Parallel Molecular Dynamics

by

Aidan Tully, Andrew Cleary, and Karolína Benková

Group project report for the course
High Performance Computing for Mathematicians

Supervised by
Prof. Ben Leimkuhler and René Lohmann

Contents

1 Introduction **1**
- 1.1 Hamiltonian dynamical systems . . . 1
- 1.2 The Verlet method . . . 2

2 Serial Case **2**
- 2.1 Assumptions and Initialisation . . . 2
- 2.2 Overview of Serial Algorithm . . . 3
- 2.3 Forces Smoothing and Interpolation . . . 3
- 2.4 Optimised Forces Computation and Periodic Boundary Conditions . . . 4

3 Parallelisation **6**
- 3.1 Assumptions . . . 6
- 3.2 Division into processes . . . 6
- 3.3 Initialisation . . . 6
- 3.4 Boundary enforcement . . . 8
- 3.5 Force calculation . . . 8
  - 3.5.1 Method One . . . 8
  - 3.5.2 Method Two . . . 9

4 Results and parallel performance analysis **10**
- 4.1 Molecular Dynamics Results . . . 10
- 4.2 Parallelisation performance . . . 11
  - 4.2.1 Strong scaling . . . 12
  - 4.2.2 Weak scaling . . . 13
- 4.3 Conclusion . . . 15

A Serial forces algorithm **16**

1 Introduction

One of the most frequent applications for high performance computing is the area of molecular dynamics, a study of movement and interaction of molecules done by solving the classical equations of motion for a set of molecules [1]. The interactions of a collection of atoms can be modelled by using force functions derived from potential energy functions, with the atoms moving according to Newton's laws of motion. Periodic boundary conditions can also be employed to restrict the atoms to lie and move on the surface of a three-dimensional torus [2]. This report describes parallelisation of a molecular dynamics simulation by splitting this domain into subdomains, each taken over by one process. In the rest of this section we briefly introduce the mathematics used in the molecular dynamics model. The serial implementation of this problem is introduced in Section 2, followed by description of its parallelisation in Section 3. Finally, in Section 4 we analyse the performance of the parallel code by doing parallel performance analysis and evaluate our implementation of the problem.

1.1 Hamiltonian dynamical systems

A molecular dynamics problem can be modelled in terms of the trajectories and momenta of the atoms in the system by using Hamilton's equations which are derived from the Hamiltonian for the system. Describing the problem as a system of N atoms in two dimensions and assuming each atom has a mass $m = 1$, the Hamiltonian is

$$H(\mathbf{q}, \mathbf{p}) := \frac{1}{2}\mathbf{p}^\top \mathbf{p} + U(\mathbf{q}), \tag{1}$$

where $\mathbf{q} = (\mathbf{q_1}, \mathbf{q_2}, \ldots, \mathbf{q_N})$ is a $2N$-dimensional vector with the position $\mathbf{q_j} = (q_{j,x}, q_{j,y})$ of the j-th atom. Similarly, $\mathbf{p} = (\mathbf{p_1}, \mathbf{p_2}, \ldots, \mathbf{p_N})$ is a $2N$-dimensional vector with the momentum of the j-th atom being $\mathbf{p_j} = (p_{j,x}, p_{j,y})$ [2]. Since the first term in eq. (1) gives the kinetic energy of the system and the function $U(\mathbf{q})$ is the potential energy function, the Hamiltonian gives the total energy of the system. Time evolution for the system of particles with the Hamiltonian $H = H(\mathbf{q}, \mathbf{p})$ in eq. (1) is described by Hamilton's equations

$$\begin{aligned} \dot{\mathbf{q}} &= \frac{\partial H}{\partial p} = \mathbf{p}, \\ \dot{\mathbf{p}} &= -\frac{\partial H}{\partial q} = -\nabla U(\mathbf{q}) = \mathbf{F}, \end{aligned} \tag{2}$$

with $\mathbf{F}$ being the $2N$-dimensional vector of forces acting on each of the vectors and dots meaning derivatives with respect to time.

For the simulations in this project we chose the commonly used Lennard-Jones (6-12) potential energy function in a simplified parameter setting

$$U_{LJ}(r) = \left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}, \tag{3}$$

with $r = \|\mathbf{q}_i - \mathbf{q}_j\|_2$ being the Euclidean distance between the atoms i and j. Summing over all pairs of atoms we get the total potential energy of the system

$$U(\mathbf{q}) = \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} U_{LJ}(\|\mathbf{q}_i - \mathbf{q}_j\|). \tag{4}$$

The distance that minimises the pair-wise potential is $r_{min} = 2^{1/6}\sigma$ and we will use this value as the distance between the atoms in the horizontal and vertical direction in the initial lattice at the beginning of the simulation. A plot of the Lennard-Jones potential is shown in Figure 1. It consists of a repulsive ($r < r_{min}$) and attractive ($r > r_{min}$) parts, i.e. the atoms are repulsed from each other up to the distance r_{min}.

An important property of the Hamiltonian (1) is energy conservation. An issue with long-term simulations in molecular dynamics can be energy drift when using a numerical method that lacks conservation properties [2]. To avoid this problem we will use the Verlet method as it is a geometric integrator, the scheme is described in the next section. We will also use the plot of the potential and kinetic energy of the system over time for guidance to see if the algorithm is working properly.

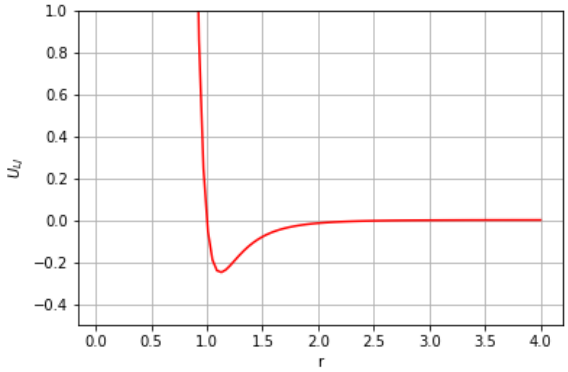


Figure 1: The Lennard-Jones (6-12) energy potential function with $\sigma = 1$.

1.2 The Verlet method

The Verlet method, also known as leapfrog or Störmer-Verlet method, is one of the most recognized and commonly used numerical integration schemes in molecular dynamics. The scheme comprises two half-steps in momenta and a full step in positions,

$$\begin{aligned} \mathbf{p}_{n+\frac{1}{2}} &= \mathbf{p}_n - \tfrac{h}{2}\nabla \mathbf{U}(\mathbf{q}_n), \\ \mathbf{q}_{n+1} &= \mathbf{q}_n + h\,\mathbf{M}^{-1}\mathbf{p}_{n+\frac{1}{2}}, \\ \mathbf{p}_{n+1} &= \mathbf{p}_{n+\frac{1}{2}} - \tfrac{h}{2}\nabla \mathbf{U}(\mathbf{q}_{n+1}), \end{aligned} \tag{5}$$

with $\mathbf{M}$ being the identity matrix due to the mass of each particle $m = 1$. Properties of the Verlet method make it suitable for long-term simulations of Hamiltonian systems. As a geometric integrator, this method is symmetric, reversible, volume-preserving, and symplectic. Its global error in both calculating the trajectory and the energy is of order two, with no significant additional cost compared to the first order Euler's method. Due to its structure, we can reuse the force vector ($\mathbf{F} = -\nabla U(\mathbf{q})$) calculated at step n in the step $n + 1$, therefore we need only one force evaluation per step. As function evaluation usually comes with a large cost, the objective of our parallel code will be to split the force calculations among the processes to achieve a good speed-up.

2 Serial Case

In this section, we give some details about the serial code that we developed. We begin by describing the assumptions that we made, then we give an overview of the algorithm, followed by how we smoothed the forces between two atoms and finally how we optimised the forces computation.

2.1 Assumptions and Initialisation

In reality atoms or molecules can interact over large distances through electromagnetic force interactions (which goes to zero as the separation grows to infinity). As we see in the plot for the Lennard-Jones potential energy function in Figure 1, for longer inter-atomic distances, the interaction between the atoms is almost zero. Thus to avoid calculations of forces between atoms that are too far away, which are negligible, we will introduce a cut-off distance R with a typical value $R = 2.5\sigma$ [1]. The distance R determines the radius of the circle around the atom where the forces with the other atoms inside the circle will be calculated. Such a circle can be seen later in this section in Figure 3 where the orange atom interacts only with the pink and green atom that are inside the circle, and not with the blue one as it is outside of the R radius. We note that the cut-off distance should not be greater than $L/2$ where L is the width of the domain as to avoid an atom 'seeing' itself through the periodic boundary conditions that we introduce below.

In order to to implement our molecular dynamics system, we needed to choose a domain to simulate. To make this realistic we followed the normal approach of simulating a small subsection of a larger material. In this way we do not need to worry about implementing artificial unrealistic boundary conditions on our domain. Instead, we can act as if outside the domain we simulate there is another 'identical' domain representing another subsection of the entire material we are within. In this manner, by leaving our domain in any direction we would arrive in another identical domain of the same size. This can now be easily implemented in our model as periodic boundaries which physically implies a continuous material but in our system acts as if we simulate the dynamics on a torus.

Before we can start our simulation we need to distribute our atoms around our domain. We chose to do this in a square lattice with separation δ equal to the minimiser of the Lennard-Jones potential energy function ($\delta = r_{min} = 2^{1/6}\sigma$). This configuration is an unstable equilibrium of the system so by initalising with this formation and a very slight momentum (to release ourselves from the equilibrium), we will hopefully evolve to a position of stable equilibrium. To ensure this lattice is compatible with our periodic boundaries, and realistically resembles a smaller section of a larger material we chose to tie the domain size to the number of atoms so the entire domain is filled on initialisation. This condition can be changed to have a different initial condition but it is the one used for all test cases in this report.

2.2 Overview of Serial Algorithm

A summary of the algorithm we implemented in the serial code to simulate the molecular dynamics described in the previous sections can be found in Algorithm 1.

Algorithm 1: Overview of the serial algorithm.

```
Initialisation;
Compute the forces for the initial system (Algorithm 2);
for i=1 : Number of time steps do
    Take momentum half-step with Verlet;
    Take position full-step with Verlet;
    Enforce boundary conditions;
    Compute the forces (see Algorithm 2) and potential energy for the new system;
    Take second momentum half-step with Verlet;
    Compute the kinetic energy of the new system;
end
```

To store the atoms in memory easily, we created a C++ `struct` called `Atom`. This data structure consisted of 2 arrays of 2 doubles, one array for position and one for momentum. We made use of the `vector` C++ class to store these atoms and so that we could easily loop over all the atoms in the domain. As we would be dealing with a large number of atoms, each of which consisting of 4 doubles, a nice feature of the `vector` C++ class is that it dynamically allocates the memory in the heap.

2.3 Forces Smoothing and Interpolation

As mentioned in Section 2.1, we assume that atoms further apart than some cut-off distance, R, do not interact. However, as can be seen in Figure 2, the Lennard-Jones force is non-zero at this distance, so we would be introducing a discontinuity into our force function. This is not ideal as it is highly non-physical for atoms to suddenly stop interacting when they reach this separation. Instead, we want our force function to smoothly go to zero at R. Thus, we create a small parameter ϵ to mark the area where the smoothing occurs, and set it to $\epsilon = 0.1$ for this project. In order to smooth our force function, we use a cubic polynomial to smoothly interpolate between the Lennard-Jones force at $R - \epsilon$, $F_{LJ}(R - \epsilon)$ and 0 at R. This gives us four

conditions on the four co-efficients of a cubic polynomial:

$$a(R-\epsilon)^3 + b(R-\epsilon)^2 + c(R-\epsilon) + d = F_{LJ}(R-\epsilon) \tag{6}$$

$$a(R)^3 + b(R)^2 + c(R) + d = 0 \tag{7}$$

$$3a(R-\epsilon)^2 + 2b(R-\epsilon) + c = \frac{dF_{LJ}}{dr}(R-\epsilon) \tag{8}$$

$$3a(R)^2 + 2b(R) + c = 0 \tag{9}$$

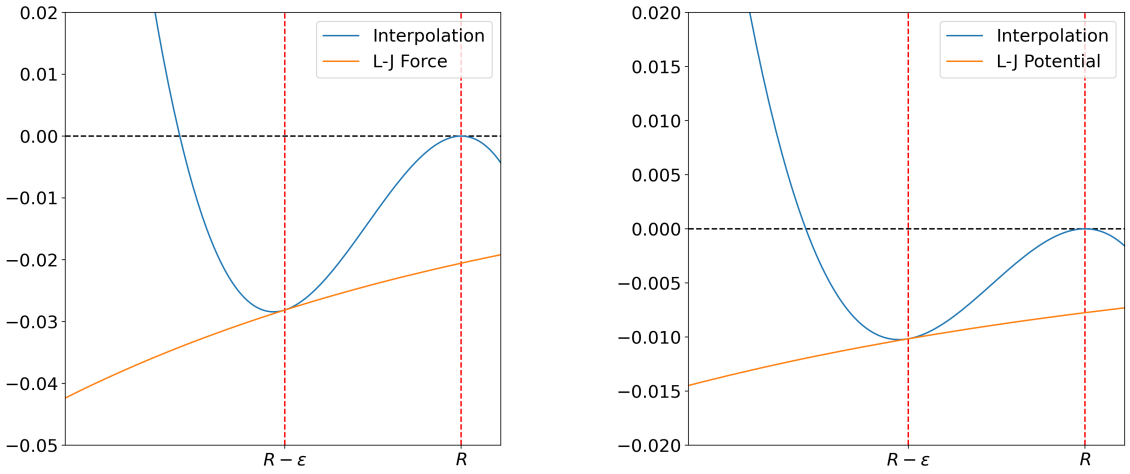
We solved for these co-efficients using Mathematica and the resulting interpolation function $I_F(r)$ can be seen in Figure 2a. The smoothed force function is $\hat{F}_{LJ}$ with values

$$\hat{F}_{LJ} = \begin{cases} F_{LJ}(r), & r < R - \epsilon \\ I_F(r), & R - \epsilon \leq r \leq R \\ 0, & r > R \end{cases}$$

We performed an identical interpolation for the Lennard-Jones potential function, $I_U(r)$, which can be seen in Figure 2b. The resulting values of the potential were then

$$\hat{U}_{LJ} = \begin{cases} U_{LJ}(r), & r < R - \epsilon \\ I_U(r), & R - \epsilon \leq r \leq R \\ 0, & r > R \end{cases}$$

Plots of both of these functions are shown in Figure 2.



(a) Force function $\hat{F}_{LJ}$

(b) Potential energy function $\hat{U}_{LJ}$

Figure 2: Interpolation of the Lennard-Jones a) force and b) potential energy function. For $r < R - \epsilon$, we take the original functions, then for $R - \epsilon \leq r \leq R$, we take the interpolation cubic polynomial $I(r)$, and then for $r > R$, we set the functions to 0.

2.4 Optimised Forces Computation and Periodic Boundary Conditions

We implemented a highly optimised forces computation which exploits the fact that the force that atom a exerts on atom b is equal and opposite to the force that atom b exerts on atom a. Thus, we loop over pairs of atoms, such that we reduce the number of force calculations by a factor of 2.

When we compute the forces between a pair of atoms, we first check if they are closer together than a cut-off radius, R. If they are further apart than R, we implement periodic boundary conditions by creating ghost copies of one of the atoms outside of the domain. We only create these ghost copies if the two atoms are both in a 'skin' of width R, on opposite sides of the domain. This can be viewed as allowing the atoms to interact in the opposite direction i.e. around the torus. Note that for each pair of atoms, we only need to create ghost copies of one of the atoms, by a symmetry argument. We then check if these ghost copies are within the cut-off radius and compute the forces accordingly. We optimised how the algorithm creates ghost atoms outside of the domain to implement the periodic boundary conditions. We do not create ghost atoms needlessly. Since we only care about the single ghost copy within R, there is no point creating a ghost copy that has no chance of being within R. Thus, as adhered to above, we only create a ghost atom if both atoms are in opposite 'skins'. Thus, at a maximum, we will create 3 ghost copies: one in a horizontal direction, one in a vertical direction and one in a diagonal direction (if a ghost atom was created in both the horizontal and vertical directions). A summary of this optimised forces computation can be found in Algorithm 2 in Appendix A, and this optimised ghost atom creation and 'skins' can be seen in Figure 3.

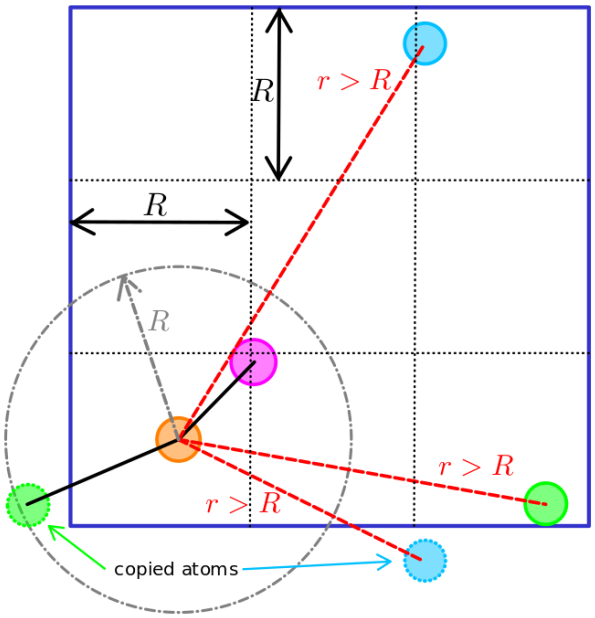


Figure 3: Visualisation of the forces computation algorithm described in Section 2.4. There are 4 atoms in the simulation domain (orange, pink, green and blue), ghost copies of the green and blue atoms were created in the opposite sides of the domain. The skins at the edges of the domain have thickness R and are marked by black dotted lines. The figure shows force calculation between the orange atom and the other atoms in the skin layer of the domain: the pink atom lies within the R radius (grey dashed-dotted circle) of the orange atom so the force is calculated (black solid line), the blue and green atoms were located at longer distance so their copies were created. We can see that the copy of the green atom lies within the R radius, while the copy of the blue one is out of the circle therefore the force is not calculated.

3 Parallelisation

Expanding our serial code to a parallel code comes with a few challenges and a need for additional assumptions which are outlined in Section 3.1. First we needed to devise a method to divide the work between multiple processes. We contemplated having a single controller process designate tasks to other 'worker' processes but decided this was difficult to implement and not very effective therefore this wasn't the method we wished to follow. Instead we decided to divide the domain into smaller subdomains and assign a subdomain to each process as described in Section 3.2. As we are dealing with liquid state molecular dynamics, we would expect our domain to be relatively full of atoms, and so the load balance on each process would be roughly equal in our current setup unlike a gas state which could exhibit clumping. Since we split the domain between the processes, we need to initialise the atoms on all the processes individually and also we need to account for the atoms moving around, especially when they cross the boundaries of the subdomain and enter the subdomain of another process. Implementation of the initialisation can be found in Section 3.3 and implementation of how atoms are crossing the boundaries is explained in Section 3.4. We also show the approach towards the force calculation along with a more effective modification in Section 3.5. We will show results of both force calculation methods in Section 4.

In order to send atoms to neighbouring processors a bit more easily, we created an MPI datatype called `MPI_atom`. This mirrored the structure of the C++ `struct Atom` that we described in Section 2.2, i.e. a contiguous array of 4 doubles. This meant that we did not have to send the position and velocity data individually to neighbouring processes.

3.1 Assumptions

To ensure the parallel version of our code could be implemented and made sense in reality, we needed to enforce a few more restrictions on the problem. In order to ease implementation and ensure an atom won't see further than one neighbour cell away we enforce that the cutoff radius for force interactions R must be smaller than the cell widths M (i.e $R < M$). Ideally, this would be at least $R < \frac{M}{3}$ to make sense with the skins in the domain which are shown for the parallel case in Section 3.5.1, but the minimum criteria is $R < M$. We also impose that the total processor count must be a square number, to ensure domain subdivision is feasible with square cells. Also related to this we enforce the starting quantity of atoms in the domain to be scaled up based on a square lattice initialised in each cell. So for example with a lattice of size 4 in each cell, for 9 processes would give 144 atoms in total in the domain.

3.2 Division into processes

We divide the main domain (which is a square) into a square number of equally sized ($M \times M$) subdomains, or cells, as shown in Figure 4 for an example with 9 processes. The computation is parallelised by assigning one subdomain to each process. Every process therefore deals with atoms located in this subdomain, force and energy calculation corresponding to these atoms, timestepping with the Verlet method, and sending the atoms to other processes if they cross the boundaries of the cell. In MPI communication, we use the coordinates of the process by computing its row and column in the grid. We number the columns and rows starting from zero, as is shown in Figure 4. Besides the inter-cell communication when calculating the forces and moving the atoms when they leave the cell, the processes can also send the positions and velocities corresponding to their atoms to the root process (rank 0), which is responsible for printing them into .dat files. This option can be toggled on or off based on whether the user wishes to track individual atoms or not. At the end of the simulation all energies are gathered from the individual processes and printed to file also.

3.3 Initialisation

Since we assign one subdomain to each process, we must initialise the atoms with the right coordinates so that when the positions are collected together, the atoms are placed in a lattice shown in Figure 5. We calculate the row and column corresponding to each process and shift the coordinates of the atom by $col \times M$ in the x-direction and $row \times M$ in the y-direction. Similarly to the serial case, the atoms are spaced with distance $\delta = r_{min}$ in both x- and y-direction. With a total of N atoms in the $L \times L$ lattice, there are initially

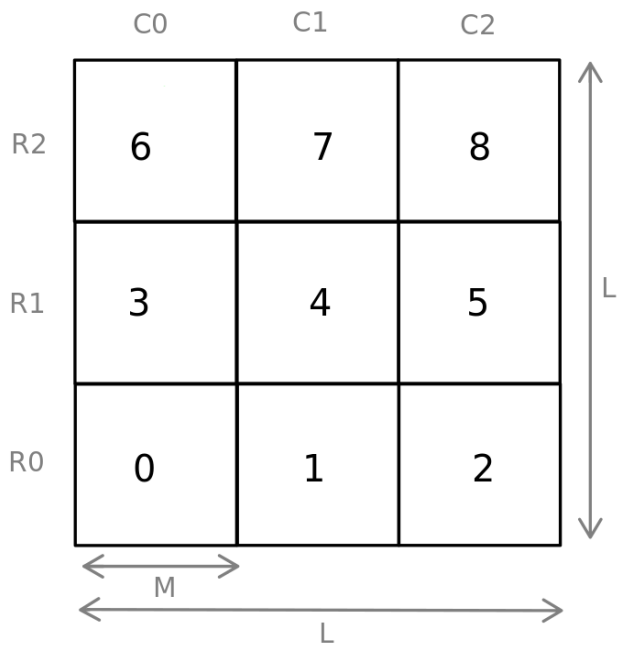


Figure 4: Division of the main domain of length L into smaller subdomains (cells) of length $M = L/\sqrt{P}$ with $P = 9$ processes. We assign one cell to each process starting from the bottom left corner, see the process numbers inside the cells in the figure. The rows are numbered from the bottom and columns from the left with starting index 0.

N/P atoms in each subdomain. The length of the main domain is calculated in the same way as in the serial code.

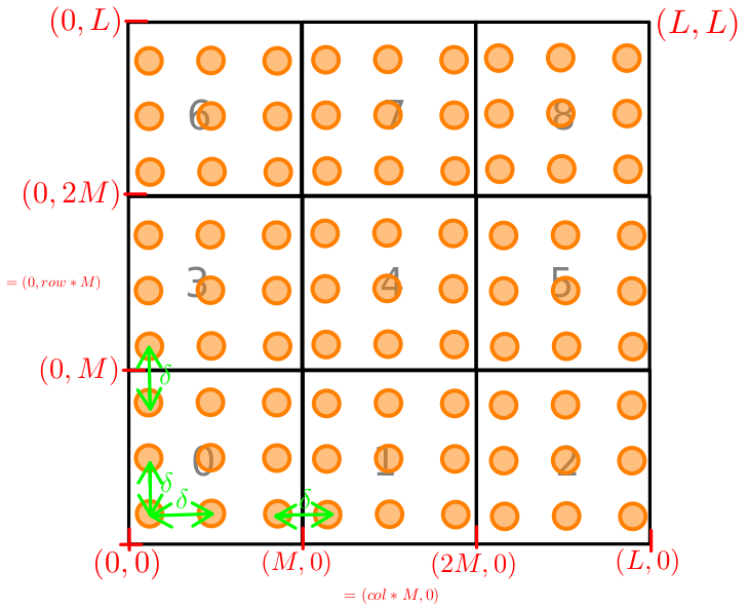


Figure 5: Initialisation of the atoms into a Cartesian lattice in parallel for 9 processes and cell latice of size 3, i.e. $N = 3 \times 3 \times 9 = 81$ atoms in total.

3.4 Boundary enforcement

As well as having to enforce periodic boundary conditions at the edges of the full domain, we now have the added complexity that when an atom moves outside of its processor's subdomain, we have to send it to the corresponding neighbouring processor. We call this 'crossing a processor boundary'. In practice, as we assume we have initialised the atoms in a fluid-like setting, the atoms will always move into one of the 8 neighbouring processors when they cross a processor boundary. Some examples of atoms crossing processor boundaries can be seen in Figure 6.

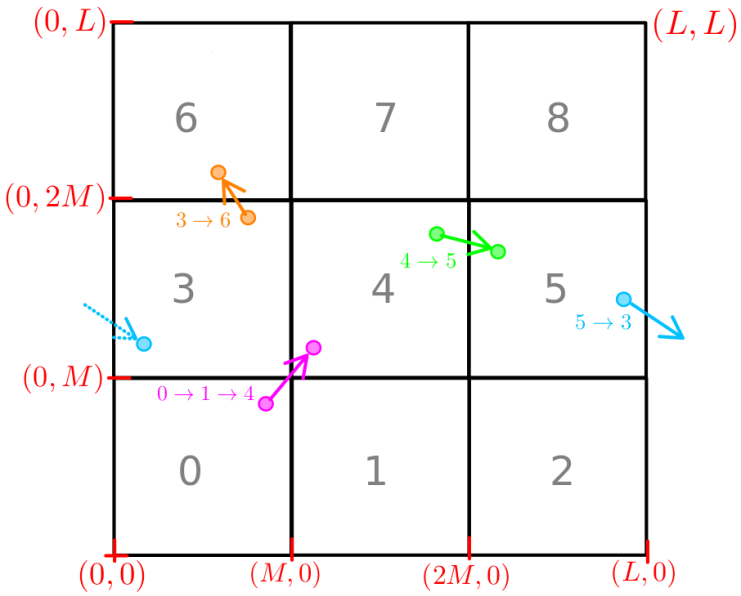


Figure 6: Examples of atoms crossing processor borders, and needing to be sent to neighbouring processors. In particular, note how the pink atom is first sent in a horizontal direction, and then in a vertical direction, which achieves a diagonal sending.

Here, we first run into the complication that the receiving processor has no idea how many atoms it will receive in each direction. The way we overcame this challenge was by first sending how many atoms each process was sending in each direction so that the receiving process had this information. Then, we were able to send the atoms themselves, one at a time, using our `MPI_atom` MPI datatype.

In practice, we counted and sent atoms that crossed borders in a horizontal direction first. The counting was easily done with a simple check of the atoms x position. We then included a `MPI_Barrier` so that every process had finished sending in the horizontal direction. Only then did we count and send the atoms that crossed borders in a vertical direction. By doing the sending this way, we ensured that atoms which moved to a diagonally neighbouring process were sent correctly. An example of this is the pink atom in Figure 6.

3.5 Force calculation

3.5.1 Method One

As mentioned at the start of the section, we need to adapt our method of calculating forces to accommodate for atoms being separated over different processes. Distances between atoms located in the same cell designated to a process can still be computed as usual though. The difficulty arises for atoms still under the cutoff separation R but in neighbouring cells which would be assigned to different processes. Here we need to find a way to communicate any atoms positions within neighbouring cells to the process containing the atoms we wish to calculate a force on. An initial approach to this problem we undertook was to have every neighbour in the vicinity of a process send its entire list of atoms to the process to allow it to compute distances and check that another atom was in range. This method was needlessly expensive as in cases where the cells

are large (with respect to the cutoff i.e. $R < M$), the majority of atoms in a neighbouring cell will never be in range of those within the original process and so a large amount of fruitless distances are calculated. Instead we would like a method that takes only the atoms within neighbouring cells that are even possible to be in range. To do this we introduced the concept of *cell skins* which are an analogy of the skin layers at the domain boundaries in the serial case. Figure 7 illustrates the concept of cell skins, showing that at worst atoms within this outer skin (of width R) of the cell contain the only atoms visible to a neighbour cells atoms. Using these skins, we only need to send atoms that fall within this region in a cell to a neighbouring cell. To do this we implement a simple check to see if an atoms lies within either of the 4 cardinal directions skins, and also then if they lie within one of the overlaps of these skins which are the atoms in range of the diagonal neighbours. The position of these atoms within the cells atom list is noted and then these lists of indices for the cell list are looped over and the relevant atoms sent to the destination processes. Here we send atoms to all 8 neighbouring cells and receive atoms from the skins in these 8 cells also. Once the skins are received, forces are calculated between only the atoms located in the skin and the atoms received from the neighbours skin (so for example the forces for atoms from the left neighbour are only computed for those in the left skin as they are the only possible atoms in range). In this approach all cells calculate forces on their own atoms, so there is a number of times forces will be computed twice needlessly in neighbouring processes which is addressed in the next section. The advantage with this method lies in the simplicity to implement as atoms are just sent to a neighbour for a force computation and special cases do not need to be considered. This method is used mainly as a proof of concept and a first step in our parallelisation, but is still included in the analysis in Section 4.

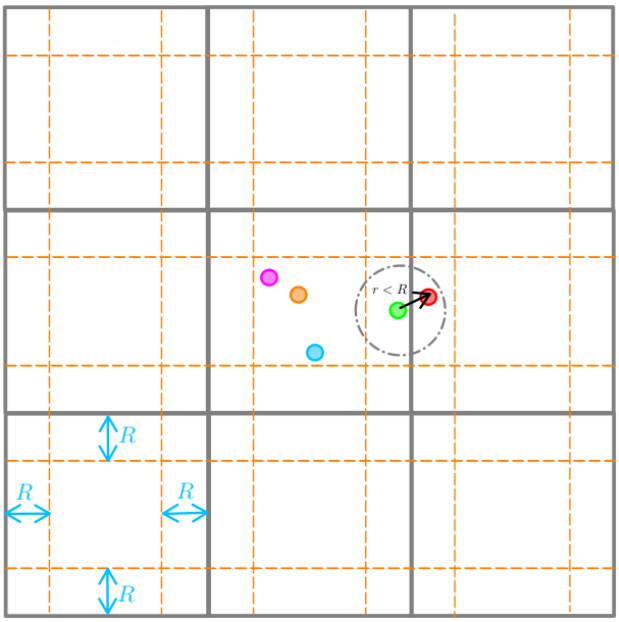


Figure 7: Force calculation in the parallel code (Method One). Each cell has skin layer around its boundary of width R, marked with orange dashed lines. The atoms lying in skin layer of a process are within reach only to skin layer atoms of the neighbouring process, as the skin width is equal to the cut-off radius (grey dashed-dotted circle).

3.5.2 Method Two

In the previous section we described implementation of force computation algorithm where each cell computed forces of the atoms within the cell with atoms in the near skins of the neighbouring cells. However, this way the inter-cell atom interactions were computed twice, i.e. forces and potential energy between skin layer atoms of cells A and B were computed both in cell A and B. By improving the code we used with Method One, we can optimise the computation cost by reducing force calculations by a half, shown in Figure 8,

similarly to what was suggested in [1], in a following way. Instead of sending skin layer atoms (with their positions and momenta) to every neighbour cell, i.e. in 8 directions, we only send them to neighbours in 4 directions (upper left corner, up, upper right corner, right) and receive from the other 4 directions. This way, besides computing intra-cellular atomic forces, each process also computes forces between the atoms in a skin layer and the atoms received from the corresponding direction. For example, in Figure 8, process 1 sends its atoms in the upper skin layer to process 4 (up direction), and process 4 computes forces between its atoms in the bottom skin layer with the atoms received from process 1.

This method is more efficient compared to Method One, as we are sending atoms into 4 directions instead of 8 and we are also calculating forces uniquely. However, since the forces acted on an atom are summed up based on all its interactions with other atoms, the calculated forces need to be sent back to the process that originally sent its skin layer atoms. To avoid hassles with sending and receiving two-dimensional vectors, we collect and send force vectors for the x and y direction separately. Thus, since each force vector we send is of the same length as the number of atoms received and has two dimensions, compared to Method One we send 25% less data and calculate half the forces. Comparison of the speed-ups and efficiency of Methods One and Two can be found in Section 4.

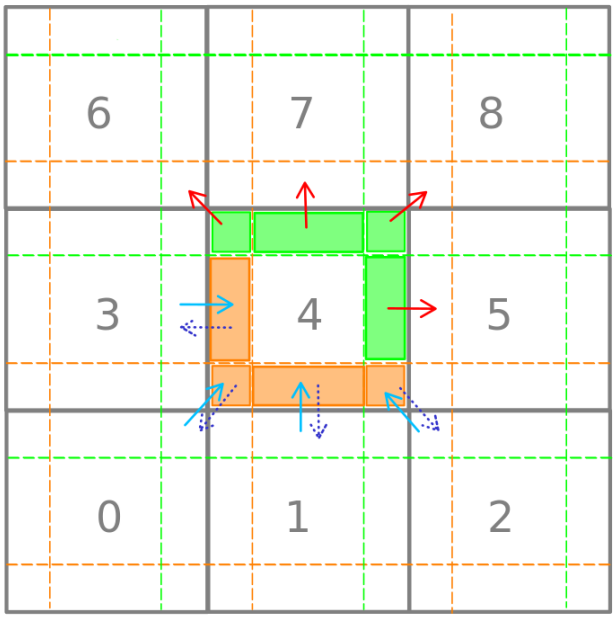
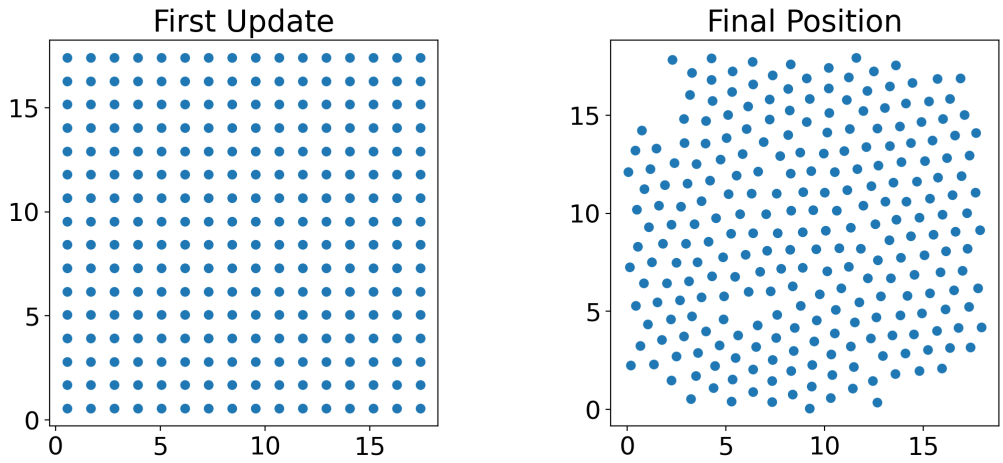


Figure 8: Force calculation in Method Two. The red arrows mark sending list of skin layer atoms lying in the green zones in the corresponding directions (there are 4 skins with corresponding 4 sending directions in the green zone), while blue arrows mark receiving these lists from neighbouring processes. The concept of cell skins stays the same as in Method One, however, the inter-cellular force calculation only happens in the orange zone. Finally, the forces are sent back to where the atoms were received from which is marked by purple dashed arrows. We note that e.g. for sending in the up direction, the atoms in the whole upper skin layer are being sent to the upper process, i.e. the corner zones are included.

4 Results and parallel performance analysis

4.1 Molecular Dynamics Results

In this subsection, we quickly give the results of our simulation, as we wish to dedicate most of this results section to analysing the performance of our parallelisation. For these results, we used a small simulation of just 256 atoms and the results were generated by the second parallelisation method. We note that the results from the first parallelisation method and also the serial code were identical.



(a) Unstable equilibrium

(b) Stable equilibrium

Figure 9: We begin the simulation in an unstable equilibrium, which is the Cartesian lattice (a). After a kick, we see that the state then evolves to its stable equilibrium (b), which is a hexagonal lattice.

In Figure 9, we see that the simulation begins in an unstable equilibrium, which is the Cartesian lattice. It is an equilibrium as every atoms experiences the same force in all four directions, and so they all remain stationary. Thus, to excite the system, we slightly increase the atom velocities towards the centre of the grid, and we see that the state then evolves to its stable equilibrium, which is a hexagonal lattice [2]. We also plot the potential and kinetic energies of the system in Figure 10. Nicely, we see that the potential energy decreases drastically as it tends to its stable equilibrium, almost like a phase transition. The kinetic energy increases symmetrically to conserve total energy.

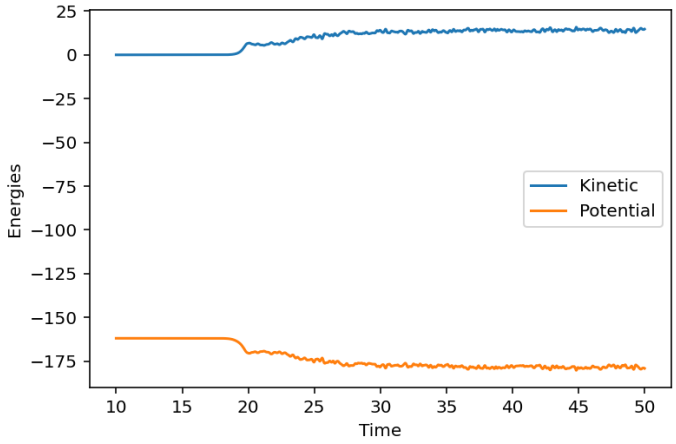


Figure 10: The potential energy decreases sharply at around 20 seconds when it transitions to its stable equilibrium. The kinetic energy increases then to match this and conserve total energy.

4.2 Parallelisation performance

In this subsection we evaluate the performance of the parallel code by comparing its computational times with the serial code with varied number of atoms and processes. We will compute parallel speed-up and

parallel efficiency for fixed number of atoms in order to get strong scaling, and with fixed number of atoms per process to get weak scaling. When measuring time, we will omit writing data into files.

It should also be noted that we ran the following simulations on the University of Edinburgh's supercomputer, Eddie. We used the intel C++ compiler and its accompanying implementation of MPI. In hindsight, it was good to experience a different architecture and a different job submission system compared to Cirrus which we used previously. For jobs using fewer than 40 cores, any number of cores can be requested. However, for jobs using more than 40 cores, Eddie requires that a multiple of 16 cores is used, and it uses 16 core nodes. These jobs are then spread over a number of different 16 core nodes, instead of being run on a single 40 core node. This has consequences on the strong and weak scaling analysis that follows. Firstly, we can only run the simulation using 1, 4, 9, 16, 25, 36, 64, 144, ... as the number of cores has to be a square number and also a multiple of 16 for jobs with more than 40 cores. Secondly, we will see a big decrease in performance in the simulations using 64 and 144 cores, as suddenly the communication overhead will be much larger. This is because the cores now have to communicate across different nodes, instead of just communicating within a single node. Note that in the strong scaling case the 25 cores are used is omitted as including it raised the lowest common multiple of the process sizes significantly (the total number of atoms has to be a square multiple of the number of processes) and for greater flexibility in choice of total atoms it was removed from the tests for the strong scaling.

4.2.1 Strong scaling

We take $N = 2304$ atoms and vary the number of processes P in order to measure the computational time $T(N, P)$. A plot of the computational times vs. the number of processes can be seen in Figure 11. As mentioned in the introduction of this section, we see the expected decrease in performance for 64 and 144 cores. However, up to 36 cores, we see a very nice decrease in computation time, as this is a log-log plot. Also as expected, we see that our second method, described in Section 3.5.2, outperforms our first method, described in Section 3.5.1. This is because we perform half as many force computations between neighbouring cells. Also, although we then send the forces back in 4 directions, this is more than compensated by only having to send the atoms in skins in 4 directions, instead of 8. This means that in total, we only send 4 $2n$-length vectors (x and y forces) and 4 $4n$-length vectors (x and y positions and velocities of the atoms). This is better than 8 $4n$-length vectors (x and y positions and velocities of the atoms). This is a reduction of 8 n-length vectors. Note that here n is the number of atoms in each skin being sent.

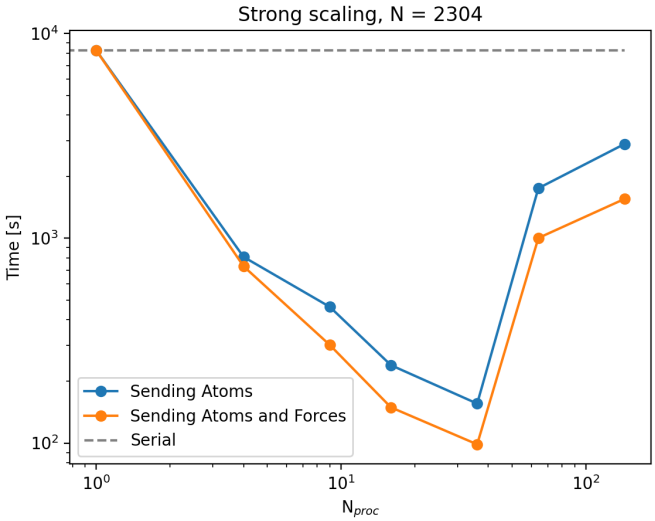


Figure 11: Total computation time vs number of processes.

The parallel speed-up $S(N, P)$ is computed by the ratio

$$S(N, P) = \frac{T(N, 1)}{T(N, P)},$$

where $T(N, 1)$ is the run time of serial code. In the ideal case we would expect the speed-up to increase linearly with the number of processes, i.e. along the line $S = P$, which is often not possible due to the amount of serial code present in the parallel code. However, in Figure 12 we see a super linear speed-up, up to 36 cores. As expected, we again see the decrease in performance when we go to 64 and 144 cores. The literature [1] states that this super linear speed-up can be expected sometimes, so this is not a new phenomenon. The improvement of our second method over our first method is even more noticeable in this speed-up analysis in Figure 12.

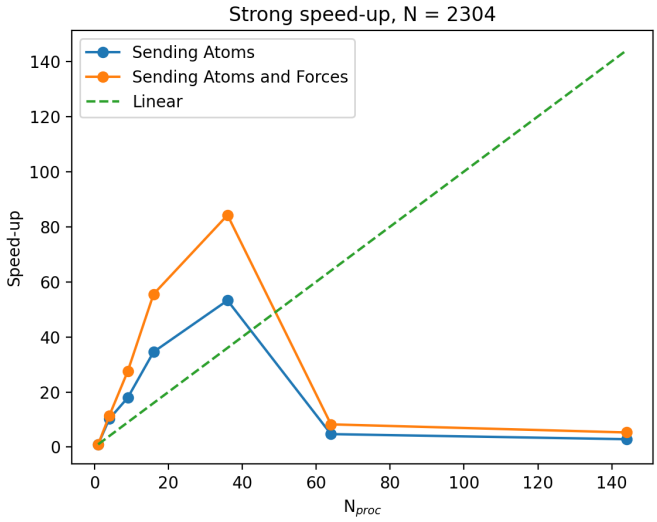


Figure 12: Strong speed-up vs number of processes.

Finally, we compute the parallel efficiency $E(N, P)$ as

$$E(N, P) = \frac{S(N, P)}{P} = \frac{T(N, 1)}{PT(N, P))}.$$

A fully efficient code would reach the efficiency $E = 1$ which would correspond to an ideal case where P processors produce a P-times faster computational time. However, again in Figure 13, we see that our code performs better than this ideal case, up to 36 cores.

We can also use Amdahl's law to extract an estimate for the fraction, α, of the code that is completely serial:

$$E(N, P) = \frac{1}{\alpha P + (1 - \alpha)}$$

We fitted this curve to the parallel efficiency plot for 4, 9, 16 and 36 cores, and a very rough estimate for α for both methods can be seen in Figure 13. We get very small numbers for α for both methods, although α for our second method is nearly 3 times smaller than our first method. This rough estimate for α suggests that our second code is approximately 0.3% serial, which is very good.

4.2.2 Weak scaling

Now we fix the number of atoms per process to be $N_p = 256$, therefore the total number of atoms in the domain is $P \times N_p$ which ranges from just 256 in our serial case to 36,864 for the case of 144 processes. A plot of the computational time vs. number of processes is displayed in Figure 14. As mentioned before, we can

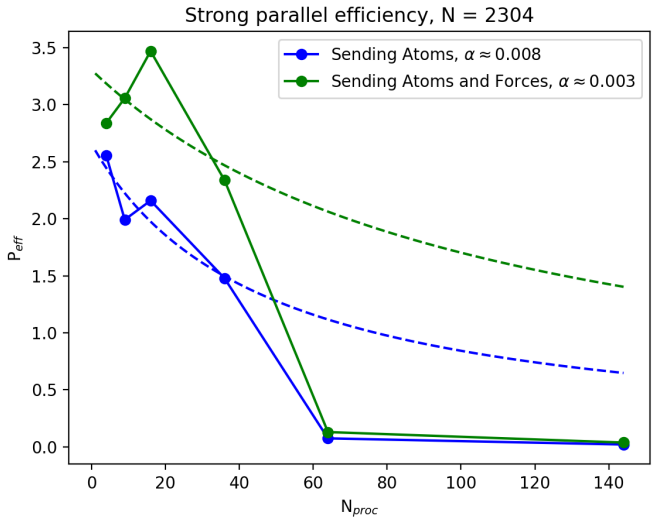


Figure 13: Efficiency vs number of processes.

see a considerable decrease in performance when using 64 or 144 cores due to the inter-node communication in contrast to the communication between cores on a single node. For weak scaling we would ideally hope to see computation time remaining constant as we scale up the simulation. Here we can see for 4, 9, 16, 25 and 36 cores this is roughly the case but as we transition to the larger core counts this fails to be true. The serial code shows slower performance here highlighting the advantages the parallel implementation has over the serial case. Much like with the strong scaling, we see our second version of the parallel implementation out preforming the first across all core counts.

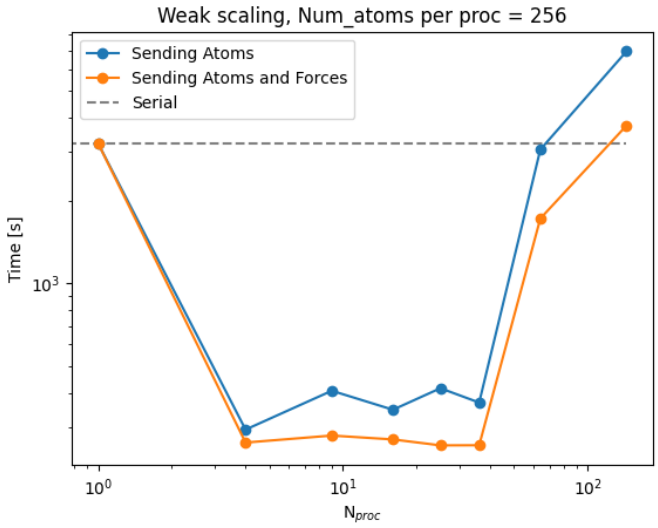


Figure 14: Total computation time vs number of processes.

The next figure shows the parallel efficiency for weak scaling. Note that speedup in terms of weak scaling doesn't really make sense as we are investigating the time taken for an increase in both processes *and* problem size. So while the method to calculate efficiency for a weak scaling is the same as the method to calculate speedup in the strong case, we refer to it as efficiency here as we are interested in the time increase to calculate larger problems.

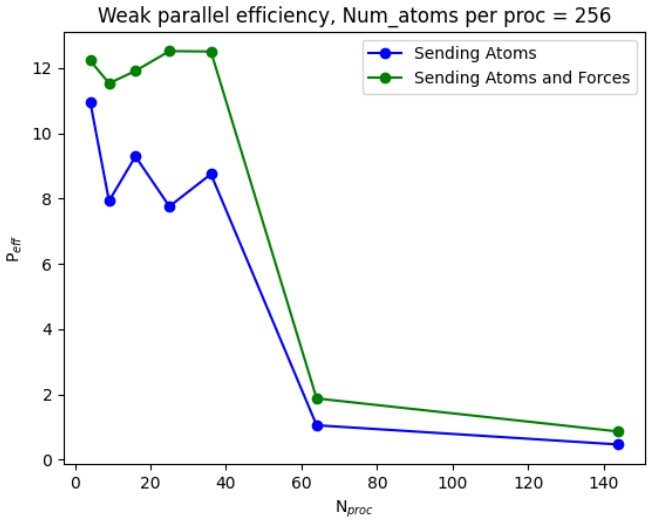


Figure 15: Efficiency vs number of processes.

From this we can see the slight decrease in performance for the parallel systems over cores 4, 9, 16, 25 and 36 as we expect. This highlights the overhead of communication within the parallel setup, as the problem size increases, and increased load is experienced as the number of atoms to communicate at any timestep increases. Once again the very large core counts exhibit a drastic performance decrease as the communication overhead here is now between multiple separate nodes but it continues the trend of decreasing efficiency between the 64 and 144 core cases. The large base efficiency rate for the small core counts is due to the underlying inefficiency within the serial case, showing the advantages that can come from a parallel implementation also. Even with the large communication penalty in the larger core counts, the efficiency is still very close to 1.

An interesting comparison between the new and old method can be seen on both of these plots also. In the original method of sending just atoms, we see a big dip in efficiency on odd square numbers. We suspect that even squares performed better than their odd counterparts due to the combination of staggering the communication between even and odd columns and the volume of sending occurring in the old method compared to the improved method. For example starting with sending right in the $P = 25$ case, with 5 cells per row, cells positioned 1st, 3rd and 5th in a row all send atoms out and wait to for them to be received by their right side neighbour before subsequently receiving atoms from their left neighbours. The cells in the 2nd and 4th position receive first from the left and then send to their neighbours on the right. But as this is a periodic domain the 1st and 5th cell are neighbours and so the 5th cell will not receive from the 4th until the 1st receives the atoms it sent. But the 1st cell is not receiving immediately and so the 5th must wait until the 1st is finished sending before it will then ask to receive atoms. The increased volume of sending causes the delay caused from the temporary deadlock over the boundaries of odd squares to be more apparent in the older method than in the improved method.

4.3 Conclusion

Simulations of molecular dynamics have been implemented in both a serial and parallel manner, with two separate methods in the parallel case shown within this report. The performance of the parallel implementations are both much better than the serial case with the second version of the parallel code consistently performing the best overall, with an estimate of 0.3% for the proportion of the code which is serial. The impact of communication speed is also seen when large process counts are examined as the communication is then performed between compute nodes rather than the fast communication within individual nodes. The results of a simulation are also presented to demonstrate the implementation does indeed perform as it should.

References

[1] M. P. Allen and D. J. Tildesley, *Computer simulation of liquids*. Clarendon Press, 1989, ISBN 978-0-19-855645-9. DOI: 10.1007/978-3-319-16375-8.

[2] B. Leimkuhler and C. Matthews, *Molecular Dynamics with deterministic and stochastic numerical methods*, ser. Interdisciplinary Applied Mathematics. Springer, 2015, ISBN 978-3-319-16375-8. DOI: 10.1007/978-3-319-16375-8.

A Serial forces algorithm

Algorithm 2: Overview of the forces computation in the serial algorithm.

```
for i=1 : Number of pairs of Atoms do
    Compute the separation between both atoms, r;
    if r < R then
        Compute the force between both atoms;
    else
        bool horizontal = False;
        bool vertical = True;
        if Atom a ∈ left skin and Atom b ∈ right skin then
            horizontal = True;
            Create a copy of Atom b with x → x + L;
            Compute the separation, r, between Atom a and the copy of Atom b;
            if r < R then
                Compute the force between both atoms;
        if Atom a ∈ right skin and Atom b ∈ left skin then
            horizontal = True;
            Create a copy of Atom b with x → x − L;
            Compute the separation, r, between Atom a and the copy of Atom b;
            if r < R then
                Compute the force between both atoms;
        if Atom a ∈ top skin and Atom b ∈ bottom skin then
            vertical = True;
            Create a copy of Atom b with y → y + L;
            Compute the separation, r, between Atom a and the copy of Atom b;
            if r < R then
                Compute the force between both atoms;
        if Atom a ∈ bottom skin and Atom b ∈ top skin then
            vertical = True;
            Create a copy of Atom b with y → y − L;
            Compute the separation, r, between Atom a and the copy of Atom b;
            if r < R then
                Compute the force between both atoms;
        if horizontal and vertical are both true then
            Create a copy of Atom b, combining the above changes in x and y;
            Compute the separation, r, between Atom a and the copy of Atom b;
            if r < R then
                Compute the force between both atoms;
    end
end
```