Parallelising the Molecular Dynamics Simulation of Dipolar Fluids

Author: Peter Duncan
Supervisor: Dr Hans-Wolfgang Loidl

A thesis submitted in fulfilment of the requirements for the degree of MSc IT (Software Systems) in the School of Mathematical and Computer Sciences

August 2015
Declaration of Authorship

I, Peter DUNCAN, declare that this thesis titled, 'Parallelising the Molecular Dynamics Simulation of Dipolar Fluids' and the work presented in it is my own. I confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (e.g., ideas, equations, figures, text, tables, programs) are properly acknowledged at any point of their use. A list of the references employed is included.

Signed: ____________________________________________

Date: ______________________________________________
Abstract

A Molecular Dynamics simulation of a dipolar fluid has been parallelised on a distributed memory computer. The parallelisation was implemented in MPI based on existing code written in C by the author. Two parallelisation algorithms appropriate to the simulation of systems with long-ranged forces were compared: the Replicated Data algorithm and the Systolic Loop algorithm.

The parallelisation of Molecular Dynamics usually concentrates on the calculation of forces, which is the most computationally expensive part of the simulation. In the Replicated Data method, the data for every particle is sent to each processor and each processor calculates the forces for a subset of the particles. This has a high communication cost. The Systolic Loop method was designed as a method that reduces the cost of communication. We have implemented both methods in order to compare their scaling properties.

The Replicated Data method requires proper load balancing. To this end, we have derived an equation giving the amount of work for a particular range iterations. This equation is used in a function called at the beginning of the program to distribute the load.

For a 1024-particle system, the Replicated Data method scale well with number of processors until around 10, when the speedup started to level off at a value of around 5. Better speedups could be obtained for a larger system of 10000 particles, yielding a maximum speedup of around 14.

It was found for the 1024-particle system, the Replicated Data algorithm performed slightly better than the Systolic Loop algorithm. For a 10000-particle system, the Systolic Loop displayed better scaling properties than the Replicated Data with high numbers of processors.

The conclusion is that for smaller system sizes, the Replicated Data algorithm may be a better choice. For systems with 10000 or more particles, the Systolic Loop algorithm is best if we use a large number of processors.
I would like to thank my supervisor, Dr Hans-Wolfgang Loidl, for his help and guidance in the running of this project.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration of Authorship</td>
<td>i</td>
</tr>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>iii</td>
</tr>
<tr>
<td>Contents</td>
<td>iv</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Objectives</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Methodology</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Requirements Analysis</td>
<td>3</td>
</tr>
<tr>
<td>1.4 Professional, Legal, Ethical and Social Issues</td>
<td>4</td>
</tr>
<tr>
<td>1.5 Project plan</td>
<td>5</td>
</tr>
<tr>
<td>2 Background and Literature Review</td>
<td>6</td>
</tr>
<tr>
<td>2.1 Introduction to parallel programming</td>
<td>6</td>
</tr>
<tr>
<td>2.2 Programming models</td>
<td>10</td>
</tr>
<tr>
<td>2.2.1 MPI</td>
<td>10</td>
</tr>
<tr>
<td>2.2.2 OpenMP</td>
<td>10</td>
</tr>
<tr>
<td>2.3 Molecular dynamics</td>
<td>11</td>
</tr>
<tr>
<td>2.4 Ferrofluids</td>
<td>13</td>
</tr>
<tr>
<td>2.5 Parallelising Molecular Dynamics simulations</td>
<td>14</td>
</tr>
<tr>
<td>3 Implementation and Evaluation</td>
<td>22</td>
</tr>
<tr>
<td>3.1 Replicated Data algorithm</td>
<td>22</td>
</tr>
<tr>
<td>3.2 Systolic Loop algorithm</td>
<td>29</td>
</tr>
<tr>
<td>4 Conclusions</td>
<td>37</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The aim of this project was to improve the performance of a research-strength Molecular Dynamics implementation by parallelising it on a cluster of (multicore) workstations. The sequential Molecular Dynamics code was written by the author as part of physics research on the thermodynamic properties of ferrofluids. Cluster computing is the most widely used platform for parallelism, particularly in scientific research, so the results of this study are highly relevant to its potential applications.

Molecular Dynamics simulations are used by Physicists to investigate bulk properties and phase behaviour of condensed matter systems. These simulations are highly computationally intense, with runs taking many hours or even days, and typically a whole series of runs are performed to investigate the system properties under different conditions of temperature or pressure etc. It is therefore very useful to consider ways of making these simulations faster, which is why this is a major application for parallel programming.

The current project is based on Molecular Dynamics code previously written by the author in Fortran. The author has re-written the code in C and used the MPI extension to C to investigate different ways to parallelise the code. The particular system simulated is a dipolar fluid. This consists of spherical ferromagnetic particles and forms a model of a magnetic fluid called a “ferrofluid”.

The aim of parallelisation is to reduce the runtime of the program. The main metric of success in this is the speedup - the factor by which the parallel runtime is shorter than the sequential runtime. We wish this to be as large as possible for a given number of processors. It is also desirable that speedup scales well with number of processors - that is, as we use more processors, we would like to obtain a proportionate improvement in speedup.
1.1 Objectives

The research to be carried out will involve parallelising existing code for performing Molecular Dynamics simulations of a system of dipolar soft spheres.

The objectives are:

- To optimise the performance of the sequential code
- To parallelise the code using C++/MPI
- To tune the performance of the parallel code by load balancing etc.
- To compare the performance to that of the original sequential code

The ultimate aim is to produce a significant speedup relative to the sequential code, which would allow either longer runs to be performed, or allow larger system sizes to be simulated.

1.2 Methodology

The research will be carried out in the following stages:

1. Sequential code for the Molecular Dynamics simulation of dipolar soft spheres has previously been written in Fortran by the author. It will be rewritten in C++ in order to use a more up-to-date object oriented language.

2. Performance tuning of the sequential code will be carried out, with the aim of reducing the runtime. Profiling tools such as gprof will be used to establish where the bottlenecks are in the program. Tools such as cachegrind could identify cache misses where the program needs to fetch data from main memory.

3. The program will be parallelised to run on Heriot Watt University’s beowulf cluster, which has 32 nodes, each with 8 cores. The technology to be used will be MPI to perform message passing between processors. A hybrid MPI/OpenMP scheme could be implemented if there is time.

4. Performance tuning of the parallel implementation. This will involve issues such as ensuring good load balancing and looking for ways to reduce communication between processors.
5. Measurement of parallel performance. The primary metric of performance will be the speedup relative to the sequential code, given as the ratio $t_{seq}/t_{par}$, where $t_{seq}$ is the runtime for the sequential code and $t_{par}$ is the runtime of the parallel code. The aim is to obtain good scaling as the number of processors is increased. We wish to come as close as possible to achieving a speedup equal to the number of processors in use. We will also investigate the scaling with the number of particles in the simulation.

1.3 Requirements Analysis

The first requirement is to write a sequential program to carry out a Molecular Dynamics simulation of a quasi-two-dimensional dipolar fluid. The author has written such a code in Fortran. This will be rewritten in C for the purposes of this research. The main requirement of this code is that it is a correct implementation of the Molecular Dynamics algorithm for the dipolar fluid system of interest. The original Fortran code was thoroughly tested by the author. The C version can be tested by ensuring by comparing the results with those from the Fortran version.

Starting from this sequential version, one or more parallel versions of the program will be produced. These will be written in C+MPI. The requirements are:

- The program must still be a correct Molecular Dynamics simulation. It is not expected that the particles will necessarily follow exactly the same trajectories because of the finite precision of the floating point numbers, but it must give the same results for the physical quantities such as energy.

- The program must be a parallel implementation of Molecular Dynamics, that is, it must be capable of running on more than one processor with different processors carrying out different parts of the computation.

- The program should correctly implement the algorithm for parallelisation (in the case of this study the Replicated Data and Systolic Loop algorithms).

While not a strict requirement, the following is highly desirable:

- The program should give a useful speedup in comparison to the sequential code, i.e. when run on larger numbers of processors, it should have a significantly shorter run-time.

- The program should be tuned to optimise its performance, e.g. by load-balancing,
• The program should scale well with number of processors. There should a linear increase in the speedup with number of processors over as large a range as possible.

1.4 Professional, Legal, Ethical and Social Issues

• The British Computing Society’s Code of Conduct and Code of Good Practice will be adhered to. The following are some of the most relevant points.

  – “Strive to achieve well-engineered products that demonstrate fitness for purpose etc.”. I have made every effort to test the code I have produced during development to ensure it produced correct results and implemented the intended parallel algorithms correctly.

  – “Strive to produce well-structured code that facilitates testing and maintenance”. I have tried as far as possible to write clear and maintainable code, adding comments where it may otherwise be obscure. The code functions as it should do, but there may be opportunities to refactor to make it more readable.

  – “Maintain your knowledge of your specialism at the highest level by, for example, reading relevant literature, ... meeting and maintaining contact with other leading practitioners”. In a project such as this where one is learning new skills to put into practice, it is particularly important to take steps to learn about the field. I have made an effort to do background reading to understand the issues. I have met regularly with my supervisor to obtain his advice, and also discussed my work with my colleagues on the course.

• Any code produced will be placed under the GNU Public License, Version 3.

• It is not intended to produce code to be used by others, but to research methods of parallelisation for a particular application. Social and ethical issues are therefore not of high concern in this project. However, it is to be noted that the Beowulf cluster is a shared resource and there is no queueing system for jobs. It is therefore important to make courteous use of the machine by not continually hogging the resource.
### 1.5 Project plan

<table>
<thead>
<tr>
<th>Time Frame</th>
<th>Task Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>January - April</td>
<td>Read literature on parallelisation of Molecular Dynamics</td>
</tr>
<tr>
<td></td>
<td>Learn MPI</td>
</tr>
<tr>
<td>4th May - 17th May</td>
<td>Improve efficiency of sequential code</td>
</tr>
<tr>
<td>18th May - 14th June</td>
<td>Implement the Replicated Data method</td>
</tr>
<tr>
<td>15th June - 30th June</td>
<td>Tune the performance of the parallel implementation</td>
</tr>
<tr>
<td>1st July - 30th July</td>
<td>Implement the Systolic Loop algorithm</td>
</tr>
<tr>
<td>13th July - 1st August</td>
<td>Collect data on speedups obtained</td>
</tr>
<tr>
<td>1st August - 16th August</td>
<td>Writing up</td>
</tr>
</tbody>
</table>

Overall, I worked steadily on the project to ensure it kept moving forward. As a result, I was able to achieve all the goals that were set out at the beginning of the project. The research was completed on schedule to allow time for writing up. However, I underestimated the complexity of producing the Systolic Loop version, and this took slightly longer than expected. I would also have benefited from sticking more rigidly to the time plan in the early stages of the project. This would have left me some time at the end to make a start on some of the extension work, such as creating a hybrid MPI/OpenMP version of the program.
Chapter 2

Background and Literature Review

2.1 Introduction to parallel programming

Usually the process of writing a parallel program starts by considering a sequential implementation of the program. We then look for potential instances of concurrency in the code, i.e. we look for parts of the program that can be performed in any order. These parts could potentially be run at the same time on different processors.

One very commonly exploited source of parallelism occurs in for loops, where we iterate over values of a loop index, say \( i \). Provided that the iterations of the loop are independent, in the sense that one iteration of the loop does not depend on the outcome of another iteration, it will be possible to carry out different iterations of the loop in parallel on different processors.

In some cases there is technically a dependence between loop iterations, but this dependence can be removed in some way. Consider, for example, this loop:

```c
for(i=0; i<N; i++)
{
    sum += f(i);
}
```

Each iteration will require the value of \( sum \) obtained by the previous iteration. However, we can remove this dependence by having each processor accumulate its own total for \( sum \) for the iterations it has carried out. At the end, we then add together the partial
sum from each processor to obtain the overall total. This an example of a reduction operation.

When discussing the types of architecture that can be used to enable parallel programming, reference is often made to Flynn’s taxonomy. This characterises parallel architectures according to their streams of instructions and streams of data. The first category is Single Instruction Single Data (SISD), which corresponds to an ordinary serial computer which can only execute instructions one-by-one. The next category is Single Instruction Multiple Data (SIMD), in which a single data stream is run on multiple processing units which each have their own data stream. Vector processors work in this way. On the level of the machine code, these processors can apply the same set of instructions to different blocks of data in parallel. Modern Graphics Processing Units (GPUs) also use the SIMD model. The Multiple Instruction Single Data (MISD) model is rarely used. The final category is Multiple Instruction Multiple Data (MIMD) in which each processing unit has its own instruction stream and in addition act on its own set of data. MIMD is the model on which most parallel architectures have been designed. It can be further broken down into two subcategories; shared memory architectures and distributed memory architectures.

Distributed memory systems include clusters, which are a collection of computers networked together to allow communication between them. This is one of the cheapest ways for an organisation to access parallel computing. In a distributed memory system, each node of the network will have its own memory space. If one node needs access to data stored on the network, it must be communicated over the network. This can be a very slow process compared to local memory accesses, and therefore it is important to design a system in which the communication is as fast as possible. The advantage of a distributed memory system is that it is highly scalable - it is relatively easy to add new nodes to the network.

In a shared memory system, there are multiple processing units, each of which has access to the same memory space. An example of this are multicore processors which are becoming ubiquitous in modern desktop computers. The processor has multiple cores which can each execute instructions independently. Each core has access to the entire memory of the computer. Shared memory systems can often be easier to program for because shared variables can be accessed easily, unlike in the distributed memory case where explicit communication is needed to share data between the processing units. The disadvantage of the shared memory model is the lack of scalability. Typical desktop computers contain four to eight cores. To have more cores requires advances in hardware technology.
When we look for concurrency in creating a parallel program we look for tasks that can be carried out independently. There are some cases in which the tasks are completely independent, so-called "embarrassingly parallel" programs, and the tasks can continue without any reference to the other tasks that are on-going. In most cases, however tasks will require access to data produced by other tasks. This means that messages will have to be passed between tasks to communicate this data. This communication can be an expensive operation. On a cluster, data transfer between nodes can be some 1000 times slower than local memory accesses. Excessive communication between tasks in a parallel program can therefore severely reduce the program’s efficiency. It is therefore important to design our algorithm to minimise communication as much as possible.

The aim is to reduce the communication to computation ratio.

It is important to ensure that each processor has roughly the same amount to perform. If the load is unevenly distributed, a situation can arise where some processors have finished their work long before others, and must wait for these processors finish before they can continue with the next piece of work. Having some processors idle while a small number complete their work can severely reduce the efficiency of the program. If this is in a loop, the situation is exacerbated. Load balancing refers to re-distributing the work that each processor must perform so that each will take roughly equal time to complete their task. In order to achieve load balancing, consideration must be given to the scheduling of work to the different threads. The scheduling can either be determined beforehand ("static scheduling"), or it can be determined dynamically while the program is running. As an example of static scheduling, consider the very common situation where a loop is parallelised by assigning different iterations of the loop to different threads. If it is known beforehand that each iteration should take roughly equal time, then an equal number of iterations can be assigned to each thread. It may be the case that each iteration could take different amount of time, but this is predictable beforehand, for example, if the loop contains an inner loop with a known number of iterations. The iterations of the outer loop can be scheduled accordingly. In cases where the load on each processor cannot be predicted beforehand, it is not possible to create a static schedule, and a dynamic schedule is required. There are various ways of doing this. In a program structured as a Master-Worker program, one way of achieving load balancing is for the Master processor to manage a pool of small chunks of work. When a processor has finished its current work, it can take its next piece of work from the pool. An alternative method is "work stealing", where a processor which has finished its work can take some work from an active processor. Static scheduling is to be preferred over dynamic scheduling when it is possible, since it allows the optimal load balancing to be selected beforehand, whereas there is an overhead associated with the ongoing scheduling that needs to occur in the dynamic case.
There are a number of pitfalls to be avoided in parallel programming. One of these is called "deadlock". This occurs when one processor is waiting to receive data from another but if that processor is also waiting to receive data, we can have a situation where neither is able to continue and the program becomes stuck at that point.

Another problem called a "race condition" can arise in shared memory models. This can occur in programs such as the one shown above. The variable \textit{sum} will be shared in the same memory space between the processing units. In the line \texttt{sum = sum + f(i)}; one processor might read the value and then before it has had time to update it, another processor could read, and so will add to the old value. Race conditions can be avoided by giving each processing unit its own value of \textit{sum} which we add up to the total at the end, or by various mutual exclusion mechanisms which allow only one processor to carry out some lines of the code at any one time.

A metric of the success of a parallel program is the speedup, which measures the factor by which the parallel version runs faster than the sequential version. It is given by \( T_1 / T_P \), where \( T_1 \) is the runtime of the sequential code, and \( T_P \) is the runtime of the parallel version.

A simple model of the speedups possible in a parallel program is given by Amdahl’s Law. This is based on considering the fraction of the program which can be parallelised. If we denote by \( \gamma \) the fraction of the code which is serial, then the serial part of the program will take time \( \gamma T_1 \), where \( T_1 \) is the runtime of the serial code, and the parallelisable part of the program will take time \( (1 - \gamma)T_1 \). In the parallel parallel, we make the assumption that the time for the parallel section will be \( (1 - \gamma)T_1 / P \), where \( P \) is the number of processors, because each will take an equal fraction of the work. The serial section will take the same time. The total runtime of the parallel program is then \( T_P = \gamma T_1 + (1 - \gamma)T_1 / P \). Therefore, the speedup on \( P \) processors is given by

\[
S(P) = \frac{T_1}{\gamma T_1 + (1 - \gamma)T_1 / P} = \frac{1}{\gamma + (1 - \gamma)/P}.
\]  

(2.1)

This is Amdahl’s Law. If we plot the speedup against the number of processors, we see that the speed up increases as \( P \) increases, but eventually reaches a limit. If we take the limit as \( T \to \infty \), we see that the maximum possible speedup is given by \( 1/\gamma \). This represents the case where we are using a very large number of processors so that the time in the parallel section is negligible. The serial time will remain the same, however, so this limits the speedup that is possible.

This maximum speedup may not be achievable in practice. There may be a certain overhead involved in setting up the parallel section during a run. In addition, Amdahl’s
Law does not take into account that communication may be required between processes, which will slow down the parallel section.

2.2 Programming models

There have been a large variety of programming models developed for use in parallel programming.

2.2.1 MPI

Message Passing Interface (MPI) is a programming environment which has been developed for distributed memory systems. The model is of processes running on different processors which can communicate with each other by passing messages.

MPI operates on a Single Program Multiple Data (SPMD) model. Each processor runs the same program. In the program, it is possible for a processor to find its rank, i.e. its number amongst the processors running this program. This gives a way for different processors to perform different actions, for example using an if statement to select a block of code to be performed on a particular processor. As another example, this can be used to parallelise a loop by assigning different loop to different processors. This can be done by using the processor rank in the loop limits. Each processor will then arrive at the for loop and decide which iterations to run based on its rank.

2.2.2 OpenMP

OpenMP is a programming environment designed primarily for shared memory systems. It provides extensions to the C, C++ or Fortran languages which allow parallel sections of a program to be specified. In a program, it is accessed mostly via compiler directives and a few library function calls. OpenMP operates on a fork-join model of parallelism. At the beginning of the program, one master thread is in operation. When a block is reached which the programmer has specified to run in parallel, a "team" of threads is created which are executed in parallel. When the end of the block is reached, the thread are joined back into one thread. There can be many of these blocks in a program.

OpenMP is designed to make it easy to develop a parallel program starting from a sequential version of the code. The programmer can incrementally look through the code and look for an opportunity for parallelism, and then write the OpenMP compiler directives at that point to specify that this block should be run in parallel.
Chapter 2. Literature Review

2.3 Molecular dynamics

Computer simulation in Physics is a mode of study that sits between theory and experiment. It can be used to test theoretical results by providing essentially exact results for model under study. It can also be used directly to help understand experimental results.

The field often called “molecular simulation” is concerned with understanding the bulk properties of matter, i.e. we are interested not in the details at the microscopic level, but are rather studying the properties on the everyday macroscopic lengthscale, where there is a collection of many billions of molecules. The properties of interest include thermodynamic properties such as temperature, pressure, entropy and heat capacity. We are also interested in structural properties which probe how the particles are arranged in the system and dynamical properties which probe the motion of particles in the system. Often an aim is to study the phase behaviour of the system, i.e. to study which phase the system will be in under different conditions of temperature, pressure etc.

The quantities we measure in a macroscopic system are an average of the behaviour of the individual particles. There are two approaches to calculating the so-called “ensemble averages”. One is the “Monte Carlo” method in which configurations of the system (positions and velocities of the particles etc.) are sampled at random from the relevant probability distribution. The second approach is “Molecular Dynamics”. The idea here is to simulate the natural dynamics of the particles in the system. Over time, the system will visit states with the appropriate probability, and the ensemble average of a quantity is available as the time average of that quantity.

The physical systems which we wish to study in the macroscopic regime contain of the order of $10^23$ particles. In statistical physics, it is usual to consider the behaviour of a system in the ”thermodynamic limit”, i.e. in the limit of an infinite system size. In order to mimic an infinite system, it is in the norm in molecular simulation to apply periodic boundary conditions. We notionally make copies of our system along each edge of the central simulation box. Thus the particles at the edges of the simulation box experience an environment which is more representative of the environment it would experience in a bulk system.

Periodic boundary conditions are usually applied by using the ”nearest image convention”, i.e. when considering an interaction between two particles, $i$ and $j$, we take use the distance from $i$ to the closest of the periodic images of $j$. 
In Molecular Dynamics, we start with a model of how the particles in the system interact with one another. We then use Newton’s Second Law, \( F = ma \) to calculate the acceleration of each particle at each point in time. This leads to a set of coupled differential equations - the equations of motion.

The equations of motion are often integrated using finite difference algorithms. These algorithms are based on a Taylor series expansion of the equations of motion. The particle positions and velocities are advanced through small steps in time.

One finite difference algorithm which is frequently used in Molecular Dynamics simulations is the Velocity Verlet algorithm. The equations for the update of the positions and velocities at each timestep are:

\[
\begin{align*}
\mathbf{r}_i(t + \Delta t) &= \mathbf{r}_i(t) + \mathbf{v}_i(t)\Delta t + \frac{1}{2} a_i(t)\Delta t^2 \\
\mathbf{v}_i(t + \Delta t) &= \mathbf{v}_i(t) + \frac{1}{2} [a_i(t) + a_i(t + \Delta t)] \Delta t,
\end{align*}
\]

where \( \mathbf{r}_i(t), \mathbf{v}_i(t), a_i(t) \) are the position, velocity and acceleration of particle \( i \), respectively.

For each timestep, the operations carried out are as follows:

- Using the \( a_i(t) \) calculated in the previous timestep, the particles’ positions are updated using Eq. (2.2)

- The particle velocities are partially updated using

\[
v_{\text{partial}}(t + \Delta t) = v(t) + \frac{1}{2} a(t)\Delta t
\]

(2.4)

- The forces at time \( t + \Delta t \) are calculated using the new positions, allowing \( a(t + \Delta t) \) to be found.

- The update of the velocities can now be completed.

\[
v(t + \Delta t) = v_{\text{partial}}(t + \Delta t) + \frac{1}{2} a(t + \Delta t)\Delta t
\]

(2.5)

For the force calculation, the force on a particle is the sum of the forces due to all the other particles. We therefore need a nested loop to evaluate the interaction of each particle with every other particle. We can reduce the number of iterations that need to be considered by taking account of Newton’s Third Law, which says that the force on particle \( i \) due to particle \( j \) is equal and opposite to the force on \( j \) due to \( i \). We therefore
only need to iterate over all the pairs of particles. This can be achieved by using the following limits in the for loops:

```c
for(i=0; i<N-1; i++) {
    for(j=i+1; j<N; j++) {
    ...
    }
}
```

### 2.4 Ferrofluids

The physical system of interest here is the ferrofluid. This is a colloidal suspension of nanoscale ferromagnetic particles dispersed in water or in an organic solvent. The colloidal particles are around 10nm in diameter and made of a ferromagnetic material such as iron, cobalt or magnetite ($\text{Fe}_3\text{O}_4$). Ferrofluids are magnetic fluids whose flow behaviour can be modified by the presence of a magnetic field and can be attracted by a magnet.

At the microscopic level, an interesting feature is the tendency for the particles to form into chains. This is caused by neighbouring particles rotating so that their magnetic dipole moments line up in a nose-to-tail fashion, which is energetically the most favourable configuration. These chains have been imaged experimentally in microscopy studies [Puntes et al. 2001]. Fig. 2.1 shows the chaining in a monolayer of cobalt ferrofluid. It should be noted that because ferrofluids are quite opaque, microscopy studies are usually carried out on thin films of the fluid. Often these films are roughly the thickness of the colloidal particles themselves, so we can regard the fluid in this situation as being quasi-two-dimensional.

In simulation studies, ferrofluids are often modelled as spherical particles with a point magnetic dipole at their centre, with a short range repulsive potential which prevents
the particles from overlapping when they come into contact - the “dipolar hard sphere” or “dipolar hard sphere” model. There have been many simulation studies of this model, both by Monte Carlo [Camp and Patey 2000], Levesque and Weis [1993], Weis [2002], Weis et al. [2002] and by Molecular Dynamics Duncan and Camp [2004], Murashov et al. [2002]. Some studies have considered the quasi-two-dimensional case, which is relevant to the experimental situation described above. Chaining is also observed to occur in simulation.

In the present study, we will be considering ways to parallelise a Molecular Dynamics simulation of a quasi-two-dimensional dipolar soft sphere fluid. The particles are confined to a two-dimensional plane, but their dipole moments are free to rotate in three dimensions. The interaction potential between each pair of particles is given by

\[ U_{ij}(r) = u_{SS}(r) + \frac{\mu_i \cdot \mu_j}{r^3} - \frac{3(\mu_i \cdot r)(\mu_j \cdot r)}{r^5}, \]  

(2.6)

where \( r \) is the interparticle separation vector and \( \mu_i \) and \( \mu_j \) are the dipole moments. \( u_{SS}(r) \) is the short range repulsive interaction, here taken to be:

\[ u_{SS}(r) = 4\epsilon \left( \frac{\sigma}{r} \right)^6, \]  

(2.7)

where \( \epsilon \) is an energy parameter and \( \sigma \) is a distance parameter.

We note that in molecular simulation, when dealing with short range potentials, it is usual to define a cutoff distance beyond which the interaction potential is negligible. There is no need to take into account interactions between particles when they are beyond the cutoff distance apart. On the other hand, potentials which do not fall off any faster than \( r^{-d} \), where \( d \) is the dimensionality of the system. The dipole-dipole interaction falls into this class. We must therefore take into account all pairs of particles.

### 2.5 Parallelising Molecular Dynamics simulations

In a Molecular Dynamics simulation, there are two main parts to the algorithm: first the forces (and torques where appropriate) on each particle are calculated, then there is the integration step where the positions and velocities of each particle are updated using these forces. The force calculation is usually by far the more expensive operation. The force on a particle is the sum of the forces due to all the other particles. Therefore the force calculation contains a double loop to iterate over all pairs of particles. This has quadratic complexity. Often this part of the program can take 80% or more of the time. For this reason, in parallelising Molecular Dynamics, effort is usually focused on
parallelising the force calculation. The integration routine involves a loop over particles, so there is some available concurrency, but given that this routine takes up only a relatively small percentage of the time, it is not usually worthwhile trying to parallelise this.

We note that there is a very close similarity between Molecular Dynamics and simulations in Astrophysics where systems of bodies interacting via a gravitational potential are modelled - in this context usually referred to as ”N-body simulations”. The basic algorithm to integrate the equations of motion is essentially the same as Molecular Dynamics. Hence there can be some crossover between the two fields in methods of parallelisation.

Another kind of phenomenon, which requires very large system sizes, is fluid dynamics - the study of the motions of liquids. These are usually studied very successfully using models which treat the fluid as a continuum. The phenomena involve length scales much greater than atomic dimensions and so would require very large scale simulations to model them atomistically. Rapaport reviews some of the work on applying Molecular Dynamics to study these phenomena [Rapaport 2014]. Typical system sizes require millions of particles. This has only been made feasible with advances in parallel programming.

Wilson et al. [Wilson and Ilnytskyi 2005] and Heffelfinger [Heffelfinger 2000] have reviewed the parallelisation of Molecular Dynamics simulations. There are three main ways of doing this. One of the simplest ways to parallelise is the “Replicated Data” method, which involves copying all the data for all particles onto each processor. This method is costly in terms of communication. The domain decomposition method associates a region of space with each processor. For short-ranged forces, this allows most forces to be calculated within one processor, with some communication needed on the boundaries. For long-ranged forces, the Replicated Data method can be used, but another algorithm, the Systolic Loop algorithm will reduce the cost of communication.

The Replicated Data method often involves very little modification to the sequential implementation. In this scheme, each processor will calculate the forces for a subset of the particles. Each processor still requires data on all the other particles in the system in order to calculate this, therefore every processor is needs to be given a copy of all the data (particle positions and velocities etc.). At the end of each step, the new data for each particle needs to be communicated to all processors. This scheme has the virtue of being easy to implement, but it has a high usage of memory, and it requires a large amount of communication between the processors. Because of the communication cost, this method scales rather poorly with the number of processors for larger number of processors.
Some scheme of load balancing is required for the Replicated Data method. Since we take account of Newton’s Third Law and loop over pairs of particles, as discussed in Section 2.3, we need to distribute the iterations of the outer loop to the processors to ensure the same number of pairs of particles is considered on each.

As mentioned before, for short-ranged potentials, it is usual to truncate the potential beyond a certain distance. Some methods of parallelisation take advantage of the fact that any interactions beyond this distance do not need to be taken into account. Plimpton (Plimpton [1995]) has compared three different ways of doing this which he termed “atom decomposition”, “force decomposition” and “spatial decomposition”.

In atom decomposition, each processor is assigned a fixed set of particles which it deals with, regardless of where these particles move in space during the course of the simulation. If we consider pairwise forces between all particles $i$ and $j$ which must be calculated, $f_{ij}$, we can think of these as forming a matrix. These contributions are later added to give the force on each particle. The atom decomposition method corresponds to one particular processor calculating one row from this matrix for every particle which resides on that processor. Some communication is required in order for a processor to know the positions of particles which belong to other processors. $O(N)$ communications will have to be carried out.

Plimpton’s “force decomposition” method divides the work differently. Instead of each processor being assigned a particular set of particles, it is assigned a block of the force matrix. Assigning the work in this manner means that less communication is required between processors; it is reduced to $O(N/\sqrt{P})$, where $P$ is the number of processors. Hence, this method scales better as we increase the number of processors.

The third method, spatial decomposition, is often called “domain decomposition”. Each processor is assigned a region of space and calculates the forces on the particles in this region. Particles within the cutoff distance of the edges of the region will need to communicate with processors dealing with neighbouring regions. Some communication is also required following the integration step to notify processors if a particle has moved across the boundary from one region to the next. However, the amount of communication between processors required is much reduced in the domain decomposition scheme because communication is required only for particles at the edges of each region (provided the forces are short-ranged).

The spatial decomposition method has proved very successful in parallelising Molecular Dynamics simulations. However, it can only deal with short-range forces. For long-range forces, the Replicated Data method is often used, but it has high communication costs. The Systolic Loop method was developed as a way to reduce communication
costs by spreading it out over a number of steps (Raine et al. [1989], Smith [1991]). The algorithm will be described in more detail in Chapter 3. Briefly, the particles are split into $2P - 1$ groups, where $P$ is the number of processors. One group is assigned to processor 0 and two groups are assigned to the other processors. Processor 0 calculates the intra-group forces between the particles in its group, the other calculate the inter-group forces between particles in their two groups. Communication now occurs, with the groups of particles and their force accumulators passed to neighbouring processors in an anticlockwise direction. This procedure is repeated $2P - 1$ times, by which time the forces between all pairs of particles has been calculated.

Smith (Smith [1991]) has made a comparison of the Replicated Data and Systolic Loop algorithms and discussed some of the practicalities of implementing these methods. The Systolic Loop algorithm was developed for the typical architectures of the 1990s - cluster computers with rather slow interconnects. It was therefore very important to try to reduce communication costs. This paper concentrates on optimising for the Intel iPSC/2 computer, a distributed memory computer with hypercube connectivity. In a hypercube of dimension $D$, there are $2^D$ nodes with each node connected to $D$ neighbours.

He makes an analysis of the scaling properties of the two algorithms from a theoretical point of view by presenting a simple model of the timings of the computation and communication steps of the force calculation. The time for a single timestep is given by $T_S = T_P + T_C$, where $T_P$ is the processing (computation) time and $T_C$ is the communication time. An important quantity for determining the efficiency of a parallel program is the ratio of the communication to computation time, $R_{CP} = T_C/T_P$. We want to reduce this to maximise the efficiency of the parallelisation.

For the Replicated Data, Smith gives the following model. The computation time is given by

$$T_P = \frac{N(fN - 1)}{2P} \alpha + N\beta + \frac{N^2}{2P}(1 - f)\alpha' ,$$  \hspace{1cm} (2.8)

and the communication time is given by

$$T_C = D(\gamma N + \delta) ,$$  \hspace{1cm} (2.9)

where $N$ is the number of particles and $P$ is the number of processors. The parameters $\alpha$ to $\delta$ are dependent on the computer being used.

In Eq. 2.8 $\alpha$ is the time to calculate the force for one pair of particles. $\alpha'$ is the time to calculate the inter-particle distance for pairs of particles whose forces are subsequently not calculated (in the case that we are using a cut-off for short range forces). $\beta$ is the time to perform the update of one particle’s position in the integration step. The
terms in Eq. 2.8 therefore have the following significance: the first term is the time to calculate the forces for all pairs of particles; the second is the total time to carry out the integrations; the third is the time used calculating the distance for pairs of particles outwith the cutoff distance.

In Eq. 2.9, $D$ is the dimension of the hypercube (the number of processors is therefore $2^D$), $\gamma$ is the time to communicate the data for one particle, and $\delta$ is the startup time for the communication.

Dividing Eq. 2.8 by Eq. 2.9 gives $R_{CP}$. For large $N$, this is approximately given by:

$$R_{CP} \approx \frac{2PD\gamma}{N(f(\alpha - \alpha') + \alpha')}. \quad (2.10)$$

Hence at large $N$, $R_{CP}$ is proportional to $1/N$, and so at larger $N$, the efficiency of the program becomes greater. It can also be seen that $R_{CP}$ is proportional to $PD$, which is a strong dependence on the number of processors.

In the case of the Systolic Loop algorithm, the computation and communication times are, (assuming $n = N/(2P-1)$ particles in each group):

$$T_P = (2P - 1)fn^2\alpha + 2n\beta + (2P - 1)(1 - f)n^2\alpha' \quad (2.11)$$

and

$$T_C = (2P - 1)(n\gamma + \delta), \quad (2.12)$$

where the machine-dependent parameters are the same as before. The factor of $2P - 1$ in the above equations takes account of the fact that $2P - 1$ systolic loop steps occur in each timestep. For the processors with two groups, each particle in the first group must be interacted with each particle in the second group, hence the $n^2$ dependence in the first term. Processor 0 has fewer interactions to calculate, and so the computations take longer, but it will be idle until the other processors are ready for the communication step, hence this has no effect on the overall timing.

As before, dividing Eq. 2.12 by Eq. 2.11 gives the communication to computation ratio, which for large $N$ becomes

$$R_{CP} = \frac{(2P - 1)\gamma}{N(f(\alpha - \alpha') + \alpha')} \quad (2.13)$$

Notice that there is a weaker dependence on the number of processors than in the Replicated Data case. This means that for larger numbers of processors (when $N$ is large), the Systolic Loop algorithm will be more efficient than the Replicated Data algorithm.
The systolic loop algorithm has been developed in a number of ways. Brugè (Brugè 1995) has created a mixed algorithm which combines the geometric decomposition with the systolic loop algorithm. The motivation for such a combination is a problem which can arise in the use of geometric decompositions, which is the algorithm typically used for short-ranged forces. If a large number of processors is used, it can be the case that the dimensions of the region of space assigned to each processor is smaller than the cutoff range for the potential. If this happens, there will potentially be interactions between a large proportion of particles in neighbouring regions, necessitating a large amount of communication. This makes the parallelisation very inefficient. To remove this problem, Brugè proposed an algorithm which partitions the simulation box into slices. A number of processors are assigned to each slice. The processors within each slice use a systolic loop. This allows a spatial decomposition for large numbers of processors by creating larger spatial regions which can then each be assigned more than one processor.

Lippert et al. (Lippert et al. 1998) point out the usefulness of the Systolic Loop algorithm in applications that go well beyond Molecular Dynamics. In astronomy, the simulation of bodies interacting via gravitational forces is essentially the same as Molecular Dynamics. But beyond that, many problems have the same structure of evaluating all the pair interactions between a set of elements. Other problems in this class include the calculation of convolutions in signal processing and autocorrelations in time series. It also applies to linear algebra operations such as vector addition and matrix multiplication. In all these cases, we need to calculate \( \frac{1}{2}N(N-1) \) interactions, as in Molecular Dynamics with long-range forces. The Systolic Loop algorithm can be of use in parallelising these cases where we need to calculate all pairs of interactions. Indeed, the systolic loop has found application in astronomical simulations (Dorband et al. 2003).

Although the systolic loop method re-distributes the communications to reduce their cost, the total number of communications to be carried out is \( O(NP) \), where \( N \) is the number of particles and \( P \) is the number of processors used, the same complexity as in the Replicated Data case. Lippert et al. (Lippert et al. 1998) proposed a modification of the Systolic Loop which reduces the amount of communication to be carried out down to \( O(NP^{1/2}) \). They term this the "hyper-systolic algorithm". Using this algorithm on a Cray T3D machine, they were able to attain a super-linear speedup, because there was a smaller volume of interactions between neighbouring processors, which relieved a cache memory bottleneck.

Some modern trends in Molecular Dynamics simulation are driven by the kind of computer systems that are available today. A major trend is the availability of multicore processors. Modern cluster architectures often consist of multicore processors networked together, and are thus a hybrid of distributed memory and shared memory system. It is
possible to program for such a system using MPI, but a more optimal implementation uses a hybrid of MPI for the distributed memory and OpenMP to exploit the shared memory capabilities of the multicores (Wu and Taylor [2013]). This hybrid scheme has been applied to Molecular Dynamics (Glass et al. [2014], Kunaseth et al. [2013]). In one study, this enabled a systems of up to 38.4 billion Lennard-Jones particles to be simulated on a 4800-processor computer (Watanabe et al. [2013]).

Pal et al. have applied the hybrid scheme to Molecular Dynamics and have investigated how well this method scales with number of processors (Pal et al. [2014]). They tested this scheme on a model of nano-indentation where a probe of carbon is pushed into a surface of chromium. They simulated various system sizes, ranging from 2331 Cr atoms to 132921 Cr atoms. They compared the scaling with number of processors when pure MPI is used and when the hybrid scheme is used. Their measurements were performed on a cluster consisting of 6 nodes, each containing 8 cores. They found in the MPI case that the speedup increased linearly with number of cores up until around 18 cores were used. Following that, the speedup started to level off up to the maximum number of cores in their system of 48. For the hybrid MPI/OpenMP scheme, they investigated configurations ranging from one OpenMP thread on one processor up to the maximum available on their system of 8 OpenMP threads on 6 processors. They found that the scaling was better than the pure MPI case when configurations consisting of the same total number of cores were compared. For example, the speedup achieved in the 6 x 8 system was significantly greater than the 48 processor system under MPI. This was due to the fact that less communication is required under the hybrid scheme. In the pure MPI, communication costs rise as more processors are used, and eventually the speedup gains from increasing parallelism is offset by the need for more communication.

Another modern trend is the general purpose use of Graphics Processing Units. A recent study by Polyakov et al. investigated the use of GPUs in a molecular dynamics simulation of a ferrofluid (Polyakov et al. [2013]). They compared a GPU computation using both a conventional all-pairs calculation of forces and a Barnes Hut implementation. The Barnes-Hut method is a method used in N-body simulations in Astronomy for the calculation of the forces. Usually, the Barnes-Hut method cannot be used for Molecular Dynamics because of the periodic boundary conditions, but in this case a cluster of particles was simulated without periodic boundary conditions. Simulating a system size of a million particles, they found a speedup of two orders of magnitude over a sequential implementation in the all-pairs case. In the Barnes-Hut case, the speedup was four orders of magnitude.

In the all-pairs case, the Molecular Dynamics algorithm is performed in two stages. First there are kernels which calculate the forces and the resulting increments to the particle
positions and orientations, then the second stage kernels apply those increments. In the first stage, each thread is assigned one particle to calculate the force on it. To speed up the calculation, shared memory is used to store data about the particles.
Chapter 3

Implementation and Evaluation

As we have discussed, in a Molecular Dynamics simulation, the force calculation is usually by far the more expensive operation, because it contains a double loop to iterate over all pairs of particles. Profiling using gprof revealed that the sequential program spends about 98% of the time in the force calculation. For this reason, in parallelising Molecular Dynamics, effort is usually focused on parallelising the force calculation. The integration routine involves a loop over particles, so there is some available concurrency, but given that this routine takes up only about 2% of the time, it is not usually worthwhile trying to parallelise this.

Because the particles in the physical system being simulated interact with long-range forces, it is not possible to use methods such as domain decomposition. These rely on the fact that for short-range forces, we can define a cut-off distance beyond which the interactions are negligible, and so these interactions do not need to be taken into account. In the present case, the long-range forces mean that we have no choice but to calculate the interactions between every pair of particles in the system.

All calculations were carried out on a Beowulf cluster consisting of 34 nodes of 8-core Intel(R) Xeon(R) 1596MHz processors connected through a CISCO 2984G full duplex 100Mb/s fast Ethernet switch.

3.1 Replicated Data algorithm

Perhaps the simplest way of parallelising a Molecular Dynamics simulation, and one which takes into account all interactions, is the ”Replicated Data” method. As the name suggests, this consists of passing all the data to every processor so that each can act independently in its calculation of the forces and torques. A subset of the particles
is assigned to each processor, and that processor is responsible for calculating the forces and torques on that particle. However, to do this the processor needs to have access to the positions and orientations of all other particles in the system, and this necessitates that all the data is sent to every processor at the start of each timestep. At the end of the timestep, each processor will send the forces and torques on its particles back to processor 0. Note that when we calculate the force on particle $i$ due to particle $j$, we also calculate the force on $j$, and so each processor also computes partial sums of the forces on particle $j$ and sends these to processor 0. A reduction operation must be carried out where processor 0 collects the partial sums and accumulates them to obtain the total.

The replicated data method allows us to exploit concurrency in the calculation of the forces and torques by allowing each act independently. However, this method is rather expensive in terms of communication, with all the data being sent to all the processors in one go.

In order to investigate the scaling properties of this algorithm, we have made measurements of the timing of the parallel program on different numbers of processors. This was carried out for simulations of different numbers of particles to examine how the algorithm scales with problem size. All programs were compiled using the mpich compiler using -O2 level of optimisation.

With 100 particles, the performance is very poor. The table shows timings for a single run of 200000 timesteps. There is a very slight speedup with up to 6 processors, but beyond that the program actually runs far slower as more processors are added. This is a clear sign that the communication time is dominating the execution. Dividing the small number of particles between processors means that each only has a small amount of computation to carry out and hence the communication to computation ratio is very high. This can readily be seen by making measurements of the time taken to perform the parts of the force calculation during a single timestep. As an example, in a particular run using six processors, it took 0.212s to communicate the co-ordinates at the start of the force/torque routine, followed by 0.149s to perform the calculation, followed by 0.008s to communicate the forces back to processor 0. It can be seen that the initial communication of co-ordinates is taking longer than the computation.

We also investigate problem sizes of 1024 and 10000 particles. For each measurement, the mean was taken of three separate runs. Absolute speedups were calculated relative to the original sequential program. Fig. 3.1 shows the speedups obtained with this algorithm. This shows a linear scaling with number of processors until we reach around 10 processors, where the speedup levels off at about 5. Beyond 15 processors, we find that adding more processors actually leads to poorer performance.
The scheduling of loop iterations in the measurements to this point has divided the iterations of the outer loop evenly between the processors. This in fact leads to poor load balancing because the limits of the inner loop depend on the value of $i$ in the outer loop (which was done in order to loop over pairs of particles).

The effect of this is demonstrated in Fig. 3.2. This shows the results of timing the different parts of the force/torque calculation using calls to MPI_Wtime(). Timings were made of the three main parts of the operation: the initial communication of coordinates to all processors; the force/torque computation itself on each processor; and the final communication of the forces and torques back to processor 0. Fig. 3.2 shows the timings for a 1024-particle system using 5 processors. The timings given are the average of 10 timesteps. The poor load balancing can clearly be seen in the fact that the higher numbered processors are taking longer to carry out the computational part of the procedure. It can also be seen that the higher numbered processors appear to be taking longer to carry out the initial communication step. What this is really showing is that these processors have completed the previous timestep earlier as a consequence of having less computation, and it cannot then continue until processor 0 has caught up and passed on the data needed to begin the next timestep. Having processors being idle while waiting for others to complete their work is an inefficient use of resources.

Here we will use static scheduling rather than dynamic scheduling, because the amount of work to be done can be predicted beforehand. The force/torque calculation for each pair
of particles will always carry out the same set of floating point operations regardless of
the actual values of the co-ordinates and velocities etc., and so each should take the same
amount of time. We can also work out how many pairs of particles will be calculated.
The static schedule is therefore to be preferred over the dynamic schedule, which is
harder to implement and has an overhead associated with it.

To balance the load, we first need to know how much work is being done for a particular
range of the loop index of the outer loop. Fig. 3.3 shows the set of $i$ and $j$ values. Let
us denote by $W(i_0)$ the total number of iterations of the inner loop when the outer loop

\[ i \]

\[ 0 \quad 1 \quad 2 \quad N-3 \quad N-2 \quad N-1 \]

\[ j \]

\[ 0 \]
\[ 1 \]
\[ 2 \]
\[ 3 \]

\[ N-2 \]
\[ N-1 \]
index, $i$ covers the range from $i_0$ to $N$, inclusive of $i_0$, but exclusive of $N$. This is a triangular section whose width is $N - 1 - i_0$. The height at $i_0$ is also $N - 1 - i_0$. Therefore the total work is given by

$$W(i_0) = \frac{1}{2} \left[ (N - 1 - i_0)^2 + (N - 1 - i_0) \right]. \quad (3.1)$$

To obtain the total work in the range from $i_0$ to $i_1$, we subtract the work from $i_1$ to $N$ from the work from $i_0$ to $N$:

$$W(i_1) - W(i_0) = \frac{1}{2} \left[ (N - 1 - i_0)^2 + (N - 1 - i_0) - (N - 1 - i_1)^2 + (N - 1 - i_1) \right]. \quad (3.2)$$

This gives the work done (the number of pairs of particles whose interactions are calculated) when the outer loop index covers the range $i_0$ to $i_1$, inclusive of $i_0$, but exclusive of $i_1$.

To divide out the work between different processors the following procedure is carried out at the beginning of the program. The total number of particle pairs to be considered was found. This value is divided by the number of processors to calculate how much work each should carry out in the ideal case. For processor 0, the first value of the $i$ index was fixed at 0. The value of the last index is incremented stepwise until the work given by Eq. 3.3 just exceeds the ideal amount. This boundary then forms the lower index value for processor 1. Its upper boundary is adjusted in the same way. This is repeated for all other processors until the last one, which is assigned an upper bound of $N$. This procedure does not give an exact load balancing, but is good enough for the purposes that we require, as is shown below.

The effect of the better load balancing is demonstrated in Fig. 3.4. This compares the speedup of the load balanced version the non-load-balanced version. It can be seen that there is an increased speedup in the load-balanced case. It is possible to extract a greater speedup for the same number of processors when there is improved load balancing. This holds up to about 12 processors, where the speedup levels off at about 5, a similar value as in the non-load-balanced case.

The detailed timings for a 5 processor run are shown in Fig. 3.5. The better load balancing is reflected in the fact that each processor is taking roughly the same time to carry out the computational part of the calculation. We also see the reported communication times are roughly equal because no processor is having to wait for another to finish. However, it can be seen that for the initial communication, processor 0 takes shorter time than for the other processors, whereas the situation is reversed in the final communication at
the end of the timestep. This is because at the beginning, processor 0 is sending out data, and move on to the next thing, while the other processors must wait to receive data before they can continue. At the end, the other processors pass their force data to processor 0, while processor 0 has to wait to receive this data from each other processor.

One reason for efficiency to decrease as we add more processors is because of the increased communication costs. To probe what is happening here, Fig. 3.6 shows the timings for the stages of the force/torque calculation for runs using different numbers of processors. The bars are normalised by the total time of the force/torque routine, and so the parts of the bars show the percentage of the time spent in each stage of the procedure. Notice first that the computation time decreases very in inverse proportion to the number of processors, which is to be expected when the work is divided evenly between processors. However, in conjunction with this, the communication time steadily increases as we add more processors which need to take part in communications. As we reach around 10 processors, the communication time has risen to become equal to the computation time. We will therefore no longer continue to have linear scaling with number of processors, and this corresponds to the point at which we start to see leveling off of the speedup graph (Fig.). Eventually, the increase in communication time as we add more processors dominates over the decrease in computation time. This is the cause of the decrease in speedup for larger numbers of processors.
Figure 3.5: Timings in milliseconds for the parts of the force/torque calculation with load balancing.

Figure 3.6: Percentage of time spent (on processor 1) in the stages of the force/torque calculation for runs with different numbers of processors. $N = 1024$. 
It is to be expected that better scaling can be achieved when we simulate larger number of particles. This is because the number of pairs of particles to be considered in the force/torque calculation scales roughly as $N^2$, whereas the amount of data to be communicated scales as $N$. Therefore, the communication to computation ratio should decrease as we simulate larger numbers of particles. To that end, we have also obtained results for a 10000-particle system. This is far more computationally intense than the 1024-particle case, so results were obtained for runs of only 25 timesteps. Fig. 3.7 compares the speedups obtained in the 1024-particle system and the 10000-particle system using the load-balanced version of the replicated data program. As expected, the performance is much better for the larger system. There is linear scaling with number of processors up to 15, and the maximum achievable speedup has increased from 5 to around 14. Here, the load-balancing leads to a significant improvement in speedup.

### 3.2 Systolic Loop algorithm

An alternative algorithm which aims to reduce the communication costs is the so-called "systolic loop" algorithm. The algorithm is illustrated in Fig. 3.8. The particles are divided into $2P-1$ groups, where $P$ is the number of processors. Group 0 is assigned to processor 0. Two groups of particles are assigned to each of the other processors, so that processor $i$ has groups $i$ and $2P-1-i$ as its "home" groups. The forces and torques are
then calculated for these groups. Processor 0 calculates the interactions between all pairs of particles in its one group, the other processors calculate the inter-group interactions between the two groups it stores. A systolic pulse now occurs, in which the groups of particles are passed in an anti-clockwise fashion to the neighbouring processors, along with the forces and torques accumulated so far. For example, processor 2 passes the positions, orientations, and the forces and torques for its first group to processor 1, while it passes the same variables for its second group to processor 3. Note that it is not necessary to communicate the velocities and angular momenta. The process can now be repeated, with each processor calculating the interactions of its new groups of particles. When this procedure of calculating interactions followed by communication has been repeated $2^P - 1$, all the particles have now arrived home, and we have considered the interactions between all groups of particles. Following this, each processor will carry out the integration step separately on their own groups of particles, since there is not one processor which has access to all the particles’ co-ordinates.

The rationale for the systolic loop algorithm is that it should reduce the cost of communication. The communication is staggered in stages, rather than occurring in one go as in the replicated data algorithm. In addition, the communication can be overlapped because the data is not be transferred from one processor to all the others or from every processor to a single processor. Instead, communication is occurring between neighbouring processor which can occur simultaneously with transfers between other processors.

A second benefit is that the algorithm is naturally load-balanced.

In (Ref.), it is also pointed out that the systolic algorithm requires less memory, because the particle variables are distributed on different processors and there is not one node that requires access to all the particles. This was a very important point when the systolic loop algorithm was created in the late 1980s, but is not such an important consideration on modern computers with large memories.
The systolic loop algorithm is considerably more difficult to implement than the replicated data method. A lot of code needs to be written to handle moving data to the right place. Great care is required to make sure data is being passed around correctly. This kind of code can be rather error prone. It is very easy to make a mistake with indices or counting how many particles are in a group. This illustrates a disadvantage of using a low-level model of parallelism such as MPI. Although it gives great control, a lot of code can be required simply to deal with message passing. In the present case, the sequential code has around 700 lines, but the systolic loop implementation increased this to around 1700 lines. The actual numerical algorithm can be lost in all this extra code.

We have made measurements of the speedups obtained using the systolic loop algorithm, again for 1024- and 10000-particle systems. The results for the 1024-particle are shown in Fig. 3.9. The speedups are comparable to those obtained using the replicated data method without load-balancing. The load balanced replicated data method is still superior to the systolic loop method.

It is in larger systems that the systolic loop algorithm makes a difference. Fig. 3.10 shows the speedups obtained for the 10000-particle system. For small numbers of processors, again the replicated data method obtains a slightly better speedup. Above about 15 processors, the systolic loop algorithm has the advantage. The systolic loop algorithm has approximately linear scaling all the way up to 30 processors, while the replicated
data algorithm is already beginning to show a levelling off in speedup. Thus, it is apparent that the advantage of the systolic loop algorithm is its better scalability. If we need to simulate large numbers of particles, this gives us the best performance for larger numbers of processors.

To investigate the causes of these observations further, we have made time measurements of the stages in the force/torque calculation. In the Replicated Data case, as above, the time for the initial communication, the computation and the final communication have been measured. In the Systolic Loop case, the time for the computations and the communications were summed up over the $2P - 1$ steps of the systolic loop to give a total for each over one timestep.

Fig. 3.11 shows the results for a run using 6 processors. We can see the load balancing is good in both cases - the time for the computation is roughly equal on all processors. The total communication time is the same for both algorithms. The computations are slightly slower in the Systolic Loop case, which accounts for the fact that the Systolic Loop algorithm is slightly slower for runs with lower numbers of processors. There are two probable reasons for this. Firstly, in the Systolic algorithm, the processors with two groups of particles will calculate a slightly greater number of particle interactions than any of the processor do in the load balanced Replicated Data algorithm. This is because in the calculation of inter-group interactions, it is necessary for both the outer and inner loops to loop over the full range of values. Secondly, in the implementation of
Chapter 3. Implementation and Evaluation

the Systolic Loop code, it was necessary to abstract out the code calculating the force and torque for a single pair of particles into a separate function. This was necessary because otherwise developing the systolic code would have been far more complex. There is, however, a slight performance penalty for this because of the overhead of the function call.

Fig. 3.13 shows the timings for processor 3 (arbitrarily chosen). In the 6 processor run, this shows that the increased computation time for the Systolic Loop is what leads to a slightly longer overall time for the timestep.

Fig. 3.12 the communications times for the two algorithms for a run using 24 processors for one timestep, plotted as a function of the processor number. As before, the in the case of the Systolic Loop, the sum over all $2P - 1$ iterations is shown. A striking feature of the figure is that for the Replicated Data method, the initial communication step seems to take longer on the higher numbered processors, whereas the reverse is true for the final communication step. In the initial communication, the data is being sent from processor 0 to each other processor in order. Later processors are having to wait until the data is sent to the earlier processors. It could be that the volume of communication is exceeding the bandwidth of the network links. The apparent reversal of the situation in the final communication may be due to the fact that the lower numbered processors have been able to start their computations earlier, and so arrive slightly ahead of time and the communication section.

**Figure 3.11:** Time in seconds for the parts of the force/torque calculation for a run using 6 processors. The timings are shown for each processor.
Chapter 3. *Implementation and Evaluation*

Figure 3.12: Communications time for each processor in a 24-processor run. Shown are the times initial and final communication and the sum of these in the Replicated Data case, and the total communication time over one timestep for the Systolic Loop case.

Note that in the systolic case, processor 0 has fewer computations to carry out, so must then wait for the communications to begin, which explains the apparently very long communication time.

Fig. 3.13 also shows the computation and communication times for processor 3 in a 24-processor run. It can be seen that even the total computation time is marginally longer in the Systolic Loop case than in the Replicated Data case, this is more than compensated for by the reduction in communication time.

Our study is being carried out on a cluster of multicore processors with 8 cores per node. So far, we have only considered the distributed memory system, making use of one core per node. Some gains can be made by making use of the shared memory on the multicore nodes. Fig. 3.14 shows a comparison of the speedups obtained using a single core on different nodes versus using the same number of cores on a single node. It can be seen that scalability is enhanced in the shared memory case, because of the far shorter time required for communication.

Fig. 3.15 shows the speedups which can be obtained by using several cores on each node, to demonstrate what can be achieved by making full use of the processing units available on the cluster. What can be seen is that the Systolic Loop far outperforms the Replicated Data method on large numbers of cores. In the Replicated Data case, the
Figure 3.13: Comparison of the total communication and computation times for one timestep on processor 3 in runs using \( P = 6 \) or 24 processors.

Figure 3.14: Comparison of speedups obtained for the Systolic Loop and Replicated algorithms using distributed memory or shared memory. \( N = 1024 \).
speedup levels off at around 40 cores at a speedup of around 14, whereas the Systolic Loop levels off only after 100 cores, and attains a limiting speedup of around 40. Thus, the Systolic Loop clearly has the advantage of greater scalability.
Chapter 4

Conclusions

We have successfully achieved speedups using both methods.

The first algorithm investigated was the Replicated Data method. This is the one of the simplest algorithms for parallelising systems with long-range forces which require the evaluation of the interactions between every particle in the system. In this method, at the beginning of the force/torque calculation, all the particle co-ordinates are communicated to all the processors. Each processor then calculates the forces/torques on the particles assigned to this processor due to all other particles. These forces are then communicated back to processor 0, which adds up all the partial contributions to obtain the total force and torque on each particle.

We were able to obtain reasonably good speedups with this method. For a 1024-particle system, there was linear scaling with number of processors until around 10 processors, where the speedup started to level off. The limiting value of the speedup was around 5.

One problem that needs to be addressed is the poor load-balancing. In the nested loop over particles in the force/torque routine, we need to loop over pairs of particles. This means that different iterations of the outer loop contain different numbers of iterations in the inner loop. This was illustrated by making measurements of the timings for the different stages of the algorithm - the initial communication of particle data, the computation itself, and the final communication of the calculated forces/torques. This showed that the higher numbered processors, which had less work to do, took less time to complete the computation part, and hence were waiting idle for the other processors to catch up.

We have derived an equation which counts the number of iterations in the inner loop for a particular range of the outer loop index, and used this in a procedure to assign iterations of the outer loop to different processors so that each is given an approximately equal
amount of work to do. The improved load-balancing led to slightly greater speedups for a given number of processors. The better distribution of work was reflected in the detailed timings of the parts of the algorithm - each processor now took roughly the same amount of time to carry out the computation. The timings also showed the increased time taken on communication as more processors were used. With around 12 processors, the communication time has risen to be equal to the computation time. This corresponds to the levelling off of the speedups which occurs around this number of processors.

The performance of the Replicated Data method was found to improve when a larger system was being simulated. With a 10000-particle system, the limiting speedup was around 15. This reflects that computation costs scale as \(N^2\), whereas communication costs scale as \(N\), so communication is relatively less important compared to the 1024-particle system for the same number of processors.

In the Replicated Data method, a large amount of communication is required to pass around the particle co-ordinates and the force matrices calculated on each processor, so it is desirable to consider algorithms which could reduce the burden of communication, especially if we want to simulate large systems or use large numbers of processors. The systolic loop algorithm is one such algorithm. In this algorithm, the particles are split into \(2P - 1\) groups, where \(P\) is the number of processors. One group is assigned to processor 0, two groups to the other processors. Processor 0 calculates the forces/torques between the particles in its group, while the other processors calculate the inter-group forces/torques between its two groups. A communication step then occurs where the groups are sent anti-clockwise to neighbouring processors. The forces/torques of the new groups are calculated, followed by another communication step. This is repeated \(2P - 1\) times, and all the groups have now arrived home, and all pair interactions have been evaluated. This algorithm splits up the communication process, so that it occurs in batches rather than all in one go at the start and end of the force/torque calculation.

We found that for a 1024-particle system, the systolic algorithm actually produces speedups comparable to the non-load-balanced replicated data case, and performs slightly more poorly than the load-balanced case. In larger systems, the communication costs of the Replicated Data method become more important, and the systolic loop algorithm is a better choice of algorithm.

In conclusion, we have achieved a useful speedup of a Molecular Dynamics program by using two different methods of parallelising the program. The Replicated Data method involves sending all the particle data to all nodes, and so is quite expensive in terms of communication. The Systolic Loop algorithm staggers the communication and so reduces its burden.
It was found that for a system size of 1024 particles, the Replicated Data method performs slightly better than the Systolic Loop. A speedup of around 5 is achievable, which is still useful for simulations of this size of system. Typical studies of thermodynamic properties use around this number of particles, and so in this case the Replicated Data method appears to be the best choice. The Systolic Loop method is considerably harder to program, and so it is probably not worth using in this case.

The Systolic Loop is, however useful if we want to simulate very large systems of particles. Our results for a 10000-particle system show that the systolic loop algorithm out-performs the replicated data algorithm when we use larger numbers of processors. We will need to use large numbers of processors to obtain an adequate speedup for very large scale simulation, making the systolic loop a better choice for study of phenomena which occur over large lengthscales, such as hydrodynamic phenomena or biological systems.

Further work could include trying non-blocking communications. This can be done by using the MPI functions MPI_Isend() and MPI_IRecv() in place of MPI_Send() and MPI_Recv(). The non-blocking calls allow the program to continue without waiting for the other end of the communication. This would allow more overlapping of the communication and computation. Extra care is needed, however, to ensure that data has arrived before it is put to use. MPI provides constructs to do this.

Another extension would be to try a hybrid MPI/OpenMP implementation. This would take better advantage of the shared memory architecture in the multi-core processor on each node than using pure MPI. This might proceed by taking the MPI implementation and directing OpenMP to parallelise the outer loop of the force calculation using. In using such a shared memory model, we must take care to avoid any potential race conditions.
Bibliography


