



Parallel Monte Carlo Simulations of Vapour-Liquid Systems

Heather Kelly

August 24, 2006

MSc in High Performance Computing

The University of Edinburgh

Year of Presentation: 2006

Abstract

The aim of this project was to parallelise a serial code which simulates the behaviour of a fluid near its vapour-liquid critical point. The code was written by Philip Camp from the School of Chemistry, and could initially simulate a system of thousands of atoms on a desktop machine. The parallelised version should allow this number to be increased by at least two orders of magnitude. The parallelisation was done using MPI, and was run on a Sun Fire 15K [2].

Contents

1 Introduction **1**

2 Background **2**
2.1 Physics of the System 2
2.2 Motivation 3
2.3 Monte Carlo 3

3 The Serial Code **5**
3.1 Details of the Code 5
3.1.1 Model Potential 5
3.1.2 Cell List 6
3.1.3 Metropolis Step 7
3.1.4 Program Control 8
3.1.5 The Radial Distribution Function 8
3.1.6 The Structure Factor 9
3.2 Portability Issues 9
3.3 Testing and Validation 9

4 Parallelisation Strategy **11**
4.1 Communication 12
4.2 Output 12
4.3 Parallel Random Number Generation 13

5 Parallel Code **14**
5.1 Main Routine 14
5.2 Parallel Initialisation 15
5.3 Start 16
5.4 Halo Swapping 19
5.5 Calculating the Potential 22
5.6 Move 24
5.7 Acceptance 25
5.8 RDF and S(q) 26
5.9 Output 27

6 Conclusion **28**

A Risk Analysis **30**

B Timeline **31**
B.0.1 Progress Analysis 32

List of Tables

B.1 Original project timeline 31
B.2 Actual project timeline 32

List of Figures

2.1 Temperature-density phase diagram showing the liquid and vapour regions. The area under the curve is where vapour and liquid can coexist, to the left only vapour and to the right only liquid. As the critical temperature (T_c) and density (ρ_c) are approached, $\Delta\rho$ becomes smaller and smaller until the critical point is reached. 2

3.1 On the left, a standard Lennard-Jones potential. On the right, a cut-and-shifted Lennard-Jones potential. 5
3.2 Splitting up the simulation box into cells of side Rc. 6
3.3 Numbering of array CELLXYZ. Numbers vary fastest in the z direction and slowest in the x. 7

4.1 A 2D slice through the cell structure of a single processor, showing haloes. 12

5.1 If the number of cells is not divisible equally by the number of processors, the extra cells are given to the highest-ranked processor in each direction. This ensures that adjacent halo sizes match up. 19
5.2 Plimpton's Scheme, diagram based on [16]. A left-right swap is done first, followed by an up-down swap that includes some of the data from the first swap. Finally there is a forward-back swap, including data from the previous two swaps. 20
5.3 The spatial coordinate system across processors, also showing the processors' topology coordinates: (0,0,0) etc. These are the coordinate ranges for each processor's own data - the halo cells mean that there is an extra cell-length in each direction, overlapping with the coordinates of adjacent processors. 25

Acknowledgements

Kevin Stratford, Alan Gray and Philip Camp were the supervisors for this project. It was based on a serial code written by Philip Camp.

Chapter 1

Introduction

This project is based around a serial code which simulates the behaviour of a fluid near its vapour-liquid critical point. The aim was to parallelise it using MPI, enabling it to simulate far larger numbers of particles than the serial version can.

This report first explains the background physics of the simulation, lays out the general Monte Carlo method, and mentions some motivation for the project. This is followed by a detailed look at the serial code supplied for the project, and the strategy taken to parallelise it. Details of the parallel code are then explained, followed by the conclusion. The appendices include a risk analysis made at the beginning of the project, plus the projected and actual timelines, with some progress analysis.

Chapter 2

Background

2.1 Physics of the System

The vapour-liquid critical point of a fluid is the temperature, density and pressure where vapour and liquid coexist. At this point, the system does not behave exactly as a vapour or a liquid, but somewhere inbetween. A slight change in conditions will push it to become fully one or the other. It forms a structure consisting of vapour and liquid regions, which can exist on all length scales. This structure determines the physical properties of the system.

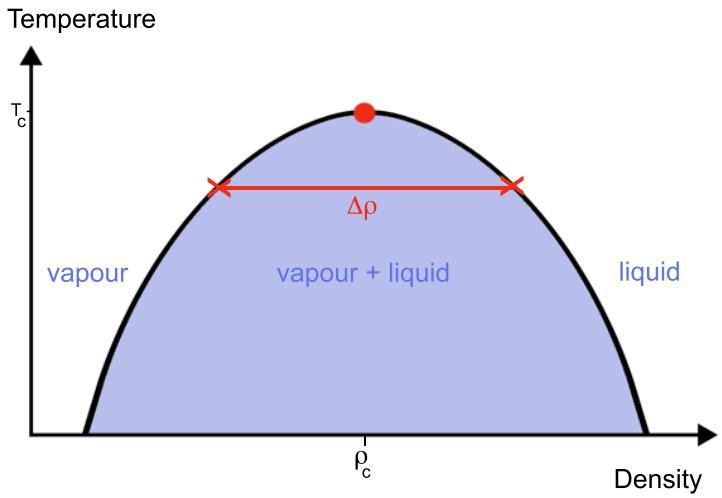

Figure 2.1: Temperature-density phase diagram showing the liquid and vapour regions. The area under the curve is where vapour and liquid can coexist, to the left only vapour and to the right only liquid. As the critical temperature (T_c) and density (ρ_c) are approached, $\Delta\rho$ becomes smaller and smaller until the critical point is reached.

Figure 2.1 is a temperature-density phase diagram showing the regions where the fluid is in each state. The area under the curve is where vapour and liquid can coexist, to the left only vapour and to the right only liquid. As the critical temperature (T_c) and density (ρ_c) are approached, $\Delta\rho$ becomes smaller and

smaller until the critical point is reached. The behaviour of the system close to this point is what this project simulates.

There are several ways to simulate a "particles in a box" system, including molecular dynamics and Monte Carlo methods. Molecular dynamics uses equations of motion to track the trajectory of each particle. This means there is a limit on the size of timestep which can be used, as it has to be small enough to accurately calculate the motion. Even with short range interactions between particles, this is more costly than Monte Carlo. Monte Carlo explores possible configurations of the particles in a statistical manner and so does not have this limitation on timestep. Various physical properties can then be calculated using the simulation data (eg. the radial distribution function and structure factor).

2.2 Motivation

The behaviour of the fluid near its critical point is dominated by very long-range correlations between its particles. Usually, such a system has to be simulated using finite-size scaling, which is used to scale up the finite model simulated on the computer in order to extrapolate the critical constants of an infinite system from the results of the finite system. An application of finite-size scaling to the properties of the cut-and-shifted Lennard-Jones potential is shown in [3].

Instead of having to do finite-size scaling, this project wishes to output data which will allow the analysis of the correlation functions in a large simulation and determine under what conditions the correlation length "diverges". This is motivated by the theoretical paper [4] and is based on the idea [5] that the decay of the total pairwise correlation function should change from monotonic to exponentially damped oscillatory at a line in the temperature-density plane. This is known as the Fisher-Widom line.

In related work, [6] discusses a recent technique for improving the convergence of Monte Carlo simulations, but it is likely to be difficult to parallelise so is not used in this project. It is a rejection-free algorithm - every move is accepted, but it is non-local, which is not good for parallelisation. It involves overlaying a pivoted version of the particle layout with the original version in order to create clusters, and the particles belonging to the clusters are exchanged between the configurations.

2.3 Monte Carlo

Monte Carlo methods are based on random number generation. Here we are using the Metropolis [1] Monte Carlo method. The general algorithm is as follows:

```
1. Choose initial conditions
2. Generate random move
3. Accept or reject move based on criteria
4. Adjust acceptance conditions if necessary
5. Repeat 2 to 4 for all objects
6. Until stopping condition is reached
```

In this case, the initial conditions are the starting atom placements. The acceptance criteria is the energy of the system, and the aim is for the energy of the system to reach equilibrium. In order for it to do this

in a timely fashion, it is necessary to keep a check on the acceptance rate of moves. If too many moves are being rejected, it will take a very long time to reach an equilibrium configuration and may become stuck in a local minimum. If too many are accepted, it will go into too many unwanted configurations and may not converge. In either case, the rate should be adjusted by changing the acceptance conditions slightly. It is usual to require an acceptance rate of 50% [1].

Chapter 3

The Serial Code

The serial code is written in Fortran and consists of one file for the code (`mcljcell.f`), an include file (`mcljcell.inc`), an input file and a file containing the starting configuration. It generates four output files: a `.OUT` general output file, a `.RDF` file containing the radial distribution function, a `.SHC` file containing reciprocal-space data, and a `.SAV` file where the configuration is periodically saved so that the simulation can be restarted from that point if necessary.

3.1 Details of the Code

3.1.1 Model Potential

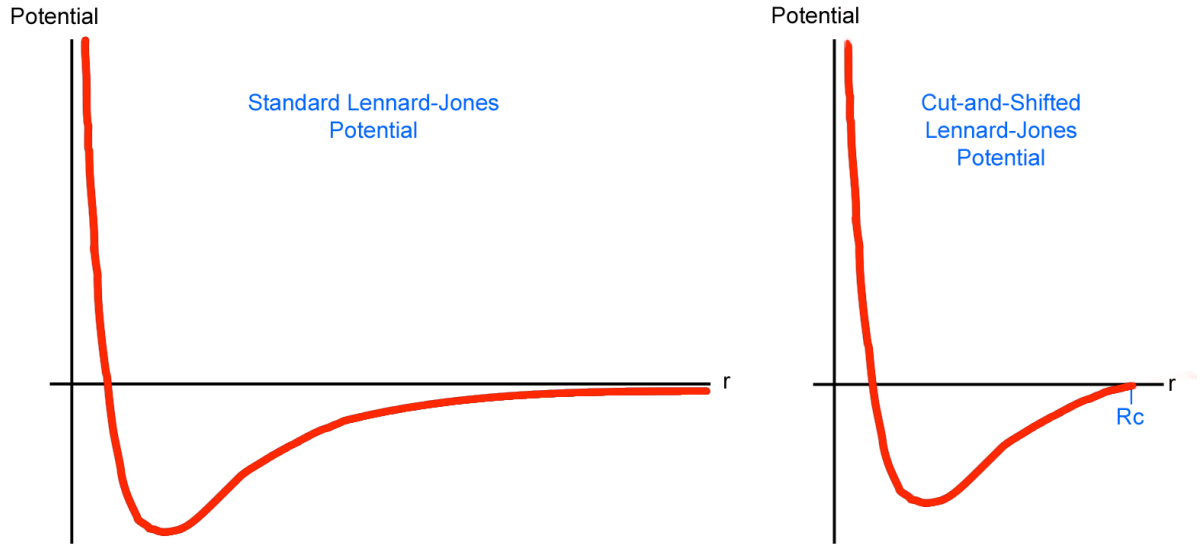

Figure 3.1: On the left, a standard Lennard-Jones potential. On the right, a cut-and-shifted Lennard-Jones potential.

The simulation uses a cut-and-shifted Lennard-Jones potential [1] to calculate the interaction between particles. This cuts off the potential at radius Rc, and shifts it by adding the potential evaluated at Rc to the potential at all points. In order to keep the interactions short-range, Rc is relatively small compared to the length of the simulation box.

3.1.2 Cell List

The simulation takes place within a cube with periodic boundary conditions. This cube is split into cells, each of which contains some number of particles. The dimensions of a cell are determined by Rc, the potential cutoff. Using this method guarantees that any particle may only be affected by particles in neighbouring cells.

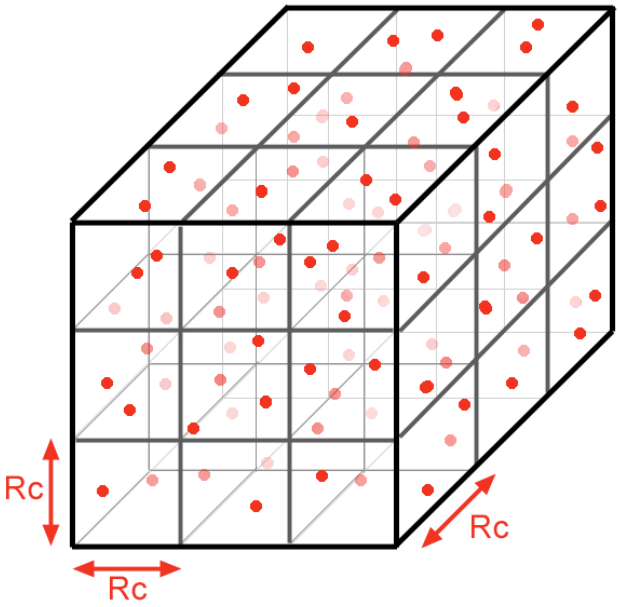

Figure 3.2: Splitting up the simulation box into cells of side Rc.

To store information about the location of each particle and their neighbours, there are a number of arrays of different dimensions. The cell structure itself is stored in a three-dimensional array, `CELLXYZ`. In the initialisation phase, the cells in this array are numbered, with numbers varying fastest in the z direction and slowest in the x.

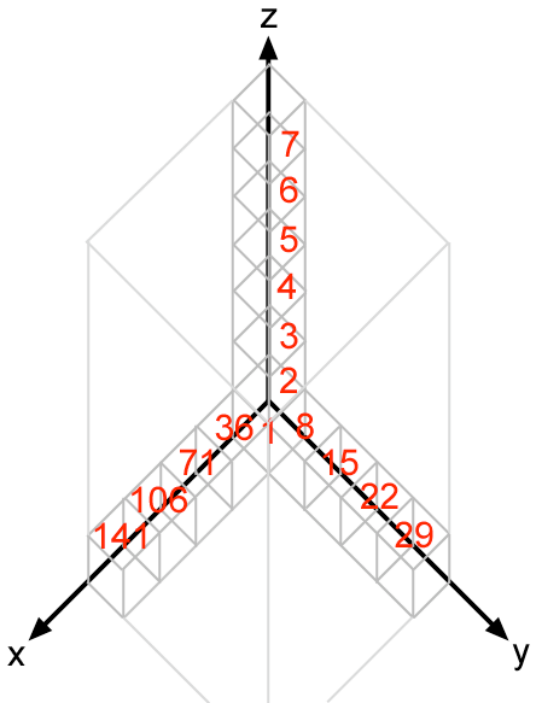

Figure 3.3: Numbering of array CELLXYZ. Numbers vary fastest in the z direction and slowest in the x.

The cells are then populated, according to the initial particle positions. Each particle is taken in turn, its coordinates are used to work out which cell it is in, and the cell number for that particle is stored in the one-dimensional `CELL` array. `CELL(I)` gives the cell for particle `I`. It is also necessary to keep a count of how many particles are in that cell - this is kept in the `NPART` array. Additionally, the `IPART` two-dimensional array is filled at this time. This is a "maximum no. cells" x "maximum no. particles" array - given a cell number and the local number of that particle in that cell, it gives the identifying number for that particle. Eg. if there were five particles in cell 2, `IPART(2,5)` would give the particle number of the fifth.

A cell neighbour list is then generated. This allows a cell to find out which numbered cells are its neighbours, and it takes into account the periodic boundary conditions. The `NEXTCELL` array is used for this. Given a cell and a neighbour number, it stores the cell number of that neighbour. Neighbours are numbered using a -1, 0, 1 displacement, with it varying fastest in the z direction and slowest in the x.

3.1.3 Metropolis Step

When moving a particle, the following steps have to be taken.

1. Store old coordinates, cell number and potential (this requires a call to the `POTI(I)` subroutine).

2. Get new random coordinates, find out what cell they are in and calculate the new potential.

3. Increment counter (irrespective of acceptance).

4. Calculate the energy change.

5. Accept or reject the move.

6. If accepted, update the system potential, update info for the new and old cells.

The acceptance condition involves calculating a Boltzmann factor. The move is accepted if a random number between 0.0 and 1.0 is less than the Boltzmann factor. The factor is set based on the energy change that the move would create divided by the system temperature (this quantity is referred to as `BETADELPOT`). If this is less than zero, the Boltzmann factor is set to 1.0, meaning this move will definitely be accepted. If it is greater than 75.0, the Boltzmann factor is set to 0.0, which is definite rejection. Otherwise, the Boltzmann factor is set to $e^{-BETADELPOT}$, and acceptance or rejection depends on the random number generated.

The code keeps track of the ratio of accepted moves and uses it to update `DRMAX`, a displacement parameter, at the end of each block of iterations. This is part of the output. The acceptance conditions are not altered during the running of the code, however.

3.1.4 Program Control

The top level of the code first does some initialisation, then consists of a triple-nested loop:

```
DO IBLOCK = 1, NBLOCK
    DO ISTEP = 1, NSTEP
        DO CYCLE = 1, N
```

The outermost loop iterates over blocks of calculation - averages are taken within these blocks, and there is output for each, including saving of state into the `.SAV` file. Within each block, there are `NSTEP` steps (`NSTEP = 1000` for the supplied input file) and for each step there are `N` particle movements. `N` is the number of particles in the system, which is 2048 for the supplied input file. The particle to be moved is chosen at random within the innermost loop.

Every 20 steps, the `AVCALC` subroutine is called. This calculates the radial distribution function g(r) and the structure factor S(q).

3.1.5 The Radial Distribution Function

The radial distribution function g(r) is a simple but effective way to describe the average structure of a fluid. It "gives the probability of finding a pair of atoms a distance *r* apart, relative to the probability expected for a completely random distribution at the same density" [1]. It can also be used to calculate the internal energy of the system.

3.1.6 The Structure Factor

The structure factor S(q) describes the Fourier components of density fluctuations in the fluid [1]. It is related by a 3D Fourier transform to the radial distribution function. It is also what is measured in experiments, so is a useful factor to be able to calculate. This calculation is the most computationally expensive part of the code, requiring the use of large numbers of Fourier components (wavevectors).

S(q) is defined as

$$S(q) = 1/N* < p(q)p(-q) >$$

where q is the wavevector, N is the number of particles and $<>$ denotes the expectation value (average over configurations). The wavevectors q are constructed using $q = 2\pi/L * (q_x, q_y, q_z)$. L is the simulation box length, q_x etc are integers. This restricts the wavevectors to fitting inside the simulation box. There are still many of these wavevectors - more than the number of particles. (`NQMAX`, maximum length of the q arrays, is 15 376 for the initial simulation). The S(q) calculation requires N sine and cosine evaluations for each wavevector.

3.2 Portability Issues

The code is written in Fortran 77, and there is a small issue when compiling and running using an f90 compiler. It occurs in the `OUTPUT` subroutine, when calculating and writing out g(r), because in some cases the log of 0.0 is being taken. F77 or g77 will return the result of this as `-Infinity`, which is then written out with no problem. F90 with default compiler options will throw an invalid floating point arithmetic exception, causing the program to core dump. To get around this, the code can be compiled using the `-ftrap=%none` compiler option. This switches off all floating point trapping, and is the default setting for f77. This was carried out on Lomond [2], a Sun Fire 15K server with UltraSPARC III processors, using Sun's Fortran compiler.

This is the way to cause the code to behave the same way when compiled with Fortran 90 or Fortran 77 compilers. However, the parallelised code will use f90's default trapping, and will deal with the 0.0 cases by including an `IF` statement in the calculation. This allows for more robust code, as there may be floating point exceptions elsewhere that should be caught.

3.3 Testing and Validation

It will be necessary to check the output of the parallel code against output from the serial code in order to validate the answer being given. Differences in floating-point rounding between the versions will lead to slight discrepancies, so there will need to be an error tolerance in the validation. The code already has built-in error checking of some things (maximum allowed values for example). It will be useful to compile the code with array bound checking on while debugging, though this must be switched off again for the finished versions as it causes a performance hit.

Example output from the `.OUT` file:

```
# LENNARD-JONES FLUID
# CUT OFF (& SHIFTED)...........   2.500000
# TEMPERATURE, kT/E.............   1.100000
# INVERSE TEMPERATURE, 1/T......   0.909091
# NUMBER OF PARTICLES...........       2048
# NUMBER OF BLOCKS FOR AVERAGES.        250
# NUMBER OF STEPS PER BLOCK.....       1000
# INTERVAL BETWEEN CALCULATIONS.         20
# TRANS. DISP. .................   0.000000
# RNG SEED 1....................      19497
# RNG SEED 2....................      18004
#  BLK TIME(s)      RHO      U/N   DRMAX    MOV
     0      1.   0.3150  -2.0887  0.0000 0.0000
     1    118.   0.3150  -2.1034  0.0000 0.0975
     2    236.   0.3150  -2.0946  0.0000 0.0973
     3    354.   0.3150  -2.1173  0.0000 0.0960
     4    473.   0.3150  -2.1015  0.0000 0.0969
     5    591.   0.3150  -2.1070  0.0000 0.0971
     6    709.   0.3150  -2.0854  0.0000 0.0986
     7    827.   0.3150  -2.1073  0.0000 0.0962
     8    945.   0.3150  -2.1018  0.0000 0.0974
     9   1063.   0.3150  -2.1122  0.0000 0.0961
    10   1182.   0.3150  -2.1145  0.0000 0.0962
    ...
   180  21295.   0.3150  -2.1037  0.0000 0.0971
   181  21413.   0.3150  -2.0979  0.0000 0.0970
   182  21531.   0.3150  -2.1057  0.0000 0.0963
   183  21650.   0.3150  -2.1039  0.0000 0.0966
   184  21768.   0.3150  -2.1132  0.0000 0.0964
   185  21886.   0.3150  -2.0951  0.0000 0.0979
   186  22004.   0.3150  -2.0953  0.0000 0.0978
```

`U/N` is a summation of the average energy per particle, and `MOV` is the proportion of moves which have been accepted. It indicates that the acceptance rate for this code is around 10%. Ideally, the parallel version should have a similar acceptance rate.

Chapter 4

Parallelisation Strategy

There were several possible ways to distribute the data in parallel. Spatial decomposition, where the data is split up by position in space, replicated data, where each processor has a copy of all the data and works on a given subset, and decomposing the data by particle number, where each processor has an equal number of particles assigned to it. The latter would ensure that the processors were load-balanced, but would lose spatial locality of particles - this would slow things down considerably, as the neighbours of a particle could be on any processor. Too much communication would be necessary. Replicated data is the simplest method, as each processor has access to all the data. However, it is limited by the amount of memory available to each processor and does not scale to large systems. Spatial decomposition does not ensure load-balancing, but it preserves the spatial locality of the data, which is important for this model. Accordingly, it was decided to decompose the data spatially.

Possible methods to parallelise the Monte Carlo steps are discussed in [7]. It is possible to generate a composite move, moving all the particles at once, then accepting or rejecting the entire move. Alternatively, particles can still be moved individually, but extra structures must be put in place for this to happen. Two neighbouring processors cannot be allowed to both move particles next to each other at the same time. If this is allowed, the potential energy must be communicated after each move. Instead, a checkerboard system is set up - each processor is divided into subdomains. Processors will all move particles within the same subdomain at the same time - these moves are far enough apart to be independent of each other. For a three-dimensional system, there must be a minimum of eight subdomains.

This second version requires an adjustment in how the particle to be moved is chosen. Currently, a particle number is chosen at random and then that particle is moved. This cannot be done any more, because each processor has access only to a random subset of the particles. Additionally, it may only move particles in the correct subdomain. It was considered that a cell within the subdomain could be randomly chosen, then a particle within that cell also be randomly chosen. This can be done as the number of cells in the subdomain and number of particles in each cell is known. However, this is not a suitable method as it may introduce a geometrical bias into the system. There is more chance of picking a specific particle in a sparser area. A second strategy would be to give the particles in each subdomain a local number as well as their global one, and randomly choose from these numbers. There are complications when particles move between subdomains, however, and renumbering is necessary.

To begin with, it was decided that the composite move strategy would be used. If the results were not satisfactory, a more complicated method would be looked into.

Each processor will have a cuboidal segment of the system. There will need to be one-cell haloes around all surfaces. Only one cell is needed because only nearest neighbour interactions are required. The haloes will also ensure that the periodic boundary conditions remain for the outer edges.

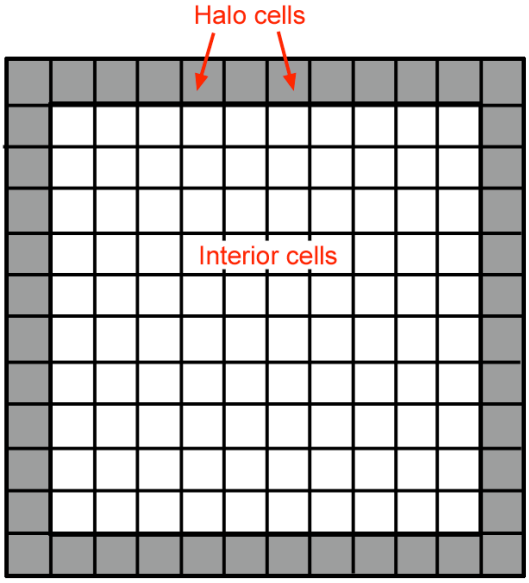

Figure 4.1: A 2D slice through the cell structure of a single processor, showing haloes.

4.1 Communication

MPI [14] is used for the communication. It is a library specification for message-passing which is widely available and has been implemented on many different platforms. It can be used on both shared memory and distributed memory machines.

The communication between steps is slightly more complicated than a single halo-swap. First, it is necessary to communicate how many atoms are in each of the cells, followed by their data. Each processor will maintain its own cell neighbour list for the cells belonging to it. It will be necessary to communicate the system potential, and the number of accepted moves along with number of total moves. A global communication is necessary after each composite move, in order to work out whether to accept or reject.

4.2 Output

To begin with, each processor will write in turn to the output file. This will be done by processor number, with the other processors blocking while one is writing. This removes the need to deal with parallel I/O for the initial versions of the code. The I/O is relatively infrequent so it should not have a great impact on the rest of the code.

4.3 Parallel Random Number Generation

Random number generation in parallel is complicated by the fact that long-range correlations in random numbers in a serial program may become short-range and thus detectable in a parallel program using a serial generator. The code currently uses a random number generator developed by Marsaglia and Zamani [11] with modifications by James [12]. It uses the same two integer seeds each time, meaning the same sequence of random numbers are generated each time the program is run.

There exist specialised parallel random number generators - SPRNG [8] is a well-known example, and details of how to use it along with description of the generation methods it implements can be found in [10] and [9]. It would be possible to use it for random number generation instead. Alternatively, the current generator could be used, but each processor given a different seed. Some thought would have to be put into the generation of these seeds, as they should also be the same between runs of the program.

For the parallel code, it was decided that `SEED1` for each processor would be the same, but `SEED2` would be altered by adding the processor's rank to it. This is an easy way to give each processor a different stream of random numbers. It is possible that a more complicated method would give better distribution of results, but this way is suitable to begin with.

Chapter 5

Parallel Code

In the parallel code, some changing of structure has been done in order to make it easier to work on. The different subroutines have been placed in separate files, and `mcljcell.inc` have been removed. Modules are used instead to make sure relevant subroutines have access to the data structures they should.

The parallel code is all in its own module, called `parallel`. It is done this way so it can be easily replaced with another parallel implementation without having to change the other code files - subroutines with the same names will be called, just their implementations will differ. The module called `variables` contains what used to be in common blocks, along with the subroutines for growing arrays.

5.1 Main Routine

The main routine is as follows:

```
! parallel initialisation
call parallelinit()

! start includes first halo swap and potential calc.
call start()

do iblock = 1 , NBLOCK
    do istep = 1 , NSTEP
! store current position data in "old" arrays
      rxold = RX
      ryold = RY
      rzold = RZ
      maskold = mask

! take each non-halo cell and move each particle in it
      do x = 2, xlength - 1
```

```
            do y = 2, ylength - 1
              do z = 2, zlength - 1
                cellno = cellxyz(x, y, z)
                npart = npart(cellno)
                do i = 1, npart
                  particle = ipart(cellno, npart)
    ! check leaving arrays are not full - if they are, enlarge them
                  call growleavingarrays()
                  call move(particle)
                end do

              end do
            end do
          end do

          call haloswap()
          call calcpot()
          call accept()
          if (mod(istep, NCALC) .eq. 0) call AVCALC
        end do
        call output()
        call safety()
    end do

    ! clean up
    call finalise()
    stop
end
```

Parallel initialisation is done, followed by the program's own initialisation routine. The `iblock` and `istep` loops have the same meaning as previously. The middle section is the new Monte Carlo step, followed by a haloswap, calculation of the total potential and acceptance or rejection. Finally, the clean up routine is called, which deallocates any remaining arrays and calls `MPI_FINALIZE`. The subroutines are explained in more detail below.

5.2 Parallel Initialisation

In this routine, any necessary initialisation for the form of parallel programming used is done. Each processor is also given its rank, and the total number of processors is discovered. The 3D topology of the processors is created, and each calculates what its neighbours in the six directions are, as well as its own coordinates. The routine is shown here:

```
subroutine parallelinit

    comm = MPI_COMM_WORLD
    call MPI_INIT(errcode)
```

```
    call MPI_COMM_RANK(comm, rank, errcode)
    call MPI_COMM_SIZE(comm, nprocs, errcode)

    ! create 3D periodic cartesian topology, with nprocs nodes.
    call MPI_DIMS_CREATE(nprocs, NDIMMS, dimms, errcode)
    call MPI_CART_CREATE(comm, NDIMMS, dimms, periods, .false.,
    &                       comm_cart, errcode)

    ! get ranks of neighbours in all 6 directions.
    ! arguments are (comm, direction, disp, rank_source, rank_dest, err)
    call MPI_CART_SHIFT(comm_cart, 0, 1, rankup, rankdown, errcode)
    call MPI_CART_SHIFT(comm_cart, 1, 1, rankleft, rankright, errcode)
    call MPI_CART_SHIFT(comm_cart, 2, 1, rankforward, rankback, errcode)
    ! get own coords in grid *
    call MPI_CART_COORDS(comm_cart, rank, NDIMMS, coords, errcode)

    ! get number of processors in x, y, z directions
    xprocs = dimms(0)
    yprocs = dimms(1)
    zprocs = dimms(2)

end subroutine parallelinit
```

5.3 Start

This routine ensures the configuration files are read in, arrays and data structures set up, and each processor is assigned its own dataset. The files are read by processor 0, which then broadcasts them in chunks to the other processors. The chunk size is adjustable - smaller input files can be sent all at once, larger ones are broken up to ensure that a processor does not have to be able to hold all the data in memory at any one time. The `.CNF` particle position file reading is shown below (`NTOT` is the total number of particles in the system - it has previously been broadcast to all processors, along with the input parameters). Local sizes and coordinates specific to each processor have also been calculated.

```
    if (rank == 0)
      j = 1
      do i = 1 , NTOT
         read(21,*) rbuffer(j), rbuffer(j+1), rbuffer(j+2)
         j = j + 3
    ! if the buffer is full, or the last particle has been read in
        if (mod(i,BUFFERSIZE) == 0 .or. i == NTOT) then
    ! broadcast chunk to all processors
          call broadcastchunk(rbuffer, 0)
          j = 1
        end if
      end do
```

```
  close(unit=21)
else
! the other processors must call the broadcast routine to receive
  do i = 1 , NTOT
    if (mod(i,BUFFERSIZE) == 0 .or. i == NTOT) then
      call broadcastchunk(rbuffer, 0)
    end if
  end do
end if
! end processor 0 section
```

An MPI broadcast requires all processors to call the routine, which is why the above is split into two sections, the one for rank 0 and the one for the receiving processors. The broadcast routine itself is shown below:

```
subroutine broadcastchunk(rbuffer, sender)

    real, dimension(:), intent(inout) :: rbuffer
    integer, intent(in) :: sender
    integer :: rsize

    rsize = size(rbuffer)
    ! broadcast the buffer data, from processor "sender" to all others
    call MPI_BCAST(rbuffer, rsize, MPI_REAL, sender, comm_cart)

    ! then copy out own data into rx, ry, rz
```

On receiving a chunk, the processor goes through it to find any data which belongs to it. The data is broadcast in this way because the file with the particle position data is not written in such a way that the first n lines belong to one processor, and the next n lines to the next. (Indeed, with the number of processors being variable, it cannot be written in such a way). Instead, they are intermingled so have to be searched through by each processor.

Each processor then sets up its own cells and cell list. The q vectors (used in the Fourier calculation) would be split up in equal chunks among processors, processor 0 getting the first chunk and so on, if the structure factor was being calculated. The processors would initialise their own q vectors. Now that each processor has its data laid out, the first halo swap has to be done.

```
    ! set up cells, including halo cells
    ncell  = 0
    do xcell = 1 , xlength+2
        do ycell = 1 , ylength+2
            do zcell = 1 , zlength+2
                ncell = ncell + 1
                cellxyz(xcell,ycell,zcell) = ncell
                npart(ncell) = 0
            end do
```

```
    end do
end do

! populate cells
do i = 1 , N
! identify cell
    xcell = int((rx(i)-startx)/lengthx *(xlength+2)) + 1
    ycell = int((ry(i)-starty)/lengthy *(ylength+2)) + 1
    zcell = int((rz(i)-startz)/lengthz *(zlength+2)) + 1
    cell(i) = cellxyz(xcell,ycell,zcell)
    celli = cell(i)
! increment counter
    npart(celli) = npart(celli) + 1
    if( npart(celli) .GT. NPARTMAX ) stop 'NPARTMAX error'
    ipart(celli,npart(celli)) = i
end do

SUM = 0
do xcell = 1 , xlength+2
    do ycell = 1 , ylength+2
        do zcell = 1 , zlength+2
            SUM = SUM + npart(cellxyz(xcell,ycell,zcell))
        end do
    end do
end do
if( SUM .NE. N ) stop 'N cell error'
```

The `xcell`, `ycell` and `zcell` lengths run from 1 to `xlength`+2 because they include the halo cells. The non-halo cells are those from 2 to `xlength`+1. `xlength` is the length in cells of the x dimension of the non-halo region. This is not necessarily the same for all processors - ideally, the number of cells in each direction is divisible equally by the number of processors, but the code can deal with non-optimal cases. They do result in load imbalance, however, as the processors with more cells contain more particles.

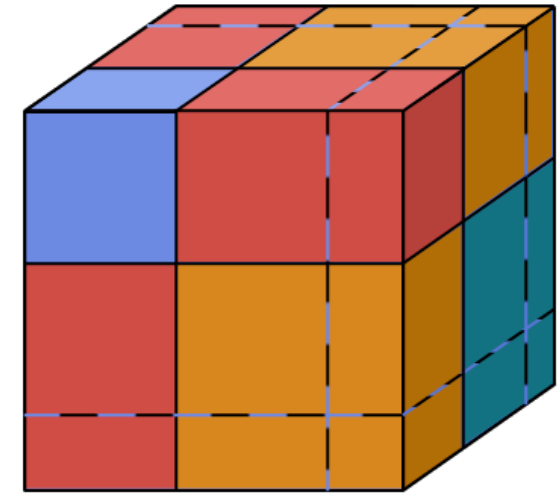

Figure 5.1: If the number of cells is not divisible equally by the number of processors, the extra cells are given to the highest-ranked processor in each direction. This ensures that adjacent halo sizes match up.

5.4 Halo Swapping

To do the halo swap, it is necessary to first know how much data is going to be sent to each neighbouring processor. In order to find this out, the total number of particles in the cells being swapped is counted. This is sent to the relevant neighbour. The next message contains the data itself - the positions of the particles being sent are put into one long 1D array.

The sends and receives now take place. Plimpton's Scheme [15] is used for the halo swapping as it takes care of corner cells while keeping the number of communications down. Without using this scheme, 26 communications from a given processor would be necessary in order to swap data between it and all its neighbours. Plimpton's Scheme limits this to a left-right communication, an up-down communication and a forward-back communication. However, it is necessary to wait for each of these communications before starting the next, as the corner data is passed by sending slightly larger domains each time, the second and third communications including data which has just been received.

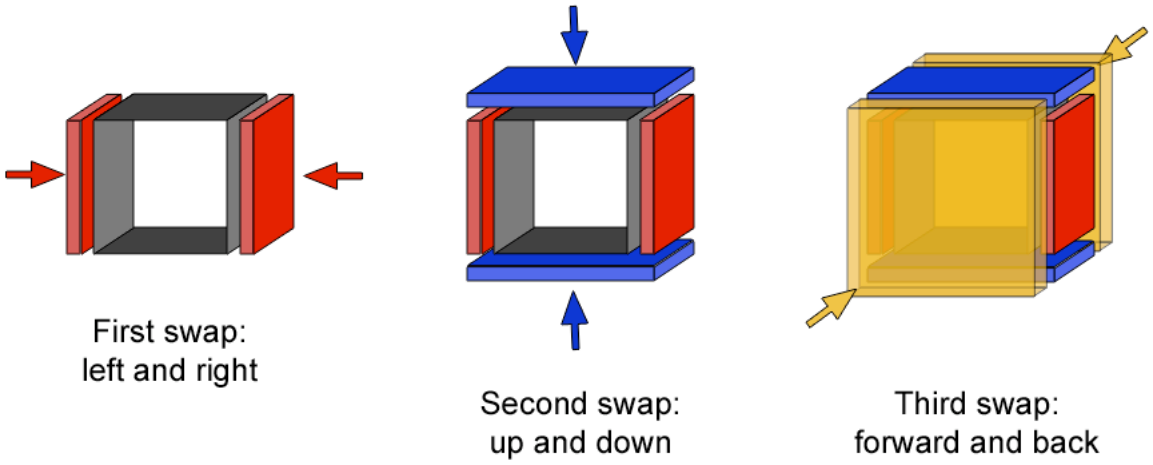

Figure 5.2: Plimpton's Scheme, diagram based on [16]. A left-right swap is done first, followed by an up-down swap that includes some of the data from the first swap. Finally there is a forward-back swap, including data from the previous two swaps.

In most of the halo swaps in the code, the amount of data sent is slightly more than mentioned above. There are particles which move off one processor and on to another. These are no longer in the halo cells on the new processor, instead they are part of its own data. To deal with this, the first message contains two pieces of data: how many particles are being sent in the halo, and how many are being sent that moved to the new processor. In the second message, the moving particles are sent in the same array, but after the halo particles.

For the very first halo swap, there are no particles moving, so the second number is zero. Parts of the haloswap code are shown below.

```
! the particle numbers currently in the haloes have to be added
! to the leaving array
call updateleaving()

! fill left and right arrays with halo data to be sent
    do xcell = 2, xlength+1
        do zcell = 2, zlength+1
            ycell = 2
            cellno = cellxyz(xcell, ycell, zcell)
            nparts = npart(cellno)
            do ipart = 1, nparts
                particle = ipart(cellno, nparts)
                left(llen+1) = rx(particle)
                left(llen+2) = ry(particle)
                left(llen+3) = rz(particle)
                llen = llen+3
            end do
```

```
                ycell = ylength+1
                cellno = cellxyz(xcell, ycell, zcell)
                nparts = npart(cellno)
                do ipart = 1, nparts
                    particle = ipart(cellno, nparts)
                    right(rlen+1) = rx(particle)
                    right(rlen+2) = ry(particle)
                    right(rlen+3) = rz(particle)
                    rlen = rlen+3
                end do
            end do
        end do

    ! send left, receive from right
    ! first, send sizes of left and leavingleft
        sizes = (/ llen, lllen /)
        allocate(sendbuffer(llen+lllen))
    ! non-blocking send, blocking receive
        call MPI_ISSEND(sizes, size(sizes), MPI_INTEGER, rankleft, tag,
                & comm_cart, requestsize, errcode)
        call MPI_RECV(recvsizes, size(sizes), MPI_INTEGER, rankright, tag,
                & comm_cart, status, errcode)
        allocate(recvbuffer(recvsizes(1)+recvsizes(2)))

    ! copy the data from left and leavingleft into sendbuffer
        sendbuffer(1:llen) = left
        sendbuffer(llen+1:lllen) = leavingleft

    ! now send and receive the data
        call MPI_ISSEND(sendbuffer, llen+lllen, MPI_REAL, rankleft, tag,
                & comm_cart, requestleft, errcode)
        call MPI_RECV(recvbuffer, size(recvbuffer), MPI_REAL, rankright,
                & tag, comm_cart, status, errcode)

    ! wait for sends to complete
        call MPI_WAIT(requestsize, status, errcode)
        call MPI_WAIT(requestleft, status, errcode)
        deallocate(sendbuffer)

    ! copy received data into rx, ry, rz arrays
    ! (may need to grow ipart. rx, ry, rz and mask)
        call addparticles(recvbuffer)

        numreceived = numreceived + size(recvbuffer)/3
        deallocate(recvbuffer)

    ! repeat for other directions
```

```
! update number of particles on processor
    numsent = (llen + rlen + ulen + dlen + flen + blen)/3
    numleaving = (lllen + lrlen + lulen + ldlen + lflen + lblen)/3
    N = N - numsent - numleaving + numreceived

! update mask for r arrays
    do i = 1, nleaving
        mask(leaving(i)) = .false.
    end do
```

The data to be sent is put into arrays - `left` to be sent left, `right` to be sent right, etc. The first communication is to send the size of `left` and the size of `leavingleft` (which contains the particles that are moving to the neighbouring processor). Similarly, the sizes being sent from the right are received. Now that the sizes are known, the receive buffer can be allocated. The sendbuffer is then filled with the data from `left` and `leavingleft`, and this is sent. The data from the right is received. It is then necessary to wait for the sends to complete, as they are non-blocking (the receives are blocking). The received data is then copied back into the `rx`, `ry` and `rz` arrays inside the `addparticles` routine. This routine also updates the cell structure with the new particles, and gives them particle numbers. This is repeated for each of the other directions, the next of which is right. The up and down sends include halo data from the left-right swap, as shown in Plimpton's Scheme. The forward and back sends then include data from both previous sends. Finally, the number of particles on the processor is updated, and the mask for missing particle numbers updated. (This mask is needed any time the `rx`, `ry`, `rz` arrays are iterated through directly).

5.5 Calculating the Potential

In the serial code, the potential is first calculated using an all-pairs method, then using the cell neighbour list. The first method is used to check the error in the second. For the parallel version, it is not practical to do an all-pairs calculation. Algorithms have been developed for N-body problems which could be used [13], but they require some extra structures to be created in the code. Generally, N-body calculations in parallel should only be done if unavoidable, which is not the case here.

Only the cell list version of calculating the potential is done in parallel. This uses only particles in neighbouring cells for the calculation. Since the halo cells exist, each processor can fully calculate the potential for its particles. The only communication needed is an all-reduce to add each processor's result together.

```
! cell-list version
POT = 0.0
do i = 1 , N
! find cell
celli = cell(i)
! loop over neighbouring cells
    do neighbour = 1 , NNEIGHBOUR
    cellj = nextcell(celli,neighbour)
```

```
    ! loop over particles in neighbouring cells
        if (npart(cellj) .GT. 0) then
            do k = 1 , npart(cellj)
                j = ipart(cellj,K)
                if (j .NE. i) then
                    RXIJ = rx(i) - rx(j)
                    RYIJ = ry(i) - ry(j)
                    RZIJ = rz(i) - rz(j)
                    if (abs(RXIJ) .GT. 0.5) RXIJ=RXIJ-anint(RXIJ)
                    if (abs(RYIJ) .GT. 0.5) RYIJ=RYIJ-anint(RYIJ)
                    if (abs(RZIJ) .GT. 0.5) RZIJ=RZIJ-anint(RZIJ)
                    RXIJ  = RXIJ*L
                    RYIJ  = RYIJ*L
                    RZIJ  = RZIJ*L
                    RIJSQ = RXIJ**2 + RYIJ**2 + RZIJ**2
    ! Lennard-Jones potential
                    if (RIJSQ .LE. RCSQ) then
                        POT = POT + 4.0*(1.0/RIJSQ**6-1.0/RIJSQ**3) - POTRC
                    end if
                end if
            end do
        end if
        end do
    end do
    POT = 0.5*POT
    energycell = POT / real(N)

    ! call all-reduce on energy to find total
    oldtotal = totalenergy
    call allreduce(energycell)
```

The all-reduce itself looks like the below, and returns an array full of the answers to each processor. The array then has to be summed:

```
subroutine allreduce(real var)

    real, intent(in) :: var
    real, dimension(nprocs), intent(inout) :: totalarray

    call MPI_ALLREDUCE(var, totalarray, 1, MPI_REAL, MPI_SUM, comm_cart,
                                  & errcode)
    ! sum totalarray for answer
    totalenergy = sum(totalarray)

end subroutine
```

5.6 Move

In the move routine, new random coordinates are generated for the particle being moved. In the serial code, they could move the particle anywhere in the system space. This has had to be limited to the space covered by the current processor and its halo cells, so it can move at most one processor-width in any cardinal direction. If the particle moves onto a neighbouring processor, it is put in the appropriate `leaving` array to be sent in the halo swap. The bounds to generate the new coordinate between were calculated in `initcoords()`, back in the `start()` subroutine.

```
celliold = cell(i)

! new (random) coordinates on this processor (inc. haloes)
rx(i)   = rannum(startx, endx)
ry(i)   = rannum(starty, endy)
rz(i)   = rannum(startz, endz)
! fix random under/overflow
if (rx(i) > 0.5)  rx(i) = 0.5
if (rx(i) < -0.5) rx(i) = -0.5
if (ry(i) > 0.5)  ry(i) = 0.5
if (ry(i) < -0.5) ry(i) = -0.5
if (rz(i) > 0.5)  rz(i) = 0.5
if (rz(i) < -0.5) rz(i) = -0.5

xcell = int((rx(i)-startx)/lengthx *(xlength+2)) + 1
ycell = int((ry(i)-starty)/lengthy *(ylength+2)) + 1
zcell = int((rz(i)-startz)/lengthz *(zlength+2)) + 1
cell(i) = cellxyz(xcell, ycell, zcell)

! cell updates
if( celliold .ne. cell(i) ) then
! update new cell data *
    npart(cell(i)) = npart(cell(i)) + 1
    ipart(cell(i), npart(cell(i))) = i
! update old cell data
    do k = 1 , npart(celliold)
        if( ipart(celliold, k) == i ) then
            ipart(celliold, k) = ipart(celliold, npart(celliold))
        end if
    end do
    npart(celliold) = npart(celliold) - 1
end if

! case where particle is moving off processor
if (xcell == 1) then
    leavingup(lulen+1) = rx(i)
    leavingup(lulen+2) = ry(i)
    leavingup(lulen+3) = rz(i)
```

```
        lulen = lulen + 3
        nleaving = nleaving + 1
        leaving(nleaving) = i
    ! repeat for other directions
```

In order to know what range to generate random numbers in, and how to find which cell the moved particle is now in, each processor knows its start and end coordinates in space. The original code uses a coordinate system which ranges from -0.5 to 0.5. The parallel code keeps these as the size of the entire system, with each processor containing a fragment. The diagram shows how the coordinates would be split up in a 2x2x2 arrangement of processors, excluding the halo cells.

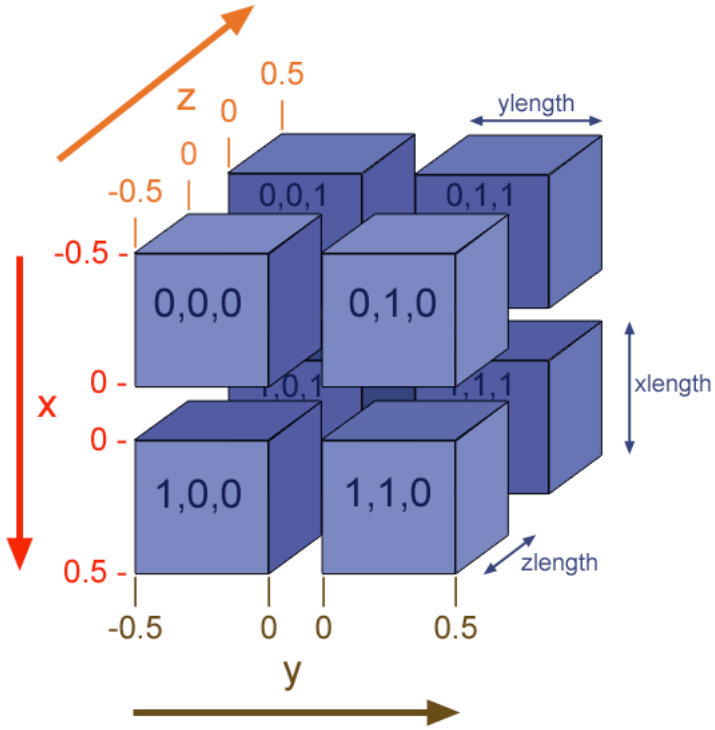

Figure 5.3: The spatial coordinate system across processors, also showing the processors' topology coordinates: (0,0,0) etc. These are the coordinate ranges for each processor's own data - the halo cells mean that there is an extra cell-length in each direction, overlapping with the coordinates of adjacent processors.

5.7 Acceptance

Acceptance is based on the total energy change after the proposed composite move (with the involvement of a small random factor), as described in section 3.1.4. This has not been changed for the parallel version. If the change is accepted, the code continues with the particles in their new positions. If it is rejected, the `rx`, `ry` and `rz` position arrays are reset to their pre-move values from `rxold`, `ryold` and `rzold`. The cell data is then rebuilt, putting it in the same state as it was before the move. The below

has to be calculated on one processor and then "true" or "false" for the move is broadcast. It cannot be calculated on all processors because of the random number generation. The `accept()` routine is shown below:

```
! have attempted to move all the particles
bmovnrm = bmovnrm + NTOT

! acceptance conditions
! calculate on one processor because of the random number generated
if (rank == 0) then
    delpot = totalenergy - oldtotal
! energy change
    betadelpot = delpot/TEMP
! Boltzmann factor
    if (betadelpot < 0) then
        boltz = 1.0
    else if (betadelpot > 75.0) then
        boltz = 0.0
    else
        boltz = exp(-betadelpot)
    end if
! accept if pass
    if (rannum(0.0, 1.0) < boltz) then
        result = 1
    else result = 0
    end if
end if

! broadcast result
call broadcastint(result, 0)

if (result == 1)
! have moved all particles. Update move count *
    bmov = bmov + NTOT
else
! not accepted
    call revert()
    totalenergy = oldenergy
end if
```

The `revert()` subroutine is where the recreation of the old position arrays and cell data takes place.

5.8 RDF and S(q)

These two sections were not parallelised in the course of the project. They had a lower priority as they were not vital to the running of the program, so it was decided to prioritise the rest of the code.

5.9 Output

Data output is done simply, with each processor writing out its data in turn and waiting while the others write. Since the RDF and S(q) are not being calculated, their output files are not being written. The block average data just needs to be aggregated.

```
  ! block averages
    BRHO = BRHO / BLKNRM
    BU   = BU   / BLKNRM
    BMOV = BMOV / BMOVNRM

    ! output to unit 10
    write(10,2040) iblock,  etime(sec),
     :                 BRHO,    BU,
     :                 DRMAX,   BMOV

2040    format(    2X,         I4,1X,       F7.0,
     :             1X,       F8.4,1X,       F8.4,
     :             1X,       F7.4,1X,       F6.4)

    call flush(10)

    ! displacement parameters
    if (BMOV .GT. 0.5) then
        DRMAX = 1.05*DRMAX
    else
        DRMAX = 0.95*DRMAX
    end if
    ! make sure DRMAX does not get too big
    if (DRMAX .GT. RHO**(-1.0/3.0)) DRMAX = RHO**(-1.0/3.0)

  ! zero block counters
    call blkzero()

  ! calculate energy
    call calcpot()
```

Chapter 6

Conclusion

The parallel code supplies the framework to allow the simulation to be carried out on a large number of processors at once. This allows the number of particles in the simulation to be increased manyfold. Indeed, when running on a large number of processors, it is likely that more particles will provide better scaling, as otherwise the code will spend most of its time communicating rather than doing computation. Unfortunately it was not possible to produce real data to confirm this, as the code is at a stage where it implements all the conceptually necessary work, but still requires an estimated week of testing before it will be in a correctly operating state.

During design and development, there were a number of issues which arose. The first was how to choose the particle to be moved, as it was noted that this could not be done in the same random way as the serial code. Looking into ways to choose a particle and move it without necessitating communication after every move prompted the decision to adopt the composite move strategy. Previously there had been attempts to keep the original model of a single particle move followed by acceptance or rejection, but this became increasingly complicated and ran the risk of introducing a bias into the simulation. Following this, there was research into generating random numbers in parallel programs. It was decided to go with a simple model, keeping the serial code's generator and slightly altering one of the seeds by including the processor's rank. This ensured that all processors have their own separate stream of random numbers, and also allows for reproducibility of results. Using seeds so close together may introduce some correlation between the streams, but indepth statistical analysis would have to be done to check this. It was decided this model was sufficient for demonstration.

The next issue was the halo swapping and being able to reset the system after a rejected move. It was necessary to work out exactly which data to send in the swap and how to send it, plus dealing with the fact that the amount of data to be sent would not be known beforehand. This required a reasonable amount of array structure to be created, with the requirement that all the arrays had to be resizeable if necessary. Local particle numbering for particles moving between processors also had to be arranged. Plimpton's Scheme was then researched to make sure that all the data was correctly swapped.

The last major issue was the coordinate system used by the original code, and how that related to the new particle positions being generated. Each processor calculated its own physical coordinates, keeping the range for the entire system the same as the serial code. The possible new coordinates for a particle being moved were then limited to the current processor plus its haloes. If moving into the halo cells, it was sent off-processor in the halo swap. In the original code, the particle could move anywhere in the

whole space.

The finished design incorporates all the above, and allows the code to be run on an arbitrary number of processors, though for optimum results there should be a cubic number of processors (2^3, 3^3 etc) and the number of processors in each direction should equally divide the number of cells.

Appendix A

Risk Analysis

The following are the risks that were identified at the start of the project.

1. There may be difficulties in getting the original code working.

Strategy: Consult with Philip Camp, as he is the author of the serial code.

2. The parallelisation strategy may be more complicated than was originally thought.

Strategy: Ensure that it is considered in sufficient detail during the design stage.

3. The parallelisation may introduce bugs to the program.

Strategy: Have a verification strategy to allow for correctness checking.

4. Time may be lost through illness or other unforseeable circumstances.

Strategy: Make sure some extra time is built in to the timeline.

5. Writeup time may become pushed back if other sections overrun.

Strategy: Write up as much as possible during the whole project.

6. The computers being used may have downtime.

Strategy: Keep backups of the code in different places, keep the code portable so that it can be run on other machines if necessary.

In practice, risk 1 was significantly higher than thought for most of the project. This is explained in the Progress Analysis section. Risk 3 is unquantified, for further testing was necessary. The acceptance rate and average energy could be compared, however. Risk 5 occurred, but was not a problem due to producing several draft reports during the course of the project. Risk 6 did not occur in any way that affected progress.

Appendix B

Timeline

Table B.1 is the original project timeline, set out at the start of the project. Table B.2 is the timeline of how the project progressed.

Week Beginning	Events	Stage
May 15 May 22 May 29	 Presentation	Background Research
Jun 5 Jun 12 Jun 19 Jun 26	 Interim Report Draft to Supervisors Interim Report	Design
Jul 3 Jul 10 Jul 17 Jul 24 Jul 31		Implementation and Testing
Aug 7 Aug 14 Aug 21	 Dissertation Due	Writeup

Table B.1: Original project timeline

Week Beginning	Events	Stage
May 15 May 22 May 29	 Presentation	Background Research
Jun 5 Jun 12 Jun 19 Jun 26	 Interim Report Draft to Supervisors Interim Report	Design and Research
Jul 3 Jul 10 Jul 17 Jul 24 Jul 31	 2nd: Pseudocode review	Implementation and Testing
Aug 7 Aug 14 Aug 21	7th: Draft 3 14th: Draft 4 21st: Draft 5, 23rd: Draft 6, 25th: Dissertation Due	Implementation and Writeup

Table B.2: Actual project timeline

B.0.1 Progress Analysis

The background research section was completed on time, although there was some more research necessary during the design stage for implementation-detail areas such as the random number generation. The background research covered the physics and the Monte Carlo process. The implementation stage took longer than expected, because the specifics of implementing the design proved to require more thinking time. Halfway through the last allotted week for implementation, there was mostly still pseudocode. Coding continued to take place throughout the first two weeks of August, along with writing up. By the end of the second week of August, the code is conceptually complete - ie. all its routines are written, but it requires some more work to make it compile.

There may be some problems in the supplied serial code: after catching two cases of dealing with infinity, the code runs up to block 186 of 250, but then core dumps with an arithmetic exception. Unfortunately this was not discovered immediately, as the output was not checked thoroughly enough and so it was only noticed towards the end of the implementation stage. Finding the source of this was unsuccessful. It was attempted to narrow it down by running a version of the serial code that did not calculate the RDF or S(q) - this was compiled with the -C flag to enable array bounds checking. Unfortunately, this brought up a problem at the very beginning of the code. In the `poti(i)` routine, which calculates the potential of the current particle, a non-existent cell was being referenced. In the time available, it was not possible to track down why the error occurred. The parallel code may not have this last problem, as it does not calculate single-particle potentials, only the potential of the whole system.

Referring to the identified risks, it should have been ensured nearer the start of the project that the serial code did in fact run flawlessly. Working more consistently in the implementation stage would have prevented it overrunning quite so much - one week saved would have allowed the time to make the parallel code runnable.

Bibliography

[1] M.P. Allen and D.J. Tildesley, *Computer Simulation of Liquids*, Oxford University Press, 1989.

[2] Lomond, Sun Fire 15K, Edinburgh Parallel Computing Centre, University of Edinburgh, `http://www.epcc.ed.ac.uk/computing/services/sun/documents/hpc-intro/html/University_Edinburgh_HPC_Se.html`

[3] Wei Shi and J. Karl Johnson, *Histogram reweighting and finite-size scaling study of the Lennard-Jones fluids*, Fluid Phase Equilibria, Volumes 187-188, 15 September 2001, Pages 171-191.

[4] Marjolein Dijkstra and Robert Evans, *A simulation study of the decay of the pair correlation function in simple fluids*, The Journal of Chemical Physics - January 15, 2000 - Volume 112, Issue 3, pp. 1449-1456.

[5] M.E. Fisher and B. Widom, *Decay of Correlations in Linear Systems*, The Journal of Chemical Physics 50, 3756 (1969).

[6] Jiwen Liu and Erik Luijten, *Rejection-Free Geometric Cluster Algorithm for Complex Fluids*, Phys. Rev. Lett. 92, 035504 (2004).

[7] Grant S. Heffelfinger, *Parallel atomistic simulations*, Computer Physics Communications 128 (2000) 219-237

[8] *Scalable Parallel Random Number Generators Library (SPRNG)*, Florida State University and Accelerated Strategic Computing Initiative (ASCI), `http://sprng.cs.fsu.edu`

[9] Michael Mascagni, David Ceperley and Ashok Srinivasan, *SPRNG: A Scalable Library for Pseudorandom Number Generation*, ACMTMS: ACM Transactions on Mathematical Software, 1998

[10] Scientific Computing at NPACI, *Parallel Random Number Generators*, National Partnership for Advanced Computational Infrastructure archives, Volume 3, Issue 7 - March 31, 1999.

[11] George Marsaglia and Arif Zaman, *Toward a universal random number generator*, Technical Report FSU-SC RI-87-50, Supercomputer and Computations Research Institute, Florida State University, Tallahassee, FL, 32306, 1987.

[12] F. James, *A review of pseudorandom number generators*, Computer Physics Communications, 60: 329-344, North-Holland, 1990.

[13] Guy Blelloch and Girija Narlikar, *A Practical Comparison of N-Body Algorithms*, Parallel Algo-

rithms, Series in Discrete Mathematics and Theoretical Computer Science, Volume 30, 1997.

[14] Steve Otto (editor) et al, *MPI: A message-passing interface standard*, The Message Passing Interface Forum, 1994.

[15] Steve Plimpton, *Fast Parallel Algorithms for Short-Range Molecular Dynamics*, Journal of Computational Physics 117, 1-19 (1995).

[16] F.A. Gilabert, Ramon Garcia-Rojo and Sean McNamara, Figure 4: Snapshot of the communication boundaries using 3 shifts in the Plimpton's scheme, *Parallelization of DEM code for fine powders*, Project HPC-EUROPA (R113-CT-2003-506079), "Structuring the European Research Area" Programme.