knowing how many swaps need to be made before all traces of this original configuration are lost and the graph can be considered random. We know that initially every points neighbours are the three closest to it in our indexing system. So, if we have point n in A we know that its neighbours are $n-1$, n and $n+1$ in set B.

If a graph is truly random then it is expected that given any node in set A, the probability of it being connected to any given node in B is simply $\frac{1}{n_B}$, where n_B is the number of nodes in B. So this means that if we divide the graph in half horizontally, as in the data division figure, we would expect, on average, half of our connections to cross this horizontal line.

In other words, if we think of how we divide a graph amongst processors by giving each processor an equar fraction of the nodes, we would then expect each processor to have the same, on average, connections to each other processor. This is easiest to think of with an example. Say we have a bipartite, trivalent graph where $n_A = n_B = 1024$, so there are 2048 nodes in total. As each node in A has three edges connecting it to nodes in B, the total number of edges in the graph is $3 \times 1024 = 3072$. If the graph is then divided by 4 processors, then there are $\frac{3072}{4} = 768$ edges on each process.

Now, it is expected at this stage that the graph is random, therefore each of these

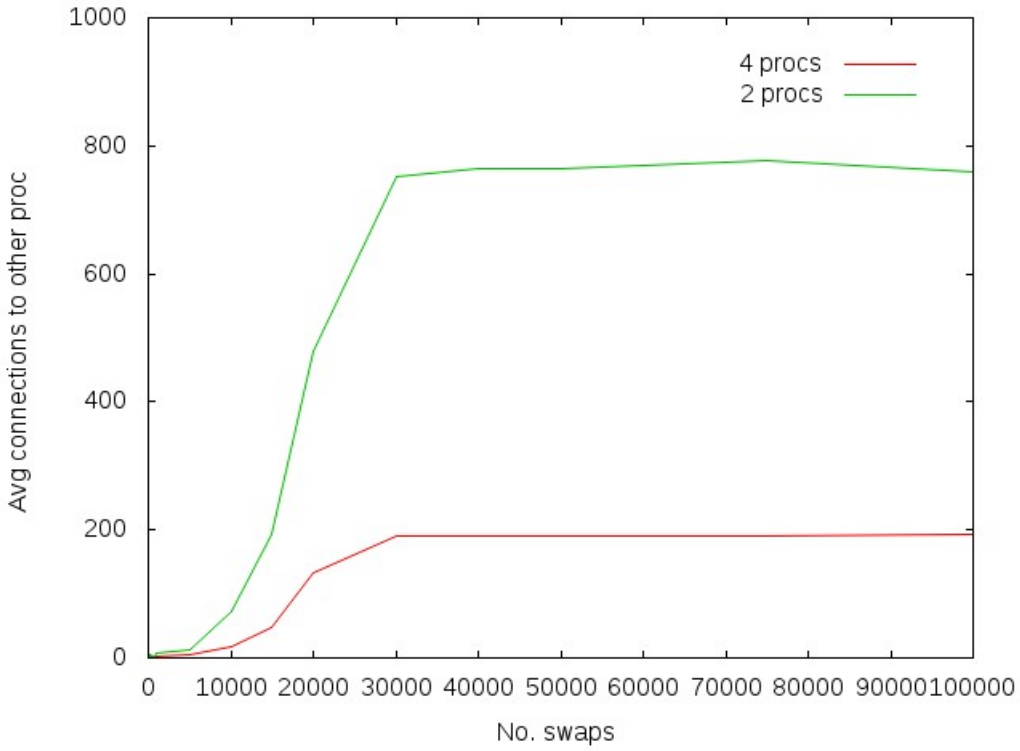

Figure 7: This plot shows how the number of connections to different processors changes with the number of swaps performed. The graph used was a bipartite, trivalent graph with 1024 nodes in each set (2048 nodes in total). It was run on both 2 (green line) and 4 (red line) processors.

768 edges on a process is equally likley to be on any of the 4 processors. So we expect $\frac{768}{4} = 192$ connections on each process.

Code was written to simulate the exact situation described above and to determine how many swaps it takes to reach a random graph with no trace of its initial configuration.

While the last simulation is useful for generating random graphs, it is also interesting, and possibly more useful, to examine how many swaps per node it takes for a graph to be considered random. This is interesting as this may be different for different sized graphs. To investigate this, the same simulation as above was run for varying graph sizes and a similar plot was made.

3.2.2 Serial Vs. Parallel

3.3 Speedup

3.4 How

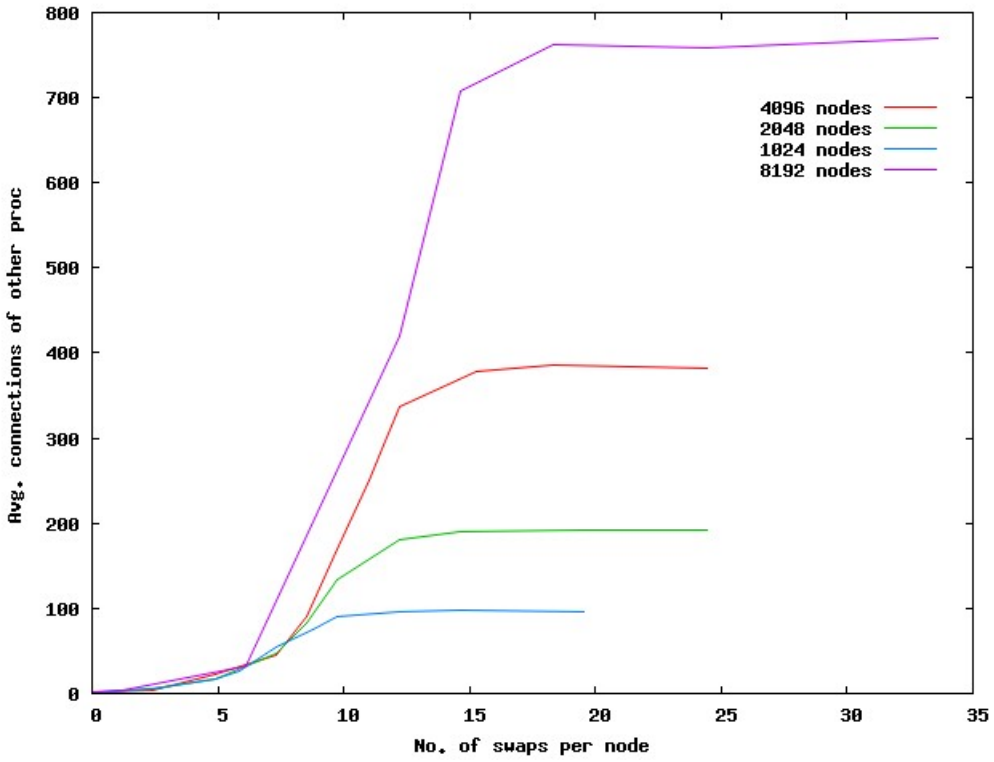

Figure 8: This plot shows how the average number of connections to another processor changes with the number of swaps performed per node. The graph used was a bipartite, trivalent graph with four different graph sizes (1024, 2048, 4096 and 8192). All were run on 4 processors.

4 Conclusion

hello

5 Future Work