A performance study of a template C++ class for parallel Monte Carlo simulations of local statistical field theories on a three dimensional lattice

_Liam Burke_

_in partial fulfilment of the Master Of Science in High-Performance Computing_

Supervised by

Mike Peardon
School Of Mathematics

2019/20
Declaration

I hereby certify that the work presented in this thesis which I submit in partial fulfillment of the degree Master Of Science in High-Performance Computing is entirely my own work and has not been taken from the work of others unless otherwise cited and acknowledged within the document.

Liam Burke
Sunday 20th September, 2020
Acknowledgements

I would firstly like to thank my supervisor Mike Peardon for all of his help and guidance throughout this work and throughout the year as the course coordinator for the M.Sc in High Performance Computing. Secondly, I owe a great deal of gratitude to Richard Morrin for consistently helping me with the many questions I asked him in relation to the C/C++ bugs I encountered throughout this work. I would also like to thank the team at the Trinity Centre for High Performance Computing for allowing me to make use of their computing resources which were essential in completion of this work. Finally, I would like to thank my family, who’s continued support have allowed me to complete this M.Sc.
# Contents

1 Introduction  
1.1 Introduction to numerical simulations  

2 Generic programming in C++  

3 The C++ lattice class  
3.1 Class constructor  
3.2 Parallelism in the class  
3.3 MPI exchange function  

4 Performance considerations and optimizations  
4.1 Memory layout and communication overhead  
4.1.1 Striped decomposition  
4.1.2 Cubic decomposition  

5 Testing the class  
5.1 The Ising model  
5.2 Metropolis algorithm  
5.3 Red/black updates  
5.4 Ising model simulation results  
5.5 Scalar quantum field theory on the lattice  
5.6 $\phi^4$ simulation details and results  

6 Performance tests  
6.1 Striped decomposition - strong scaling for int type matter fields  
6.2 Cubic decomposition - strong scaling for int type matter fields  
6.3 Striped decomposition - strong scaling for double type matter fields  
6.4 Cubic decomposition - strong scaling for double type matter fields  

7 Conclusions and future additions to the Class
Abstract

In this thesis we investigate the performance properties of a template C++ class designed to run parallel Monte Carlo simulations of local statistical field theories on a three dimensional lattice. The generic nature of the class allows for data type flexibility when defining the matter fields at every site of the grid while incorporating this flexibility into a generic MPI exchange function to allow for correct data transfer in parallel simulations. This allows one to overcome the book-keeping issues associated with the parallel software development of code needed to run simulations of different field theories with different physical properties. We will investigate how the system performs in parallel by looking at its scaling behaviour for different matter fields and will examine the factors affecting its performance - particularly in relation to how the lattice is stored in memory and how this could influence an optimal choice of MPI decomposition. To test the performance of our class, we will run parallel simulations of the 3D Ising model and investigate its critical behaviour by computing observables such as average magnetization per spin, magnetic susceptibility and its average energy. The case of a double precision data type on the lattice sites will then be tested by running simulations of $\phi^4$ quantum field theory in $2+1$ dimensions using its discrete lattice action. We will run simulations in both the Ising limit and the limit of a free scalar field theory, and examine how the mass of a particle behaves as the action parameters are varied. We compare our results to that obtained in the literature and comment on the similarities to our Ising model results.
1 Introduction

1.1 Introduction to numerical simulations

In recent decades, numerical simulations through Monte Carlo methods have emerged as being one of the most powerful tools used to study the behaviour of physical systems. From understanding how ferromagnetism arises from a collection of interacting spins in the well known Ising Model [7,8], to the study of Quantum Field Theory [2,9] - these methods of computer simulation have become increasingly important in working as a theoretical tool used to study the properties physical phenomena.

The starting point of any Monte Carlo simulation typically involves the discretization of some continuous model onto a discrete finite lattice, whose discrete nature then lends the problem naturally to a computer simulation [2]. The model we are studying on the lattice is characterised by some function $\phi$ which associates to every point on the grid $x$ a quantity $\phi(x)$ which takes on different states depending on the properties of the system in question. In the simulation we typically start off with some initial configuration of $\phi$ giving some value at every point of the grid. We then attempt to follow the stochastic evolution of the model, who’s behaviour depends on a sequence of random numbers generated during the simulation. This sequence of configurations generated allows us to sample a region of phase space in order to estimate properties of the model by calculating thermal averages of the observables of interest using the values of $\phi$ at every point of the lattice from each configuration in the sample. As mentioned, one of the most simplest and well studied examples of an application to Monte Carlo simulations is the study of the Ising model, a simple model of ferromagnetism named after Ernest Ising [7], where on each lattice site we have we have $\phi(x) \in \{-1, +1\}$ representing one of the two possible directions of the interacting magnetic dipoles of a collection of atomic spins. The properties of the system can be measured by sampling from a collection of configurations of these spins. Other types of field theories however may require that $\phi(x)$ take on a more general structure at every lattice site. For example the study of scalar field theory on the lattice [5,9] for which instead of two integer values, can take on a
continuum of values given by \( \phi(x) \in \mathbb{R} \) such as in simulations of \( \phi^4 \) quantum field theory [5]. The degrees of freedom in Lattice QCD [4,10] simulations are even more complicated. Here, one attempts to make theoretical predictions of the properties of hadrons through numerical simulations of Quantum Chromodynamics on a four dimensional space time lattice where we have fields representing quarks placed on lattice sites while the gluon fields are defined on the links connecting those sites and represented as an SU(3) matrix. These theoretical predictions have played an important role as a precision test of the Standard Model of particle physics through comparison with results obtained in collider experiments, and have in many cases agreed with extraordinary precision. For example the mass of a proton has been determined theoretically through computer simulations of QCD with an error of less than two percent [6]. And so it is clear that Monte Carlo simulations have an important role across a wide range of applications.

In this thesis, we will use C++ templates to develop a toolkit which allows us to run parallel Monte Carlo simulations of different types of field theories on a three dimensional lattice by allowing us to vary the data type of the matter fields placed at every site of the grid. The parallel nature of the class will allow us to overcome much of the computational overhead associated with running large scale Monte Carlo simulations on large lattices, while the template nature of the class will allow for datatype flexibility when defining the matter fields at every lattice site. It would also aid in decoupling the book-keeping issues associated with the parallel software development of code needed to run simulations of different field theories with different physical properties. A common example of which would be an MPI exchange function whose implementation would need to be rewritten to deal with different MPI Datatypes depending on the type of field variables. Our template framework however offers a generic MPI exchange which functions independently of the data type defined on every lattice site.

The main question we hope to answer in this thesis is how well a template system performs in parallel by looking at its scaling performance for different matter fields. We are particularly interested in examining the factors affecting its performance -
particularly in relation to how the lattice is stored in memory and how this could influence an optimal choice of MPI decomposition. It is well known that for a three dimensional grid, only two of its surfaces can be stored contiguously in memory, and for this reason we developed two implementations of the class. The first considers an MPI typology for which there is one one direction of data exchange, allowing us to exchange only the two contiguous surfaces. The second allows for a more complex MPI decomposition which can exchange data from all six surfaces and requires additional methods to pack fragmented data into a contiguous array before sending, and then unpack this data into the correct ghost surface on the receiving MPI process. We will investigate for which lattice sizes and data types these two implementations becomes an optimal choice. To test the performance of our class, we will run parallel simulations of the 3D Ising model and investigate its critical behaviour by computing observables such as average magnetization, magnetic susceptibility and its average energy. This will test the case of an int type instance of the class and the case of a double datatype will be tested by running simulations of $\phi^4$ quantum field theory in $2 + 1$ dimensions using its discrete lattice action. We will run simulations in both the Ising limit and the limit of a free scalar field theory, and examine how the mass of a particle behaves as the action parameters are varied. We will look at the critical behaviour of the model and compare our results to that obtained in the literature.

2 Generic programming in C++

Generic programming is a concept by which we can reuse the same code to work on different data types. In C++ this can typically be achieved using templates. Templates allows for functions and classes to have the flexibility to act on multiple data types. So instead of writing a separate function for each datatype we could instead write just one function using templates so as to avoiding code duplication. This in turn could be of great advantage if one wants to write code to perform the same operations on multiple data types using reusable code. As a trivial example suppose we wanted to write a function which returns the max of some pair of numbers.
Without templates, this function would have to be written multiple times to deal with different data types. The first of these functions might be used to deal with the case of an int data type, but a second would have to also be implemented to deal with the case of a double as below.

```cpp
// a max function for "int" type
int max( int a, int b)
{
    if ( a > b )
        return a;
    else
        return b;
}

// a max function for "double" type
double max( double a, double b)
{
    if ( a > b )
        return a;
    else
        return b;
}
```

Figure 1: A simple example of how one may have to re-write the same max function multiple times to deal with different data types

This type of code reuse quickly becomes inefficient and can be avoided with use of templates which allow us to achieve the same functionality with just one function capable of working on a range of data types like the function below.

```cpp
template <typename T>
void swap( T& x, T& y)
{
    T temp = x;
    x = y;
    y = temp;
}
```

Figure 2: A template max function capable of working on a range of different data types
3 The C++ lattice class

The purpose of this template C++ lattice class is to be run generic Monte Carlo simulations of statistical field theories in parallel on a three dimensional lattice with periodic boundary conditions. It is defined by

\[
\text{template <class T> class lattice}
\]

where T defines the data type of the field at every point of the grid. This allows the class to work for a range of different data types on the lattice and thus makes it capable of running simulations of a range of different field theories. For example, a code to simulate the Ising or the q-state Potts model might use the template instantiation

\[
lattice <\text{int}>
\]

while a scalar field theory would use

\[
lattice <\text{double}>
\]

We will now discuss the characteristics of the class and talk about its constructor and member functions.

3.1 Class constructor

The function of the class constructor is to dynamically allocate memory to store the three dimensional lattice. This was done by having three layers of pointers, but still having all the values of the lattice sites stored in one contiguous chunk of memory. We have a member variable to store the grid

\[
\text{T } *** \text{ data ;}
\]

and we allocate the memory inside the constructor
lattice(int nx, int ny, int nz): xdim {nx}, ydim {ny}, zdim {nz}
{
    xghost = xdim + 2;
    yghost = ydim + 2;
    zghost = zdim + 2;

    data = new T **[xghost];
    data[0] = new T *[xghost*yghost];
    data[0][0] = new T [xghost*yghost*zghost];

    for (suto i = 0; i < xghost; i++)
    {
        data[i] = data[0] + yghost*i;
        for (auto j = 0; j < yghost; j++)
        {
            data[i][j] = data[0][0] + zghost*j + i*zghost*yghost;
        }
    }

    packagedBottomSurface = new T [xghost*zghost];
    packagedTopSurface = new T [xghost*zghost];

    recieveFromTop = new T [zghost*zghost];
    recieveFromBottom = new T [xghost*zghost];

    packagedFrontSurface = new T [xghost*yghost];
    packagedBackSurface = new T [xghost*yghost];

    recieveFromFront = new T [xghost*yghost];
    recieveFromBack = new T [xghost*yghost];
}
Dynamically allocating memory in this way has two advantages. It means we can guarantee that two faces of the grid will be stored contiguously in memory. This is of great advantage if we just wanted to consider an MPI exchange of data in one direction only as it means we do not have to worry about exchanging non-contiguous data. Secondly we can still access the elements of the lattice using the standard notation

\[ \text{data} [i][j][k] \]

which is actually equivalent to accessing via

\[ \text{data} [0][0][i*yghost*zghost+j*zghost+k] \]

This also allows the user to change how it is stored in memory later on if they wish but without having to change our actual client code. A simple diagram to illustrate the allocation of the 3 dimensional lattice is shown in the figure below. The actual data values are on the last layer in one contiguous chunk of data, while the two pointer layers act as helpers to access each of the data elements.

![Figure 3: The memory allocation layout for the 3D lattice](image)

We next move to talk about how the class supports parallelism as well as how we implemented the templated system into an MPI exchange routine through use of member functions.
3.2 Parallelism in the class

The most common approach to the parallelization of these types of Monte Carlo simulations on a grid is through domain decomposition where each process updates its own subsection of the lattice as the system moves from one configuration to the next. The class allows for this type of parallelism in the following way. First we have a one dimensional decompose function external to the class. This function takes in the size of an array and the number of processes and divides that array among the processes and returns the starting and end address for each process denoted $s$ and $e$ respectively. After determining the process topology from the MPI cart shift and MPI cart create functions, we can call this function three times to decompose all dimensions of the lattice among each process. Assume after process $p$ has called the decompose function three times and owns the section of the full lattice with start addresses $s_1, s_2, s_3$, and end addresses $e_1, e_2, e_3$ in each dimension. It can then declare its own instance of the class like

```cpp
lattice<double> *A {new lattice<double> {e1−s1+1, e2−s2+1,e3−s3+1};
```

and the grid stored on this process would then represent its own sub grid. The propose of the $+1$ in the above instantiation is because the decompose function returns a 1 instead of 0 in the starting address and so we require an additional element to store the first element of the vector. Inside the constructor, an extra two elements are added to the end of each dimension meaning that each MPI process now has its own sub grid with an additional ghost surface on every face of the lattice. This ghost surface serves as an extra layer of data on top of every face of the grid to which neighbouring MPI processes can send relevant data. This is because when a process is updating lattice points on the boundary of its sub grid it requires the neighbouring points which are owned by another process and to which it has no access. The ghost surface then acts a place the neighbouring process can send this data to to allow for the current process to have access to it. In order to exchange the data the class has it own MPI exchange function
3.3 MPI exchange function

The purpose of the MPI exchange member function is to have all the necessary MPI send and receive messages needed to correctly exchange data between processes. These MPI functions require us to specify the MPI data type involved in the exchange which, due to the template nature of the class can vary between different class instances and different types of simulations. In order to incorporate this type of generality into an MPI exchange we need to ensure the MPI exchange function will also be generic and input the correct MPIDatatype into the send and receive functions depending on the original standard C++ type defined at each site of the lattice. To achieve this the class contains a function

\[\text{MPI
datatype lattice } <T>::\text{get_mpi_type}()\]

This function uses the C++ build in function to return a const char used to indicate the original template type T like

\[\text{const char* local\_type = typeid(T).name();}\]

And the function then returns an MPIDatatype after checking which char was obtained. For example, if the user declared an instance of the class of type int then in this function an "i" would be returned and a variable of type MPIDatatype would then check to see which char was returned and then knows an MPI\_INT is needed in the exchange function. For this reason the exchange function contains the line of code

\[\text{MPI
datatype mpi\_type = get\_mpi\_type();}\]

And the variable \text{mpi\_type} is inputted into each MPI send and receive function in the exchange and so the class can deal with a general exchange for a wide range of types.
4 Performance considerations and optimizations

4.1 Memory layout and communication overhead

In order to begin thinking about questions regarding the performance of the class, the first thing to consider is how the lattice is stored in memory. For a three dimensional grid we can guarantee that two faces will be stored contiguously, the rest will not. This will be an important point to consider when we consider different types of MPI decomposition as we shall see. Consider a simple example of a $3 \times 4 \times 5$ lattice as shown in the diagram on the next page. Six copies of the lattice are shown and on each copy a different surface is highlighted in green. We see that only the two first two surfaces of the lattice are stored in a contiguous block of memory while rest of the surfaces are all fragmented in some way. This is an important property of the memory allocation of a 3 dimensional lattice we need to consider when looking at optimal ways to decompose the lattice among MPI processes. For example, we may decide to decompose the lattice in such a way so that we only need to exchange the two contiguous surfaces. Although this will have the advantage of avoiding the overhead of exchanging non contiguous surfaces, we may decide to use more complex MPI topology in order achieve finer parallelism at the expense of additional overhead in having to exchange all surfaces of the lattice including those that are fragmented. A decomposition of this type would require us to pack up any fragmented data into one contiguous vector on the sending MPI process, send the data, and unpack the contiguous vector back into the relevant ghost surface on the receiving MPI process. To determine when each of these types of decomposition becomes an optimal choice will be the main avenue of research in this thesis. For this reason we implemented two versions of the class - a striped decomposition and a so called cubic decomposition.
Figure 4: An example of a $3 \times 4 \times 5$ lattice. 6 copies of the same lattice are shown in the diagram with a different surface highlighted on each copy - highlighting the fact that we can guarantee two surfaces will be stored contiguously in memory while the other surfaces will be stored non-contiguously.
4.1.1 Striped decomposition

The first type of decomposition we proposed to test the performance of is a "striped decomposition", where we divide the lattice in one direction only so that each process has a neighbour in only one direction (left/right). This means that each process needs to exchange data between its left/right neighbour only. From the argument in the previous section, because we can always guarantee two surfaces will be stored contiguously in memory, this decomposition will be a good choice provided the two surfaces we are exchanging are the two contiguous faces. A simple illustration can be seen in the diagram below.

Figure 5: Striped decomposition of the lattice among 4 processors. Communication is in one direction only (left/right). For example process 2 needs to communicate data on its left surface with process 1, and on its right surface with process 3. It also needs to receive data from process 1 right surface and process 3 left surface.

4.1.2 Cubic decomposition

A more general decomposition strategy we considered which we refer to as a "cubic decomposition" involves decomposing the lattice in all three spatial dimensions. This means each process has a smaller subgrid to work on, but more neighbours in every direction with which it needs to exchange data. In this case each process now has a
left/right, up/down and forward/back neighbour, and due to the memory allocation argument presented above, this means we cannot avoid the difficulty in the sending of non-contiguous surfaces between processes. This can be visualized using the diagram below of a general cubic decomposition.

Figure 6: Cubic decomposition of the lattice among 8 processors to enable communication in all directions. Each process has a left/right, up/down and top/bottom neighbour with which it needs to exchange data.

In order to consider the best approach to dealing with this type of exchange of fragmented data we refer again to figure 4 on page 16, and extend the idea to a general lattice of size $M \times N \times O$. As noted, we have two surfaces stored contiguously in memory shown by the first two grids in the diagram. The first is "right" surface given by the first $N \times O$ elements and the next a "left surface" given by the last $N \times O$ elements. We see the next two surfaces are in general non-contiguous but do come in contiguous blocks. The first is a "top" face for which we have the first $O$ sites stored contiguously, then a break of $(N - 1) \times O$, then another $O$ sites stored contiguously, followed by another break of $(N - 1) \times O$. This pattern then repeats itself $M$ times.
in total. For the bottom face we have the reverse, a break of \((N - 1) \times O\), then \(O\) and so on. Finally, we have the last two surfaces completely fragmented. For the "front" face we have the first site, then a break of \(O - 1\), then the next site and so on. There is \(M \times N\) of these sites and the reverse holds for the other back surface. Understanding these patterns have been essential in developing different ways to deal with the exchange of the fragmented data in the MPI exchange function. We considered a few approaches which we outline below.

- For the two fragmented surfaces which come in contiguous blocks we tried two different approaches. The first involved using two for loops in the MPI exchange function. The outer one looped through each of these blocks while the inner one looped through the elements of each contiguous chunk. We then attempted to send and receive each element individually which involved a total of \(M \times O\) sends and \(M \times O\) receives. The second approach we used involved for each block in the outer loop, we packed all of its elements into a single vector and sent each block separately and unpacked the data on the other side. This involved a total of \(M\) sends and \(M\) receives.

- For the two completely fragmented surfaces we tried using one for loop which loops through the elements and sends and receives each element individually - a total of \(M \times N\) sends and \(M \times N\) receives.

After first implementing these approaches I found that the performance was too poor due to communication overhead from too many MPI sends and receives, particularly across inter network communication. A more optimal approach hover involved pack and unpack member functions to pack up the entire surface on the sending process before sending and unpack the data on the receiving process. For example, suppose we wanted to send the top and bottom surfaces of the grid to the process above and below respectively. Instead of performing multiple sends and receives as in the previous approaches we instead map the entries of the surface to just one vector whose elements are stored contiguously in memory. We then send that vector over to the receiving process and unpack it on the other side. Meaning that only one send
and one receive is necessary. The second implementation of our class then contained these additional methods to deal with the cubic decomposition. Packing up the top/bottom surfaces require storing the data in two vectors of the same dimension and so can be implemented in the same pack method like

```cpp
template <typename T>
void lattice<T>::packTopBottom()
{
    for (auto i = 0; i < xghost; i++)
        for (j = 0; j < zghost; j++)
            packagedBottomSurface[i * zghost + j] = data[i][1][j];

    packagedTopSurface[i * zghost + j] = data[i][yghost - 2][j];
}
```

We then send the contiguous vectors packagedBottomSurface, packagedTopSurface to the processor above and below and unpack into the ghost surfaces on the receiving processes using

```cpp
template <typename T>
void lattice<T>::unpackTopBottom()
{
    for (auto i = 0; i < xghost; i++)
        for (j = 0; j < zghost; j++)
            data[i][0][j] = receiveFromBottom[i * zghost + j];

    data[i][yghost - 1][j] = recieveFromTop[i * zghost + j];
}
```

Although this was the approach we found most optimal in implementing the cubic decomposition, it still has an additional overhead to the striped decomposition by having to use using these functions at the expense of finer grained parallelism across the grid. We will compare the performance of both our class which imposes a striped decomposition, and the class which allows for a more general cubic decomposition and requires additional methods to pack fragmented data. We will do so by looking at its scaling behaviour for different data types of every site of the grid. The class
diagrams for both implementations are shown here.

<table>
<thead>
<tr>
<th>Class diagrams for both striped and cubic decompositions</th>
</tr>
</thead>
<tbody>
<tr>
<td>lattice (striped)</td>
</tr>
<tr>
<td>lattice()</td>
</tr>
<tr>
<td>MPIvoid Datatype getmpidatatype()</td>
</tr>
<tr>
<td>void exchange3D(...)</td>
</tr>
<tr>
<td>void packTopBottom()</td>
</tr>
<tr>
<td>void unpackTopBottom</td>
</tr>
<tr>
<td>void packFrontBack()</td>
</tr>
<tr>
<td>void unpackFrontBack()</td>
</tr>
</tbody>
</table>

In the next section we will discuss the details of the simulations we used to test our class and comment on the performance improvements we made in these simulations in the methods external to the class by exploring the nature of how the class itself was implemented.

**Remark.** At the beginning of this work it was intended that each process store a copy of the full grid and only update its own subsection. However this implementation was found to introduce more difficulty when dealing with fragmented data as well as being inefficient in terms of memory usage and was not considered further.
5 Testing the class

In this section I will present the details of the types of simulations that were used to test the class. We begin by first testing the template class has correct built-in type deduction, meaning that the class works when different instances are created with different data types. This ensures the template class works for different types of field theories with different matter fields by ensuring the MPI exchange function exchanges the correct data to the correct ghost cells. We will then make sure results of the simulations that are run in serial can replicate results obtained in the literature, and finally to ensure these same results are obtained when the code is run in parallel on multiple compute cores. We first test the case of an int type instance of the class being used for simulations of models that require an integer data type of every site of the grid. We do so by running simulations of the Ising model [3,7,8] where we have $\phi(x) \in \{-1, +1\}$ at every lattice site. We then test the capability of the class to run simulations of scalar field theory [5] which differs by requiring a double data type at every site $x$ of the grid given by $\phi(x) \in \mathbb{R}$. This is because in scalar field theory the "spin" variables do not just take on two integer values but can instead take on a continuum of values. We will now give a brief background on these types of simulations.

5.1 The Ising model

It is well known that electrons have their own intrinsic angular momentum which is commonly referred to as "spin" which acts as an influence over the electrons magnetic dipole moment [8]. These dipoles induces a local magnetic field surrounding that electron which points in a given direction dependant on the orientation of the spin. In a paramagnetic material the magnetic dipoles tend to favour a random orientation and point in anti-parallel directions from neighbouring spins. This causes the effects of local magnetic fields from each dipole to cancel each other out and lead to an overall net magnetization of zero on a macroscopic scale. However in ferromagnetic materials such as cobalt or iron the dipoles prefer to align in parallel with each other.
and an overall non-zero magnetization can be felt on a macroscopic scale. These materials only remain ferromagnetic within a certain temperature regime. At low temperatures the magnetic dipoles tend to align inducing a magnetic effect, but as the temperature keeps increasing until it reaches a special temperature known as the "critical temperature" or "curie point" a phase transition occurs and the material becomes paramagnetic with a random orientation of the magnetic dipoles. The magnetization is then lost. The simplest, most well known model used to study this behaviour is the Ising model. This is a lattice model which places a +1 or -1 on every lattice site representing each possible direction of the interacting magnetic dipoles. In the absence of an external magnetic field the Hamiltonian of this system can be given simply by

\[ H = -J \sum_{<i,j>} s_i s_j - H \sum_i s_i \]  \hspace{1cm} (1)

where \( J \) is a coupling constant between neighbouring spins and \( <i,j> \) denotes the sum over nearest neighbours and \( H \) represents the strength of an external magnetic field. The Hamiltonian represents the energy of the system and is a key tool in running simulations of the Ising model using the Metropolis algorithm [1], which we introduce below.

5.2 Metropolis algorithm

The most important tool of a Monte Carlo simulation is the idea of a partition function [2,9] which contains all of the information about the system under consideration in terms of its energy. The partition function for a system takes the form

\[ Z = \sum_{\forall S} e^{-\mathcal{H}/\kappa T} \]  \hspace{1cm} (2)

where \( S \) represents some state of the system, \( H \) is the Hamiltonian, \( T \) temperature and \( \kappa \) is the Boltzmann constant. For large systems with a large number of degrees of freedom the partition function cannot be evaluated exactly, since it requires us to sum over all possible states of the system. The probability of any particular
configuration of the system denoted $\mu$ occurring can be determined by the partition function by

$$P_\mu = \frac{1}{Z} e^{-\mathcal{H}(\mu) / \kappa T}$$  \hspace{1cm} (3)$$

where $\mathcal{H}(\mu)$ is the Hamiltonian when the system is in the $\mu$th state. The Metropolis algorithm can be derived (see [2]) from this by considering the master equation given by

$$\frac{\partial P_\mu(t)}{\partial t} = - \sum_{\mu \neq \sigma} [P_\mu(t) W_{\mu \rightarrow \sigma} - P_\sigma(t) W_{\sigma \rightarrow \mu}]$$  \hspace{1cm} (4)$$

where $P_\mu(t)$ is the probability of the system being in state $\mu$ at time $t$ and $W_{\mu \rightarrow \sigma}$ is the transition rate for $\mu \rightarrow \sigma$. In equilibrium we have $\frac{\partial P_\mu(t)}{\partial t} = 0$ and resulting equation is known as detailed balance

$$P_\mu(t) W_{\mu \rightarrow \sigma} = P_\sigma(t) W_{\sigma \rightarrow \mu}$$  \hspace{1cm} (5)$$

Since $P_\mu(t)$ is generally not known for large systems the Metropolis algorithm attempts to overcome this by generating a Markov chain of states where each new state is generated directly from the previous one, then the relative probabilities is the ratio of the individual probabilities and one has

$$\frac{P_\mu(t)}{P_\sigma(t)} = \frac{\frac{1}{Z} e^{-\mathcal{H}(\mu) / \kappa T}}{\frac{1}{Z} e^{-\mathcal{H}(\sigma) / \kappa T}} = \frac{e^{-(\mathcal{H}(\mu) - \mathcal{H}(\sigma))/\kappa T}}{e^{-\Delta E / \kappa T}} = \frac{W_{\sigma \rightarrow \mu}}{W_{\mu \rightarrow \sigma}}$$  \hspace{1cm} (6)$$

and so we see we can cancel out the dependence of the partition function and as a result the Metropolis algorithm generates new configurations of the system using a transition probability which depends only on the energy difference between the current configuration $E_0$ and a newly proposed configuration $E_1$, namely $\exp(-\Delta E / \kappa T)$ where $\Delta E = E_1 - E_0$, $T$ is the temperature and $\kappa$ is the Boltzmann constant. Any transition rate which satisfies detailed balance is thus accepted and the Metropolis choice is
\[ W_{\mu \rightarrow \sigma} = \begin{cases} 
e^{-\Delta E / \kappa T} & \Delta E > 0 \\ 1 & \Delta E < 0 \end{cases} \]

A new configuration can be proposed by selecting a spin on the lattice and flipping the spin to a new configuration with this probability. After every sweep of the lattice using the metropolis update the observables of interest are measured and added to the statistical average of the ensemble. So at each step we need to evaluate the Hamiltonian which is a sum over all the nearest neighbours of the given site we are on. When implementing this in parallel using the domain decomposition approach discussed in the previous section, this means that when we want to evaluate the Hamiltonian on a site on the boundary of a processors sub domain, we require information about the nearest neighbour owned by the neighbouring processor and so communication between processes is an essential part of the parallel simulation. In the simulation that I ran, each processes uses the exchange3D function from the class to send its surfaces to the ghost surface of the relevant neighbouring processor and from here each processor is able to take the the data from that ghost to continue to evaluate the Hamiltonian correctly.

5.3 Red/black updates

When working with the Metropolis algorithm in parallel it is necessary to update the sites of the lattice using the so called "red/black" update approach. We will use a 2 dimensional lattice for illustration of this approach but the idea is very easily extended into 3 dimensions. Red/Black updates means we update points on adjacent processors that do not depend on the points currently being updated on adjacent ones in the same time step. Consider the following decomposition of a two dimensional lattice with periodic boundary conditions. Assume we have divided the lattice among two processors with the appropriate ghost columns. We have process 0 on the left and process 1 on the right.
We first begin by updating the red sites only on each processor. When we update the site circled in blue, we require the data from the ghost point marked with an "X". But because each process is only updating red points, then this ensures that this point is not stale meaning that the real, non-ghost point circles in green could not possibly have already been updated by processor 1. This ensures that the correct data is used in the evaluation of the Hamiltonian at every step in the Metropolis update. Once the red points are updated, we call the MPI exchange function, so that each process sends the correct updated values, then we update the blacks points and call the exchange function again. This approach ensures we have no stale points. The original Metropolis update we used involved sending single points each time an edge point was updated. However we incurred significant overhead in this approach due to synchronization and a large amount of messages and so decided to use the red/black approach to decrease the communication cost.
5.4 Ising model simulation results

The simulations of the Ising model were run in serial and in parallel for 2, 3, 5, 8, 12 and 32 processors using the red/black updates as described above across a range of different lattice sizes $L = 10, 20, 40, 60$. The simulation code was divided into two parts. The first part involved running simulations for different values of $\beta = J/\kappa T$ in the interval $(0, 1)$ which were generated randomly from the built in random generator

```cpp
std::uniform_real_real_distribution<double> dist(0.0, 1.0);
```

This is the same random number generator used to generate uniformly distributed random numbers in $(0, 1)$ needed to update the lattice using the metropolis algorithm. We then looked at how the magnetization and energy change as a function of $\beta$ by computing observables such as average absolute magnetization per spin

$$<m> = \frac{1}{L^3} \sum_x |\phi(x)|,$$  \hspace{1cm} (8)

average energy per spin

$$<E> = \frac{1}{2L^3} \langle \mathcal{H}_{i,j,k} \rangle,$$  \hspace{1cm} (9)

and the magnetic susceptibility $\chi$ defined by

$$\chi = \beta(<M^2> - <M>^2).$$  \hspace{1cm} (10)

The second part involves plotting the results of these observables after each metropolis sweep versus the number of Monte Carlo iterations for a fixed value of $\beta = 0.8$. We see that they all approach a constant value meaning that the simulation has now thermalized and has reached equilibrium. The plots will give us an indication of how many iterations were needed to achieve this. We see that for lattice sizes $L = 10, 20, 30$ at least 200 iterations are required for the observables such as magnetization and energy to thermalize and as expected since we choose a high value of $\beta$ the magnetization thermalizes at a value of 1.
Figure 8: Thermalization of the average magnetization for lattice sizes $L = 20, 40, 60, 80$ with $\beta = 0.8 > \beta_c$.

Figure 9: Thermalization of the average energy for lattice sizes $L = 20, 40, 60, 80$ with $\beta = 0.8 > \beta_c$.

We next look at how these observables change by calculating them after equilibrium.
has been reached for a range of different values of $\beta \in (0, 1)$. These plots will give an indication of where the phase transition occurs by a sharp change in the observables behaviour. The results of the average magnetization per spin, energy and magnetic susceptibility can be seen below and all give a critical point of about $\beta_c = 0.22$ which is in good agreement with results obtained in [3]. We see the model is in the paramagnetic phase for $\beta < \beta_c$, and in the ferromagnetic phase for $\beta > \beta_c$.

Figure 10: Average magnetization per spin as a function of inverse temperature obtained from running simulations in parallel on twelve processors. A critical value of $\beta_c \approx 0.22$ was obtained, in line with the serial results.

We also obtained the following plots of both energy and magnetic susceptibility - a quantity that measures how well a material will become magnetized when placed inside an external magnetic field. These results also indicate a phase transition around the value 0.22.
Figure 11: Energy as a function of inverse temperature obtained from running simulations in parallel on twelve processors. A critical value of $\beta_c \approx 0.22$ was obtained, in line with the serial results.

Figure 12: Magnetic Susceptibility as a function of inverse temperature obtained from running simulations in parallel on twelve processors. A critical value of $\beta_c \approx 0.22$ was obtained, in line with the serial results. Note: We were not concerned with scaling or units in this plot, but more so just the critical behaviour.
The Ising model simulation was a simple test of the class for an integer data type since the matter fields were only restricted to two integer values of +1 or −1. We now extend our test of the class to account for a double data type placed at every lattice site by looking at $\phi^4$ quantum field theory on the lattice where the "spins" can now take on a continuous set of values $\phi(x) \in \mathbb{R}$.

5.5 Scalar quantum field theory on the lattice

Quantum field theory describes the interactions of matter at the sub-nuclear scale. The simplest type of a field theory is that of a free scalar field, whose Lagrangian is given by

$$\mathcal{L} = \frac{1}{2} \left( \frac{\partial \phi}{\partial t} \right)^2 - \frac{1}{2} (\nabla \phi)^2 - \frac{1}{2} m^2 \phi^2$$ (11)

The particles defined by a free scalar field theory do not interact. In order to describe particle interactions we add an additional $\frac{g}{4} \phi^4$ term to the Lagrangian. This then becomes the Lagrangian of the $\phi^4$ scalar field theory, and many interesting phenomena of scalar fields can be studied through simulations of this theory. The starting point of these simulations is the continuum Euclidean action

$$S = \int dt d^2 x \left[ \frac{1}{2} (\partial_\mu \phi)^2 + \frac{1}{2} m^2 \phi^2 + \frac{g}{4!} \phi^4 \right]$$ (12)

with scalar field component $\phi$, bare mass $m$ and the bare strength of the self coupling $g$. To discretize the continuum theory we must replace the continuous field with one defined on a discrete lattice. Any integrals then become a sum over lattice sites

$$\int dt d^3 x \rightarrow a^3 \sum_x$$ (13)

while any derivatives are replaced with finite difference approximations in terms of nearest neighbour sites.
\[ \partial_\mu \phi_0 \rightarrow \frac{\phi_0(x + \hat{\mu}) - \phi_0(x)}{a} \]  \hspace{1cm} (14)

where \( a \) is the lattice spacing and \( \hat{\mu} \) is a unit vector in the \( \mu \) direction. The discretized lattice action is then

\[ S = \sum_x \left[ -2\kappa \sum_\mu \phi(x)\phi(x + \hat{\mu}) + \phi(x)^2 + \lambda(\phi(x)^2 - 1)^2 - \lambda \right] \]  \hspace{1cm} (15)

Note that the original formulation had two bare parameters \( m \) and \( g \) and these have been mapped according to the mapping presented in [5] to the two parameters \( \kappa \) and \( \lambda \). To initialize the lattice we define \( \Phi(x) = \sqrt{2\kappa}\phi(x) \) on every site \( x \), where \( \phi(x) \in \{+1, -1\} \) as in the Ising model simulation. These values represent the amplitude of the field throughout the lattice and the additional factor of \( \sqrt{2\kappa} \) means that the value at every lattice site can no longer be an integer and would have to be represented in double precision. The update of the value at every site is no longer an integer spin flip and so the metropolis algorithm would have to be altered accordingly to account for this. It does so by looping through every site \( x \) of the lattice for any given \( x \) update \( \phi(x) \rightarrow \phi'(x) = \phi(x) + \Delta \epsilon \), where \( \Delta > 0 \) is some tunable parameter of the simulation and \( \epsilon \) is a random number uniformly distributed in the interval \([-1, +1]\). We then calculate a change in the action in the system \( \Delta S = S[\phi'] - S[\phi] \), and the update \( \phi_x \rightarrow \phi' \) is accepted with according to the probability \( \min(1, e^{-\Delta S}) \).

This process is repeated for every site and so since there is a dependence of data on nearest neighbours serves as a good example for testing the parallel nature of the class. On the next page we will discuss the details of the simulation that we used and the results we obtained.

### 5.6 \( \phi^4 \) simulation details and results

In the Ising model simulation we had one free parameter we varied throughout the simulation. Here, we have two namely the \( \kappa \) and \( \lambda \). Simulations were run by holding \( \lambda \) fixed and looking at fluctuations in \( <\phi(x)> \) for random values of \( \kappa \) in the interval
As in [5] we are interested in the two limits $\lambda = 0$ and $\lambda = \infty$. We computed $\langle \phi(x) \rangle$ using

$$
\langle \phi(x) \rangle = \frac{1}{L^3} \sum_x \phi(x)
$$

(16)

And examined its fluctuations about 0 for a range of values of $\lambda$. We demonstrate correctness of our C++ class by observing the same behaviour as in [5], namely the following

- When $\lambda = 0$, the interaction term is zero and we have a free theory. There is both an ordered and disordered phase to this theory which can be seen across different values of $\kappa$ and for which there exists a phase transition for some $\kappa_c$. In the same way as in [5] we note the disordered phase for small $\kappa$ where $\langle \phi(x) \rangle = 0$, and an ordered phase $\langle \phi(x) \rangle \neq 0$ when $\kappa$ is large. The symmetry is said to be broken when $\kappa > \kappa_c$ corresponding to the vanishing of the mass. The critical point at which this occurs seems to be about $\kappa_c \approx 0.22$, the same critical point as the one we obtained in the 3 Dimensional Ising model. In the Ising limit however as we increase $\lambda > 0$, it appears as though $\langle \phi(x) \rangle$ remains at zero for $\kappa \in (0, 1)$. 


The magnetic susceptibility for $\phi^4$ theory can also be defined analogously to that of the Ising model, namely

$$\chi = \frac{1}{L^3}(<\phi^2> - <\phi>^2).$$  \hspace{1cm} (17)

The results we obtained at $\lambda = 0$ are also indicative of a phase transition at $\kappa_c = 0.22$ the same as the 3D Ising model. This can be seen for different lattice sizes in the plot below.
Figure 14: Magnetic Susceptibility as a function of $\kappa$ for lattice sizes $L = 20, 40, 60, 80$

The importance of $\phi^4$ type simulations have been noted in [5] and mainly serve as a tool used to further understand the Higgs particle. This is a particle from the standard model of particle interactions which gives a mass to all other particles by coupling to them. The field theory describing the Higgs particle is very much like that of $\phi^4$ except the matter fields now take the form of a complex doublet whose bare parameters are chosen so that the system is in its broken symmetry phase. We will not go further into the details of these types of simulations in this thesis but we do note the significance of the results we obtained in our simulation. Observing similar behaviour to that obtained in [5] and observing similar critical behaviour to that obtained in our Ising model simulation is a solid means of ensuring the template class, especially the parallel exchange of data, works efficiently for different data types corresponding to different matter fields. Now it only remains to test the performance of the class for different data types and lattice sizes.
6 Performance tests

In order to test how both the striped decomposition of the 3D lattice and the cubic decomposition perform in comparison to each other, we look at strong scaling results of both our Ising Model and scalar field theory simulations across a range of problem sizes and a different number of processors. We look at strong scaling by testing how the overall compute time of the job scales as we increase the number of processors and keep the problem size fixed. In an ideal situation we would like the speedup to continue to grow in proportion to the number of processing elements, but this is almost never the case due to increased overhead associated with the parallel execution of the code eg. the additional communication overhead. The strong scaling results would thus give an indication of how much processing elements to request for different problem sizes for both types of the MPI decomposition. This will allow us to further understand when one decomposition becomes more favourable over the other when running the parallel Monte Carlo simulations. In order to test this we ran the performance tests on the Lonsdale cluster at the Trinity Centre for High Performance Computing which has the following specifications

<table>
<thead>
<tr>
<th>Lonsdale Cluster, TCHPC</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>154</td>
</tr>
<tr>
<td>Processors</td>
<td>1232</td>
</tr>
<tr>
<td>Processors Per Node</td>
<td>8</td>
</tr>
<tr>
<td>RAM</td>
<td>128 ×16GB, 24 × 32GB, 2 × 64GB</td>
</tr>
<tr>
<td>Architecture</td>
<td>64 bit</td>
</tr>
<tr>
<td>OS</td>
<td>Scientific Linux 6 ×</td>
</tr>
<tr>
<td>Interconnect</td>
<td>DDR Infiniband</td>
</tr>
<tr>
<td>Performance (Peak)</td>
<td>11.33 TF</td>
</tr>
<tr>
<td>Performance (Linpack)</td>
<td>8.9 TF</td>
</tr>
</tbody>
</table>

Because the Lonsdale cluster has 8 processors per node we have ran simulations on 1, 2, 3, 5, 8, 12, 16 and 32 cores so as to get an idea of the effects that network communication has on the performance of the simulations. This is because when we
request more than 8 nodes for a job then we will have to run the simulations across multiple nodes and this will incur additional communication across the network, not just local communication between the cores of a node.

6.1 Striped decomposition - strong scaling for int type matter fields

<table>
<thead>
<tr>
<th>Num Procs</th>
<th>$10^3$ lattice</th>
<th>$20^3$ lattice</th>
<th>$40^3$ lattice</th>
<th>$60^3$ lattice</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>60.17</td>
<td>475.3</td>
<td>3788.7</td>
<td>13055.2</td>
</tr>
<tr>
<td>2</td>
<td>38.88</td>
<td>245.63</td>
<td>1944.02</td>
<td>6620.43</td>
</tr>
<tr>
<td>3</td>
<td>28.26</td>
<td>173.99</td>
<td>1365.25</td>
<td>4381.07</td>
</tr>
<tr>
<td>5</td>
<td>18.69</td>
<td>104.64</td>
<td>794.95</td>
<td>2642.31</td>
</tr>
<tr>
<td>8</td>
<td>16.43</td>
<td>77.97</td>
<td>506.36</td>
<td>1781.89</td>
</tr>
<tr>
<td>12</td>
<td>11.76</td>
<td>55.05</td>
<td>400.13</td>
<td>1129.19</td>
</tr>
<tr>
<td>16</td>
<td>24.46</td>
<td>54.31</td>
<td>308.77</td>
<td>921.93</td>
</tr>
<tr>
<td>32</td>
<td>89.58</td>
<td>76.05</td>
<td>213.64</td>
<td>473.01</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Num Procs</th>
<th>$10^3$ lattice</th>
<th>$20^3$ lattice</th>
<th>$40^3$ lattice</th>
<th>$60^3$ lattice</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1.547</td>
<td>1.935</td>
<td>1.948</td>
<td>1.97</td>
</tr>
<tr>
<td>3</td>
<td>2.1291</td>
<td>2.73</td>
<td>2.775</td>
<td>2.979</td>
</tr>
<tr>
<td>5</td>
<td>3.219</td>
<td>4.370</td>
<td>4.765</td>
<td>4.94</td>
</tr>
<tr>
<td>8</td>
<td>3.662</td>
<td>5.865</td>
<td>7.482</td>
<td>7.3266</td>
</tr>
<tr>
<td>12</td>
<td>5.125</td>
<td>8.6339</td>
<td>9.468</td>
<td>11.5615</td>
</tr>
<tr>
<td>16</td>
<td>2.459</td>
<td>8.751</td>
<td>12.27</td>
<td>14.16</td>
</tr>
<tr>
<td>32</td>
<td>0.671</td>
<td>6.2498</td>
<td>17.73</td>
<td>27.002</td>
</tr>
</tbody>
</table>
### 6.2 Cubic decomposition - strong scaling for int type matter fields

<table>
<thead>
<tr>
<th>Num Pros</th>
<th>$10^3$ lattice</th>
<th>$20^3$ lattice</th>
<th>$40^3$ lattice</th>
<th>$60^3$ lattice</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>63.41</td>
<td>461.15</td>
<td>3561.18</td>
<td>12068.5</td>
</tr>
<tr>
<td>2</td>
<td>42.32</td>
<td>245.36</td>
<td>1846.58</td>
<td>6245.32</td>
</tr>
<tr>
<td>3</td>
<td>31.26</td>
<td>171.13</td>
<td>1308.27</td>
<td>4142.21</td>
</tr>
<tr>
<td>5</td>
<td>20.94</td>
<td>103.1</td>
<td>736.08</td>
<td>2446.52</td>
</tr>
<tr>
<td>8</td>
<td>15.93</td>
<td>71.71</td>
<td>480.73</td>
<td>1578.05</td>
</tr>
<tr>
<td>12</td>
<td>91.12</td>
<td>61.08</td>
<td>338.45</td>
<td>1064.81</td>
</tr>
<tr>
<td>16</td>
<td>66.67</td>
<td>50.49</td>
<td>251.38</td>
<td>810.02</td>
</tr>
<tr>
<td>32</td>
<td>1305.02</td>
<td>212.7</td>
<td>183.11</td>
<td>426.39</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Num Pros</th>
<th>$10^3$ lattice</th>
<th>$20^3$ lattice</th>
<th>$40^3$ lattice</th>
<th>$60^3$ lattice</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1.498</td>
<td>1.87</td>
<td>1.928</td>
<td>1.932</td>
</tr>
<tr>
<td>3</td>
<td>2.028</td>
<td>2.694</td>
<td>2.722</td>
<td>2.913</td>
</tr>
<tr>
<td>5</td>
<td>3.028</td>
<td>4.472</td>
<td>4.8380</td>
<td>4.932</td>
</tr>
<tr>
<td>8</td>
<td>3.9805</td>
<td>6.430</td>
<td>7.4078</td>
<td>7.64</td>
</tr>
<tr>
<td>12</td>
<td>0.695</td>
<td>7.594</td>
<td>10.52</td>
<td>11.3339</td>
</tr>
<tr>
<td>16</td>
<td>0.951</td>
<td>9.13</td>
<td>14.166</td>
<td>14.8990</td>
</tr>
<tr>
<td>32</td>
<td>0.048</td>
<td>2.168</td>
<td>19.448</td>
<td>28.303</td>
</tr>
</tbody>
</table>
Figure 15: Striped decomposition strong scaling for int data type

Figure 16: Cubic decomposition strong scaling for int data type
From these strong scaling plots we are able to gain an insight into the performance properties of the parallel simulation when run using both the striped and cubic decompositions. In general, for the case of an integer data type the performance scales very well for large lattice sizes using both decompositions. For smaller lattice sizes however, the performance does not scale well once we exceed 8 cores. This is most likely due to the fact that each Lonsdale node only has 8 cores and so by requesting more means the work will be split across nodes and so network communication will be a bottleneck of the implementation. This is especially true for smaller problem sizes, but seems to be overshadowed in simulations involving larger lattices. The class seems to perform most poorly when using the cubic decomposition on small lattices, but the cubic decomposition then begins to perform better than the striped decomposition as the lattice size is increased. This observation is more likely due to the fact that for smaller lattices we have a large amount of data to exchange in all directions in comparison to the amount of computation that we do on a small sub-grid, whereas for larger lattices the cost of communication becomes worth splitting up the large amount of work needed to work on a large lattice.

6.3 Striped decomposition - strong scaling for double type matter fields

<table>
<thead>
<tr>
<th>Num Procs</th>
<th>10^3 lattice</th>
<th>20^3 lattice</th>
<th>40^3 lattice</th>
<th>60^3 lattice</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>83.91</td>
<td>712.2</td>
<td>5701.85</td>
<td>20958.1</td>
</tr>
<tr>
<td>2</td>
<td>51.8</td>
<td>367.17</td>
<td>2933.64</td>
<td>9925.74</td>
</tr>
<tr>
<td>3</td>
<td>39.73</td>
<td>252.21</td>
<td>1991.93</td>
<td>6536.19</td>
</tr>
<tr>
<td>5</td>
<td>23.83</td>
<td>152.29</td>
<td>1191.6</td>
<td>4170.22</td>
</tr>
<tr>
<td>8</td>
<td>22.45</td>
<td>115.04</td>
<td>745.28</td>
<td>2662.05</td>
</tr>
<tr>
<td>12</td>
<td>15.63</td>
<td>81.88</td>
<td>598.85</td>
<td>1770.25</td>
</tr>
<tr>
<td>16</td>
<td>21.28</td>
<td>79.77</td>
<td>464.55</td>
<td>1386.65</td>
</tr>
<tr>
<td>32</td>
<td>94.41</td>
<td>53.09</td>
<td>320.77</td>
<td>720.87</td>
</tr>
<tr>
<td>Num Procs</td>
<td>$10^3$ lattice</td>
<td>$20^3$ lattice</td>
<td>$40^3$ lattice</td>
<td>$60^3$ lattice</td>
</tr>
<tr>
<td>-----------</td>
<td>----------------</td>
<td>----------------</td>
<td>----------------</td>
<td>----------------</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1.6</td>
<td>1.93</td>
<td>1.943</td>
<td>2.11</td>
</tr>
<tr>
<td>3</td>
<td>2.11</td>
<td>2.82</td>
<td>2.8639</td>
<td>3.360</td>
</tr>
<tr>
<td>5</td>
<td>3.52</td>
<td>4.67</td>
<td>4.785</td>
<td>5.02</td>
</tr>
<tr>
<td>8</td>
<td>3.73</td>
<td>6.19</td>
<td>7.65</td>
<td>7.872</td>
</tr>
<tr>
<td>12</td>
<td>5.36</td>
<td>8.69</td>
<td>9.521</td>
<td>11.839</td>
</tr>
<tr>
<td>16</td>
<td>3.94</td>
<td>8.92</td>
<td>12.273</td>
<td>15.114</td>
</tr>
<tr>
<td>32</td>
<td>0.88</td>
<td>13.414</td>
<td>17.775</td>
<td>29.073</td>
</tr>
</tbody>
</table>

### 6.4 Cubic decomposition - strong scaling for double type matter fields

<table>
<thead>
<tr>
<th>Num Procs</th>
<th>$10^3$ lattice</th>
<th>$20^3$ lattice</th>
<th>$40^3$ lattice</th>
<th>$60^3$ lattice</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>104.47</td>
<td>789.93</td>
<td>6072.63</td>
<td>22458.4</td>
</tr>
<tr>
<td>2</td>
<td>64</td>
<td>418.42</td>
<td>3258.9</td>
<td>10778.4</td>
</tr>
<tr>
<td>3</td>
<td>51.15</td>
<td>298.32</td>
<td>2172.08</td>
<td>7208.7</td>
</tr>
<tr>
<td>5</td>
<td>30.01</td>
<td>173.29</td>
<td>1303.1</td>
<td>4345.25</td>
</tr>
<tr>
<td>8</td>
<td>24.28</td>
<td>116.85</td>
<td>823.2</td>
<td>2751.28</td>
</tr>
<tr>
<td>12</td>
<td>90.16</td>
<td>93.08</td>
<td>564.24</td>
<td>1823.77</td>
</tr>
<tr>
<td>16</td>
<td>69.71</td>
<td>74.13</td>
<td>421.59</td>
<td>1381.11</td>
</tr>
<tr>
<td>32</td>
<td>265.47</td>
<td>238.06</td>
<td>263.97</td>
<td>715.3</td>
</tr>
<tr>
<td>Num Procs</td>
<td>$10^3$ lattice</td>
<td>$20^3$ lattice</td>
<td>$40^3$ lattice</td>
<td>$60^3$ lattice</td>
</tr>
<tr>
<td>-----------</td>
<td>----------------</td>
<td>----------------</td>
<td>----------------</td>
<td>----------------</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1.63</td>
<td>1.8878</td>
<td>1.86</td>
<td>2.083</td>
</tr>
<tr>
<td>3</td>
<td>2.042</td>
<td>2.647</td>
<td>2.79</td>
<td>3.115</td>
</tr>
<tr>
<td>5</td>
<td>3.48</td>
<td>4.558</td>
<td>4.66</td>
<td>5.168</td>
</tr>
<tr>
<td>8</td>
<td>4.302</td>
<td>4.192</td>
<td>7.37</td>
<td>8.162</td>
</tr>
<tr>
<td>12</td>
<td>1.15</td>
<td>8.48</td>
<td>10.66</td>
<td>12.314</td>
</tr>
<tr>
<td>16</td>
<td>1.498</td>
<td>10.65</td>
<td>14.404</td>
<td>16.261</td>
</tr>
<tr>
<td>32</td>
<td>0.393</td>
<td>3.31</td>
<td>23.005</td>
<td>31.3971</td>
</tr>
</tbody>
</table>

Figure 17: Speedup plot for striped decomposition with a double data type
For the case of a double data type, similar observations were made to that of an int type. Namely, both decomposition scale better for larger lattice sizes, with the cubic decomposition performing most poorly for smaller lattices and performing more optimally than the striped decomposition for larger lattices. To conclude our analysis we now compare the speedup obtained between ints and doubles together for the smallest and largest lattice sizes we used. From the figures in the next page we see that the speedup of simulations with both int and double data types take the same shape as we increase the number of cores except the speedup for double precision seems to be shifted upward slightly to that of an int regardless of the problem size.
Figure 19: $10 \times 10 \times 10$ lattice comparison of int and double speedup

Figure 20: $60 \times 60 \times 60$ lattice comparison of int and double speedup
7 Conclusions and future additions to the Class

In this thesis we investigated the scaling behaviour of a template C++ class designed to run simulations of field theories on a three dimensional lattice. Two different MPI decompositions were considered and their performance was tested on both the Ising model and scalar field theory simulations.

The main conclusions that we drew from our results were that the template system performs well in parallel for large problem sizes for both decompositions that we tested. These observations remained much the same regardless of the data type placed at every lattice site, with a slight favour for the double precision data types. The striped decomposition performs better for smaller problem sizes due to overheads in communication and pack/unpack methods in the cubic decomposition. However as we increase the lattice size the finer level of parallelism in the cubic decomposition overshadows additional communication costs leading it to become an optimal choice for larger lattice sizes. Finally we note that no major issues were present in developing a template system for these types of simulations and we conclude by outlining future plans and ideas on how the class can be improved.

First, we hope to extend the class to higher dimensions. We then hope to add functionality to the class to allow support for non-local interactions. At present the class can only work with running simulations where the updates depend on nearest neighbour interactions and only one ghost surface is needed on each subgrid. For future work the class could be made more generic by allowing for more general interactions where more than just nearest neighbours are needed in the MPI exchange. In addition, we hope to extend the functionality of the class to be able to run simulations of gauge theories, such as quantum chromodynamics, where in addition to the usual matter fields on the lattice we also have gauge fields. In any case it would be interesting to see the affects of adding more generality to the class would have on its performance and reliability.
References