Efficient Algorithms for Molecular Dynamics Simulations and Other Dynamic Spatial Join Queries

Author:

Andrew Noske

Department of Information Technology

James Cook University, Cairns Campus

andrew.noske@jcu.edu.au

Supervisors:

Dr Dmitry Konovalov

Dr Jason Holdsworth

Dissertation submitted by Andrew Noske in partial fulfilment of the requirements for the Degree of Bachelor of Information Technology with Honours in the Department of Information Technology at James Cook University.

Date of submission:

21/11/2004

Declaration

I declare that this thesis is my own work and has not been submitted in any form for another degree or diploma at any university or other institute of tertiary education. Information derived from the published and unpublished work of others has been acknowledged in the text and a list of references is given.

Andrew Noske

Andrew Noske

____

21/11/2004

Abstract

This thesis investigates several methods of optimising molecular dynamics simulations using a cutoff radius. The verlet neighbour list technique, combined with a fixed grid to execute range searches, is accepted as the most efficient method for performing such simulations, however building a list of neighbour atoms within the cutoff radius of each other takes the bulk of processor time. Optimising this self-spatial join query problem is critical to improving simulation speed, and the primary focus of this thesis. Some proposed techniques, such as selective checking of verlet neighbours and using half range searches to execute self-spatial join queries, produced good performance improvements. Several other techniques, including space-filling curves and use of minimum bounding rectangles, yielded poorer than expected results. The thesis also compares and evaluates the traditional cell list, a minimum cell list and an atom list technique. It is argued that the atom list is superior. Furthermore, the optimal number of cells per side for the fixed grid and optimal verlet radius for the verlet neighbour list are also analysed. Simple algorithms for dynamically finding the optimal number of cells per side and optimal verlet radius are tested. This thesis is valuable as a guide to implementing and optimising molecular dynamics simulations, but also relevant to those investigating dynamic self-spatial join queries or range queries in "real world" vector space.

Keywords: Molecular dynamics fluid simulations, fixed grid, cell list, verlet neighbour list, spatial join, range search, space-filling curves.

Acknowledgements

Firstly, I would like to thank my two supervisors, Dr. Dmitry Konovalov, who supplied me with this project idea, and Dr. Jason Holdsworth, who provided helpful reviews of my thesis and whom few people realise is a brilliant musician/composer. I would also like to extend my gratitude to Dr. Bruce Litow for scientific reading and writing skills develop through CP5080. Thanks also to all members of our very friendly School of Information Technology (Dr. Phillip Musumeci, Dr. Chris Gaskett, Dr. Jason Holdsworth, Dr. Eoin Hyden, Mrs Marion Hooper and Mr Colin Lemmon) and my fellow honours/MIT students (Eugene McArdle, Joshua Bennett, Tahawar Durrani and Jon Roberts) for offering helpful critical analysis during CP5090, and helping us develop a deeper understanding of scientific process. Both these subjects were invaluable in preparing us for our projects.

I also wish to give thanks to two special friends of mine; Mrs Emmy Kerrigan, who always offered me encouragement, and our secretary Miss Amber Davison, who somehow managed to keep me organised all year and (apprehensively) cleared my pay checks. Thank you to all my friends who have helped make this year memorable. Finally, I would like to thank Dr. Phillip Musumeci, Dr. Chris Gaskett, Dr. Jason Holdsworth and Susan Barclay for their helpful review of my thesis. It hasn't been easy balancing tutoring, subject work and a thesis, but I've had lots of understanding from people. Surely, without that support, waiting all those hours for simulations to finish running on my machine would have driven me insane.

Table of Contents

List of Figures

List of Tables

List of Definitions

1 Introduction

1.1 Background and Motivation

The spatial join query has become a well-known computing problem with numerous applications across a broad range of fields including geographic information systems, computer graphics, databases, astronomy and bioinformatics [7].

Self-spatial join: given a set of points, retrieve all unique pairs of points, such that the distance between the points is less than or equal to some fixed radius r_s.

Some practical examples of this proximity problem and other forms of range search queries in vector space include air traffic control, moving wireless devices, computer games, numerous types of queries on topological data and various types of particle simulations [25].

Molecular dynamics simulations, like so many other forms of computer simulation, have started to play a valuable role in modern science and now form a broad and significant field of research [2, 11, 8]. Without recent high-speed computers, testing theoretical models, simulating, visualising and running tests on millions of interacting particles would not be possible. There are many forms of molecular dynamics simulations [11]. One of the most common of these is a simulation of liquid which involves finding all pairs of particles (neighbours) within some cutoff radius, calculating the forces between them, and moving them forward a single timestep [2]. This process is repeated over many minute timesteps, and results are recorded. The process of finding all neighbours (a self-spatial join query) is computationally expensive and such simulations must be run for substantial periods of time to obtain meaningful results. This thesis focuses primarily on practical ways to improve the processing time of existing techniques for this specific molecular dynamics simulation problem.

For testing in this thesis a bulk liquid was simulated using a periodic boundary condition [2]. Using a periodic condition all atoms and forces exist inside a fixed box, but can wrap around the edges of the box. In addition, a Lennard-Jones pair potential interaction model [11] was used to simulate the movement of atoms. The most efficient known techniques to perform this type of simulation are the verlet neighbour list [32] and use of a fixed grid data structure to execute range searches. In the verlet neighbour list technique, a radius greater than the cutoff radius is used to build a neighbours list [2]. Between rebuilds, this neighbour list is updated so as to isolate only those neighbours within the cutoff radius of each other. Furthermore, a fixed grid partitions a cubic box containing all atoms into a grid with a fixed number of equal sized cells per side and is the optimal structure for proximity problems in a system where particles are evenly distributed. This thesis builds on the use of both these techniques.

Although all results and discussion are based on the fixed grid, many of the techniques discussed could be applied to other multidimensional access methods, for example grid files [23], quad-trees [28] and R-trees [14] which are often used in N-body simulation problems and range search problems where the distribution of points or objects is highly skewed. This thesis is a useful guide to anyone implementing molecular dynamics simulations; however, it should also be useful to readers in any field investigating dynamic range queries in Euclidean space.

Furthermore, the practical value of this project can be appreciated within the context of the Towards Molecular Structure Kinetics (TOMSK) project coordinated by Dr Dmitry Konovalov [19]. The eventual aim of TOMSK is to simulate vast numbers of molecules interacting and protein folding. Results from this thesis have been used to code and construct an efficient engine layer and set of generic classes [24] which will be used by future students in this project.

1.2 Thesis Objective

Obtaining a list of all neighbours each timestep takes the bulk of processor time in molecular dynamics simulation and therefore represents a significant problem. The main goal of this thesis was to optimise the computation time of executing this dynamic self-spatial join problem in main memory. Computation of pair-wise interaction is another process which often lends itself to optimisation, but this is dependent on the type of interaction model used, and therefore considered outside the scope of this thesis. The objective was instead to propose and investigate several innovative techniques to speed up the execution of self-spatial joins over a fixed grid and also speed up the process of updating of verlet neighbour lists. The success of some well known existing techniques, such as the use of space-filling curves and cell lists, was also investigated. The final objective was to analyse the performance of varying the verlet radius used to build a verlet neighbour list and number of cells per side in the fixed grid, so that algorithms to find optimal values for these could be tested.

1.3 Summary of Contribution

The most successful technique this thesis proposes is the half range search technique, whereby the neighbours for each atom can be found by searching half the volume necessary for an ordinary range search. The thesis also demonstrates that the commonly used cell list, whereby each range search encompasses only adjacent cells, is not always the most efficient technique and has significant disadvantages over a minimum cell list. Moreover, it is found that a cubic atom list is a more intelligent choice, because parameters can be changed dynamically with minimal detriment to performance.

The thesis also finds that the verlet neighbour list technique can be improved by selective checking of neighbours using the displacement of atoms, and investigates two algorithms to find the optimal values of the verlet radius and optimal number of cells per side. Not all techniques were as successful. For example, the use of sub-girds and minimum bounding rectangles inside cells both had limited success, and in some proposed techniques even worsened performance. However, even these findings are helpful, and this thesis and code provide a useful guideline for anyone wishing to implement or optimise similar simulation problems.

1.4 Thesis Structure

The remainder of this thesis is organised in several chapters. Chapter 2 is a brief literature review and outlines the various molecular dynamics simulation methods used in testing. Chapter 3 proposes a handful of techniques to optimise spatial joins over a fixed grid. Chapter 4 briefly describes how tests were designed and implemented. Chapter 5 presents and discusses all results, including the testing of all techniques proposed in Chapter 4. Finally, Chapter 6 concludes the thesis with suggestions on future research directions and a summary of findings.

2 Background and Literature Review

This chapter introduces molecular dynamics simulations and some of the most popular known models and techniques used to implement these simulations. The sections in this chapter are logically ordered such that each builds on preceding sections. Section 2.1 introduces molecular dynamics simulations, Sections 2.2 to 2.9 review commonly known algorithms used in these simulations, and Sections 2.10 and 2.11 introduce existing concepts that can be applied to these simulations.

2.1 Introduction to Molecular Dynamics

Computer simulations of molecular systems will never be able to perfectly model the real world. At the atomic level, particles obey complex quantum laws; however there are a number of statistical ensembles which closely approximate real particle behaviour using classical laws. *Molecular dynamics (MD)* is a computer simulation technique in which the time evolution of interacting atoms is followed by integrating their equations of motion. That is, at intervals of some timestep, the forces between all pairs of atoms are calculated and accumulated, and then each atom is moved.

In a typical molecular dynamics simulation a number of statistical analysis functions are also performed at certain intervals. For instance, temperature, pressure and potential/kinetic energy drifts are commonly calculated and recorded after each timestep [2, 11]. This is important, because such results, averaged over large timeframes, can be compared to measured results of real simulations of liquids, whereas it would be impossible for any real experiment to provide detailed information about individual atoms. Calculation of these functions falls outside the scope of this thesis and were omitted during all tests. For the purposes of testing, the Lennard-Jones pair potential model was used to simulate the movement of atoms in a bulk liquid.

2.2 Interaction Model: Lennard-Jones Potential

The *Lennard-Jones pair potential* model is the most commonly used *interaction model* in molecular dynamics [8], and is described by the following equation:

$$\o_{LJ}(r) = 4\varepsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right]$$

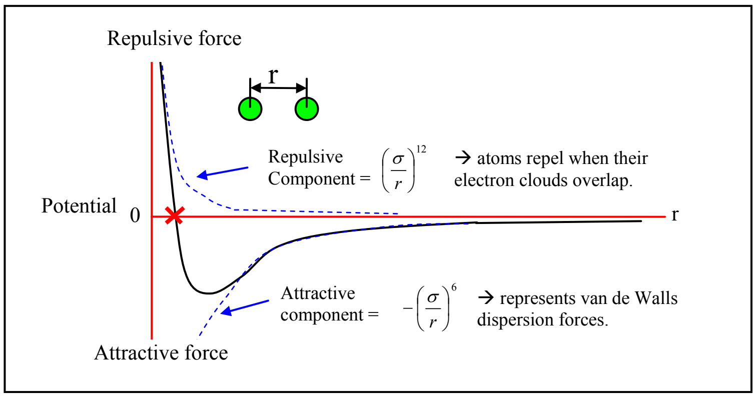


Figure 2.1. Lennard-Jones potential vs. separating distance.

The parameter *r* represents the distance between the centres of two particles. Figure 2.1 shows that if particles are far away from each other the force is negligible. Closer particles have weak attraction, but if they get too close they repel with exponentially increasing force. The parameters ε (characteristic distance) and σ (characteristic energy) are chosen to fit the physical properties of the simulated material/particles, but it is customary to work in a system of units where ε=1 and σ=1 [8].

2.3 *Periodic Boundary Condition*

Simulating bulk matter is difficult, because most molecular simulations can only handle so many thousands, perhaps millions, of molecules. Using these quantities is only enough to simulate a small liquid droplet or microcrystal; whereby molecules on surface boundaries have fewer neighbours and experience different forces from molecules further inside.

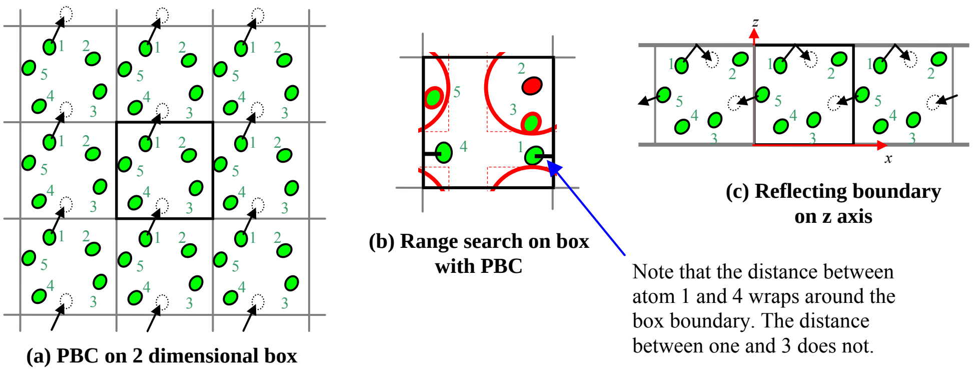

(a) PBC on 2 dimensional box
(b) Range search on box with PBC
(c) Reflecting boundary on z axis
z
x
1
2
3
4
5
Note that the distance between atom 1 and 4 wraps around the box boundary. The distance between one and 3 does not.

Figure 2.2. Periodic boundary condition.

The *periodic boundary condition (PBC)* is a method to simulate bulk matter and eliminate surface effects. Using PBC, a cubic box is replicated throughout space to form an infinite lattice. Forces and moving particles effectively wrap around the boundaries, so that a particle which leaves one face will enter through the opposite face as shown in Figure 2.2.a. There are a few variations on PBC [2], and in some experiments (Figure 2.2.c) this condition is removed along certain axes [8], but the method illustrated in Figure 2.2.a is the most common, and is the method used in this thesis.

Note that if PBC is used, the distance from point *i* to *j* along each dimension should be defined as the closest distance whereby the distance may or may not wrap around the box boundary (Figure 2.2.b). This can be computed as follows:

$$distNoWrap = j_x - i_x$$

$$distIfUsingPBC = \begin{cases} distNoWrap - boxLen, & if\ distNoWrap > (boxLen/2) \\ distNoWrap + boxLen, & if\ distNoWrap < -(boxLen/2) \\ distNoWrap, & if\ -(boxLen/2) \le distNoWrap \le (boxLen/2) \end{cases}$$

2.4 N-body Solutions

A molecular dynamics simulation is a classic N-body problem. The classical *N-body problem* simulates the evolution of a system of *N* bodies, whereby force is exerted on each body due to its interaction with all the other bodies in the system [5]. N-body problems are quite specialised, and many papers propose quite specialised algorithms to solve such problems. To compare all particles to all other particles (the *brute force approach*) is not scalable. However if a set of nearby particles is far enough away from an individual particle, the set can be treated as a single particle with a composite mass, at the centre of the array. This principle gave birth to a number of solutions based on specialised data structures which approximate long range forces. The *O(N log N) Barnes-Hut algorithm* [3], is possibly the simplest of these, and uses a quad-tree structure, as does the *O(N) Fast Multipole Method (FMM)* [13] and *Parallel Multipole Tree Algorithm (PMTA)* [5]. Other solutions, such as the *O(N log N) Particle Mesh Ewald (PME)* algorithm and the *O(N) Multigrid Summation technique (MG)*, are based on grid structures.

Although these approaches are suitable for star simulations, molecules exhibit strong directional forces, and there is no easy way to approximate these. Instead, the typical approach used in molecular simulations is to choose a cutoff radius (r_c) beyond which forces are considered negligible and ignored. Each timestep, all pairs of particles (neighbours) within this radius must be determined. Searching a given radius from a single point is called a *range query*; more specifically, this problem is a *moving self-spatial join query* [26].

2.5 Spatial Data Structures: Fixed Grid

Building a list of neighbours quickly is of critical importance, because it usually forms a bottle-neck in the processing time of a molecular dynamics simulation [11]. To determine which atoms are in range without resorting to a brute force, objects must be sorted. A *spatial data structure* (also known as an *index*, *spatial access method* or *multidimensional access method*), is a structure used to sort data in *k* dimension space [12]. Many types of spatial data structures exist, and most of these are tree-based structures with *O(N log N)* build times. Many papers investigate different types of indexes [16, 4, 12, 6, 31]. However unless the dataset is extremely skewed, the best performing structure is the fixed grid, the simplest of all structures, which builds in *O(N)* [17]. In a liquid, particles are evenly distributed, and the number of particles is fixed throughout the simulation, so a fixed grid is ideal.

The fixed grid, not to be confused with the grid file [18, 23], has a fixed number of equal sized cells along each axis, as shown in Figure 2.3. Unlike other structures, the cell which contains an atom can be determined in constant time by dividing the atom's position by the cell length along each dimension. A fixed grid is typically stored as an array of cells, where each cell contains a pointer to the list of atoms it contains.

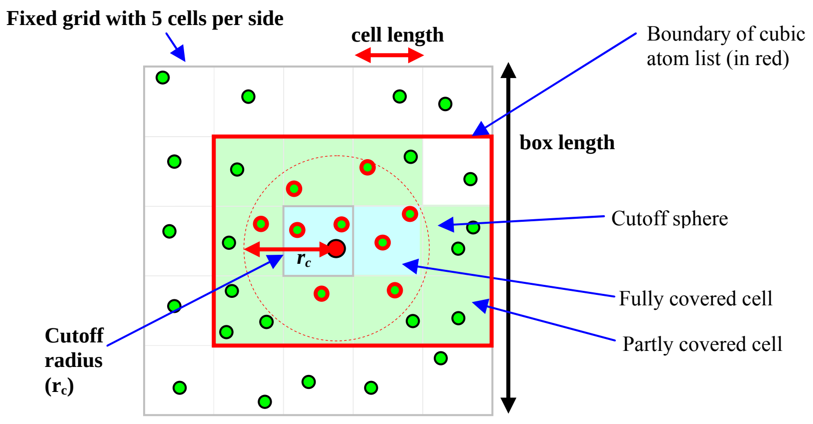


Figure 2.3. Fixed grid showing range search.

2.6 Atom List

To perform a range search, all atoms in a cell within r_c of the root atoms, are candidate neighbours. In Figure 2.3, only the shaded cells need to be searched, although often this list is approximated to a rectangular region (i.e.: the top right cell will be included).

Definition 1. *Let **cubic atom list** refer to a list of all cells within plus and minus r_c of a given atom along each dimension.*
Definition 2. *Let **minimum atom list** refer to a cubic atom list whereby all cells outside r_c are removed.*

Notice that, a cubic atom list will not always be symmetrical about the *root cell* (the cell which the atom resides in). Also notice that some cells are fully covered while others are only partly covered. A disadvantage of atom lists is they involve determining which cells are in range of each atom, and this must be done every iteration. For this reason, the cell list was created.

2.7 Cell List

The *cell list* is a commonly used technique in molecular dynamics and builds on the fixed grid data structure [2]. In this technique, a fixed grid is set up such that the size of each cubic cell side is slightly larger or equal to the cutoff radius r_c (Figure 2.4). Each particle in a given cell therefore only interacts with particles in neighbouring cells. The same list of neighbouring cells is used as a candidate list for each particle in the same cell, but the disadvantage is that a high proportion of candidate particles will be rejected.

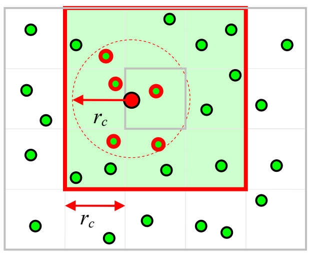


Figure 2.4. Cell list.

2.8 Verlet Neighbours List

The most effective and commonly used time integration algorithm in molecular dynamics is the *verlet neighbour list* [2]. In this technique the cutoff sphere (with radius r_c) around each molecule is surrounded by a larger sphere, called a “skin” with radius r_v. During the first timestep a large neighbours list is constructed, containing all pairs of neighbours within r_v of each other, and this list is rebuilt at intervals. Between these intervals, the neighbours list is simply updated by recalculating distances between neighbours and determining which are within the actual cutoff radius r_c. Rebuild intervals of 10-20 timesteps are common [2]. The algorithm is successful because the skin is chosen to be thick enough that no molecule can penetrate through the skin and into the cutoff sphere between these intervals. For instance, in Figure 2.5, point 2 will never be able to get further than the position 2', and penetrate the cutoff sphere before the list is rebuilt.

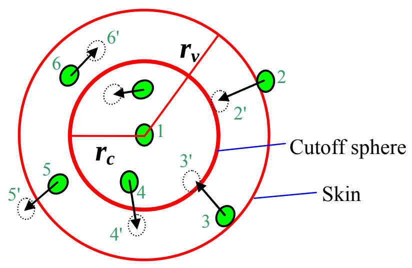


Figure 2.5. Cutoff sphere and skin around an atom.

A refinement of this technique is to store the total displacement for each atom since the last rebuild and only rebuild the neighbours list again when the sum of the two largest displacements exceeds r_v - r_c. It has been shown that a typical simulation using this verlet scheme requires 16 times less pair distance calculations than a pure cell list technique [15]. However, to build the verlet neighbour list itself the cell list technique is commonly used. Each rebuild is accomplished by placing all atoms into cells, and using a cell list to determine which atoms are within r_v (instead of r_c) of each other. Between rebuilds it is unnecessary to place atoms into cells. This combined technique yields the best performance results [15].

2.9 Indexing Pair Potentials

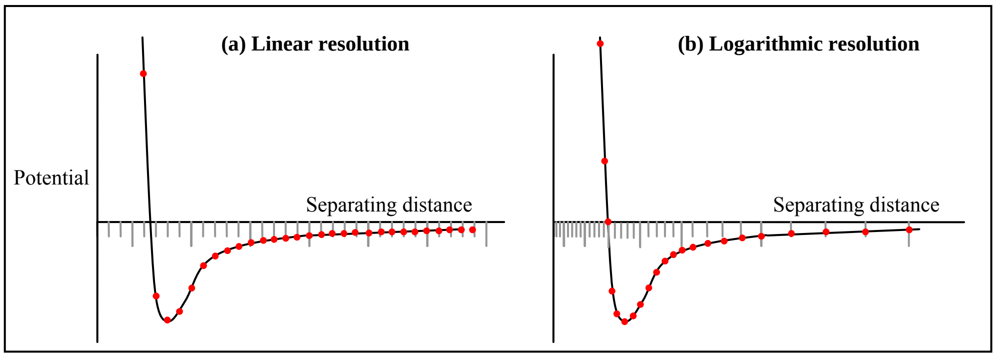


Figure 2.6. Resolving distances to pre-calculated pair-potential force.

Pair potential force must be calculated on the separating distances between every neighbour pair found each timestep. Rather than calculate this every time, a useful technique is to pre-calculate forces for a large series of distance measurements. Distances can then be indexed to an array and an approximated force value resolved. A problem with indexing points at evenly spaced intervals (i.e.: *index = distance between atoms / constant*) is this results in poor accuracy in regions where the curve changes sharply (Figure 2.6.a). A better index resolution strategy would have points (representing pre-calculated distances) closer together near the origin, where the curve changes sharply, and much fewer points toward the cutoff radius. Figure 2.6.b. demonstrates what the effective distribution of points might look like if a logarithmic function was used to resolve distances to the index, resulting in many more points near the origin. It has been demonstrated that using approximate non-linear sampling of the separation distance, only 440 points are needed to create an accurate molecular dynamics simulation [9]. The disadvantage is that logarithmic functions can be expensive (Figure 7.3 in Appendix B), although this too can be determined using a lookup table.[1]

2.10 Improving Spatial Locality with Space-Filling Curves

The algorithm used to build the neighbour list, like most spatial algorithms, will typically require processing of all points in any given cell at a time, and then points in nearby cells in sequence. If the actual array which contains the locations of points is unsorted, it is likely that two nearby points within the same cell will be far apart in memory and accessing one after the other will result in a processor cache miss.

Spatial locality principle: It is probable that objects close to referred ones will be requested again in the future.

To improve spatial locality, an obvious step is to group points in cells together, but main memory performance can be even further improved by sorting points and/or cells using a space-filling curve. A *space-filling curve* is a line passing once through every point in a space, in some order, according to some algorithm. All techniques first partition the universe, in this case the box, into a grid and then assign an order to all cells. The points in the given data set are then sorted and indexed according to the grid cell in which they are contained.

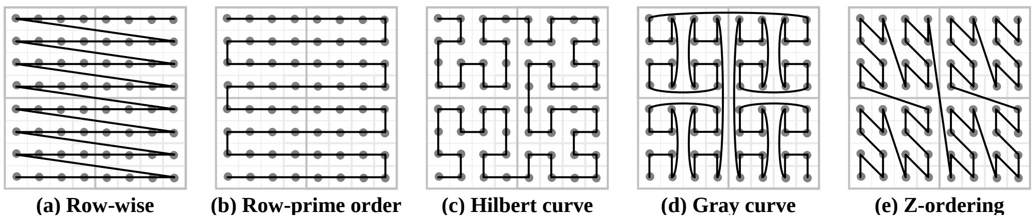


Figure 2.7. Space-filling curves.

[1] The accuracy and performance of these methods were not tested, since the focus of this thesis was only to improve dynamic self-spatial joins.

Figure 2.7 illustrates five common space-filling curves: *row-wise ordering* (which may occur along any dimension), *row-prime ordering* (a slight improvement), *z-ordering* (also known as the *Peano curve* or *quad codes*), the *Hilbert curve* and the *Gray curve*. A good overview of space-filling curves, is provided in [29], and references to algorithms are provided in [12]. All space-filling curves can be applied to any number of dimensions. Studies show that the *Hilbert curve* and *z-ordering* (which is simpler, but slightly less efficient) are the most effective methods [1, 22].

Space-filling curves lend themselves well to fixed grids. However, a characteristic of space-filling curves in moving point environments is that, if points (or atoms in this case) are only ordered once, performance degrades as points move away from their original positions [17]. To prevent this, the points are often re-ordered periodically. Results of using space-filling curves in this thesis are investigated in Section 5.12.

2.11 Spatial Data Structures for Skewed Data

Exploring the skewed data set is outside the scope of this thesis; however an important disadvantage of the simple fixed grid is that it does not deal well with "dead space" – large regions of space with no points. Figure 2.8 shows the use of a fixed grid for a uniform and skewed set of points. In order to optimise performance and prevent too many atoms falling in a single cell, the number of cells per side in a skewed distribution will be greater than that for a uniform distribution with the same number of points. In Figure 2.8.b, the number of cells per side has been doubled for the skewed distribution. However, even with four times as many cells, certain cells still contain numerous points, which will slow down processing every time that cell is found in range of an atom. More importantly, most cells are completely empty, and therefore represent wasted data storage.

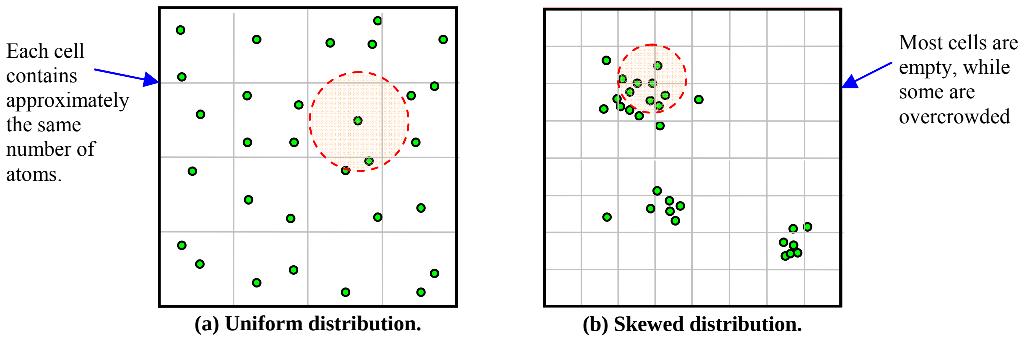
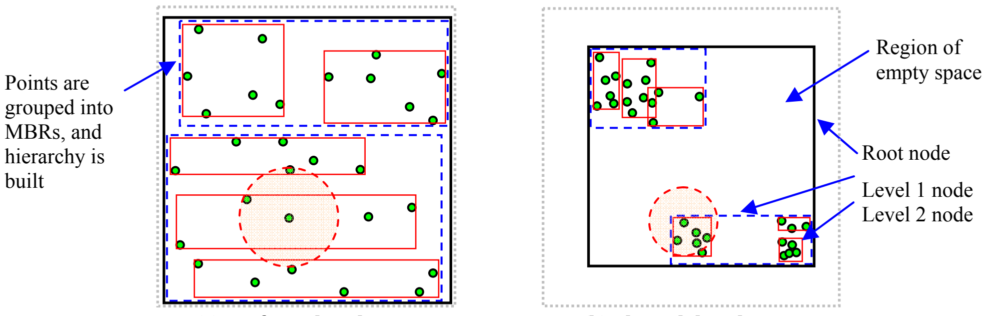

Each cell contains approximately the same number of atoms.
(a) Uniform distribution.
(b) Skewed distribution.
Most cells are empty, while some are overcrowded

Figure 2.8. Fixed grid for skewed and uniform particle distribution.

Definition 3. *A **minimum bounding rectangle (MBR)** is the smallest rectangle completely enclosing a set of points and whose sides are parallel to the coordinate axes. MBRs are typically represented by two points, the lowest point (minimum edge along each dimension) and the highest (maximum edge along each dimension).*

Unlike the fixed grid, most tree-based structures, including kd-trees [10] and quad-trees [28], greatly minimise wasted storage of dead space. For the simulation of extremely skewed data, such as a galaxy of stars, the R-tree is particularly effective [14]. The R-tree is a balanced tree whereby each subtree groups nearby objects together inside a *minimum bounding rectangle (MBR)*, as shown in Figure 2.9. In liquid, however, particle distribution is uniform; therefore MBRs occupy larger volumes. Furthermore, hierarchical data structures are much more expensive to rebuild each timestep than a fixed grid.



(a) Uniform distribution. **(b) Skewed distribution.**

Figure 2.9. R-tree for skewed and uniform particle distribution.

Notably, it has been discovered that, even for highly skewed data, a fixed grid performs orders of magnitude better than hierarchical data structures including the R-tree, R*-tree and quad-tree for the simulation of moving points in two dimensions [17]. Furthermore, a special variation of the fixed grid called a "two-tier grid", which is similar to a quad-tree, has been designed to better deal with highly skewed data [17].

2.12 Summary

In this chapter, an overview of molecular dynamics was provided, including the popular Lennard-Jones potential pair algorithm, periodic boundary condition and the use of a fixed grid to perform range searches. Two methods to execute range searches over a fixed grid, the traditional cell list and an "atom list", were also introduced, as was the verlet neighbour list technique. Furthermore, this chapter reviewed several popular space-filling curves and concepts related to a skewed distribution of points.

3 Proposed Techniques to Improve Self-Spatial Joins

In this chapter, several new optimisation techniques for spatial join queries on a fixed grid are proposed and explained. Note that the performance result of each technique appears under the corresponding sub-heading in Chapter 5. A fundamental approach to increasing query speed is to reduce the search space, which is what many of these techniques attempt to achieve. Immediately following are the definitions of some important values used throughout the rest of the thesis.

Definition 4. *Let* $\mathbf{r_s}$ *be defined as the search radius* r_s *used to find neighbour pairs. Note that this value of* r_s *will depend on whether the verlet technique is used (Section 2.8) as follows:*

$$r_s = \begin{cases} r_v \,, & \textit{if verlet technique is used} \\ r_c \,, & \textit{if verlet technique is not used} \end{cases}$$

Definition 5. *Let* ***cellSidesPerRs*** *be defined as the side length of a single cell in the fixed grid divided by the search radius* r_s *as follows:*

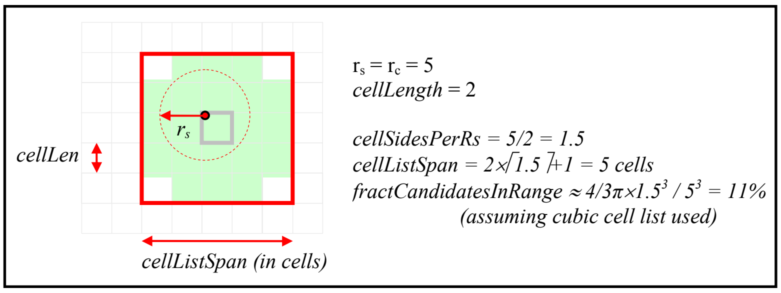
$$cellSidesPerRs = \frac{length\ of\ cell\ side}{r_s}$$

Definition 6. *Let* ***cellListSpan*** *be the number of cells which a cell list will span in each direction, as given by:*

$$cellListSpan = 2 \times \lceil cellSidesPerRs \rceil + 1$$

Definition 7. *Any atom checked against another atom (to check if it is within* r_c *and therefore a neighbour) will be called a* ***candidate neighbour****, and the fraction of candidates in range, denoted by* ***fractCandidatesInRange****, will be given by:*

$$fractCandidatesInRange = \frac{total\ \#\ of\ candidate\ neighbours\ in\ range}{total\ \#\ of\ candidate\ neighbours} \approx \frac{volume\ of\ cutoff\ sphere}{volume\ of\ average\ cells\ searched\ per\ atom}$$



r_s
cellLen
cellListSpan (in cells)
$r_s = r_c = 5$
$cellLength = 2$
$cellSidesPerRs = 5/2 = 1.5$
$cellListSpan = 2\times\lceil 1.5 \rceil+1 = 5$ cells
$fractCandidatesInRange \approx 4/3\pi\times 1.5^3 / 5^3 = 11\%$
(assuming cubic cell list used)

Figure 3.1. Example of cell list illustrating definitions 5-7.

3.1 Minimum Cell List

The original cell list technique (Section 2.7) proposes choosing a *cell length* $\leq r_c$. This effectively means executing each range search over a cube with a volume of at least $3^3 \times r_c$, which is 6.45 times the volume of the cutoff sphere ($4/3\pi \times r_c^3$) when r_c = *cell length*. On average, only 15.5% of candidates will be in range. A way to improve on this is to make a finer grid by choosing a cell length greater than r_c. The cell list will now span more than 3 cells along each dimension (Definition 6).

A cell list spanning more than three cells can be refined to eliminate any cells beyond r_c of the root cell; this effectively describes a box with rounded edges, as illustrated in Figure 3.2. To avoid confusion with the original cell list, this technique will be called a *minimum cell list*, and a cell list which has not been refined will be called a *cubic cell list*.

Definition 8. *Let **cubic cell list** denote a traditional cell list which includes all cells within plus and minus r_s of a root cell along each dimension.*

Definition 9. *Let **minimum cell list** denote a cubic cell list which has been refined to exclude any cells further than r_s from the root cell.*

Notice that the difference in volume between the minimum cell list and cutoff sphere reduces as the grid becomes finer. Whereas a circle occupies 78% of the area of its bounding square, a sphere only occupies 52% of the volume of its bounding cube. Generation of a minimum cell list only needs to occur once at the start of each simulation (or each time r_c changes relative to the cell length), and the same single list can be used like a template for each range search from any cell. Results of using a minimum cell list as opposed to a cubic cell list technique are investigated in Section 5.4.

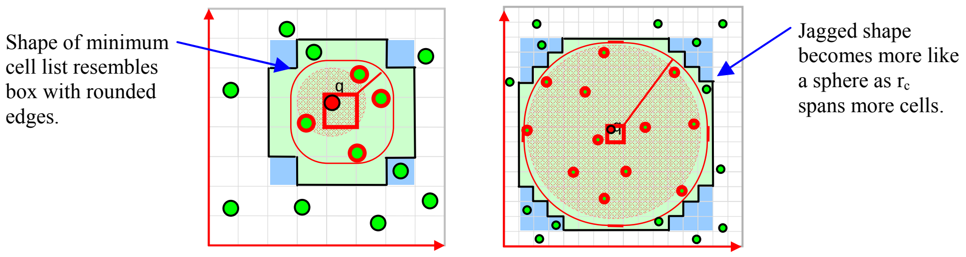

Shape of minimum cell list resembles box with rounded edges.
q
q
Jagged shape becomes more like a sphere as r_c spans more cells.

Figure 3.2. Minimum cell list.

3.2 Half Range Searches

Often it may be desired to store a list of neighbours of each atom separately, but in molecular dynamics simulations it is more efficient to find/store a single exhaustive list of all unique neighbour pairs. A problem with the original cell list is that each pair is captured twice; making it necessary to check for duplicates. In Figure 3.3.a, point *j* will be captured when searching from point *i*, and point *i* will be captured when considering point *j*.

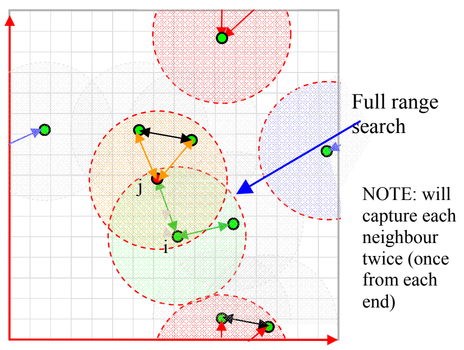
The proposed solution is to restrict range searches so that all points on a chosen side of a chosen axis are ignored, thereby only searching one half of the cutoff sphere. This technique will be called *half range search* and half range searches can be used to resolve self-spatial join queries, since the radius of each range search is the same.

Definition 10. *Let* ***half range search*** *denote a range search which excludes a chosen side of a chosen dimension. In the case of three dimensions, only a hemisphere is searched.*

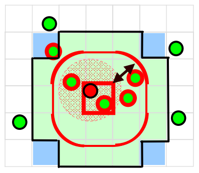
So now, instead of generating a perfectly symmetrical minimum cell list, all cells above or below the root cell, along one chosen axis, can be safely eliminated. Figure 3.3.b shows a half range search whereby only the upper hemisphere is searched. When considering *j*, the lower point *i* can be ignored, since it will obviously be captured when considering *i* to *j*.

Definition 11. *Let* ***half minimum cell list*** *denote a minimum cell list (Definition 9) designed to execute half range searches (Definition 10) by excluding all cells above or below the root cell along a chosen axis.*

Normal approach: Search sphere

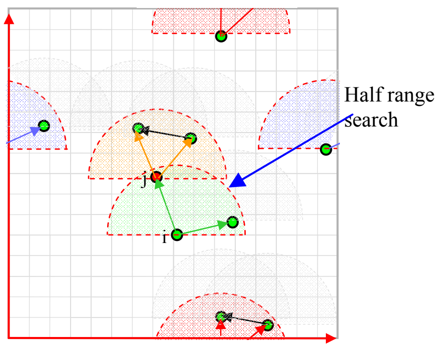

Full range search
j
i
NOTE: will capture each neighbour twice (once from each end)

(a)
Full minimum cell list

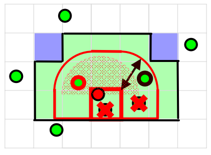


Representation: { (-1,-2), (0,-2), (1,-2)
(-2,-1), (-1,-1), (0,-1), (1,-1), (2,-1),
(-2,0), (-1,0), (0,0), (1,0), (2,0),
(-2,1), (-1,1), (0,1), (1,1), (2,1),
(-1,2), (0,2), (1,2) }

Faster approach: Search upper hemi-sphere



(b)
Half minimum cell list



Representation: { (-2,0), (-1,0), (0,0), (1,0), (2,0),
(-2,1), (-1,1), (0,1), (1,1), (2,1),
(-1,2), (0,2), (1,2) }

Figure 3.3. Use of full range search vs. half range search.

Note that the axis which is "halved" should be chosen carefully to improve spatial locality. For instance, the code used in testing [24] stored the master cell array as a one-dimensional vector such that the cell at location *(x, y, z)* maps to the element at index *(x×cellsPerSide*^{2} *+ y×cellsPerSide + z)*. By halving along the x-axis instead of the y or z axis, each half range search will be spread across a smaller, more contiguous, range in memory. Results of using half ranges searches versus full range searches are investigated in Section 5.7.

3.3 *Early Elimination of Non-Neighbours*

For any given search, even one with a high value of *cellSidesPerRs* (Figure 5.7), the fraction of candidates in range (Definition 7) is usually low. Normally, to check the distance between points[2], the programmer would simply calculate:

$$dist\{i,j\} = \sqrt{(j_x - i_x)^2 + (j_y - i_y)^2 + (j_z - i_z)^2}\ .$$

To reduce the cost of "*neighbour misses*", the programmer should first discard any candidate pair where *j* is above *i* according to half range search criteria (i.e. if $j_y > i_y$), using some

[2] For the sake of simplicity all distances are represented as $j_y - i_y$. Notice however, that if periodic boundary condition is used (as it was in this thesis) distances along each dimension are determined by first checking if they wraps around the box boundary (Section 2.3).

tiebreaker if $j_y=i_y$. Next, the distance along each axis can be calculated and checked against r_c separately; and the pair discarded immediately if any distance is greater than r_c (i.e. if $j_x-i_x > r_c$). The latter step will be referred to as the *early elimination*.

Definition 12. *Let* ***early elimination*** *describe the process of eliminate a candidate pair early if the distance along any axis is greater than the cutoff radius, rather than waiting to calculate the final distance.*

Definition 13. *Let* ***fractionAtomsEliminatedEarly*** *denote the fraction of atoms subject to early elimination. If a rectangular region of cells is searched, this is given by:*

$$fraction\ atoms\ eliminated\ early = 1 - \frac{(2 \times r_c)^3}{volume\ of\ cells\ searched}$$

For instance, using a full cubic cell list and the common case of *cellSidesPerRs=1*, the *fractionAtomsEliminatedEarly* is 70% ($(\{2 \times 1\})^3 / 3^3$), which is considerable, although one should note that if a minimum cell list is used, *fractionAtomsEliminatedEarly* decreases quickly as *cellSidesPerRs* increases, since a minimum cell list cuts out most of the cells outside r_c.

Finally, the distance squared should be calculated {i.e.: $dist^2\{i,j\} = (j_x - i_x)^2 + (j_y - i_y)^2 + (j_z - i_z)^2$ } and discarded if greater than r_c squared (i.e. if $dist^2 > r_c^2$). The reason for this is that square root operations are expensive (Figure 7.3 in Appendix B) and should be avoided wherever possible. Notice that, using the Lennard-Jones equation, it is possible to use the distance squared directly and avoid ever calculating the actual distance (Section 2.2). Results of using the early elimination of non-neighbours technique are investigated in Section 5.8.

3.4 Sub-grids and Cell List Template Guides

In the half range search shown in Figure 3.4, it is obviously a waste of time to check over the points in the shaded adjacent cells. Although they are in range of the root cell, they are not in range of the root atom. However, calculating the minimum distance from every atom to every adjacent cell can be expensive. The idea of a sub-grid is that every cell in the main grid is divided into a smaller sub-grid and, for every range search, the appropriate sub-cell for each root atom is determined in constant time. Notice that atoms are stored and sorted in cells, but not stored in sub-cells. For each sub-cell, an array is pre-computed before the simulation begins, whereby each element corresponds to a cell in the cell list and effectively says true or false: is this adjacent cell in range. For instance, in Figure 3.4 only two of the six adjacent cells (33%) are in range of the bottom left sub-cell.

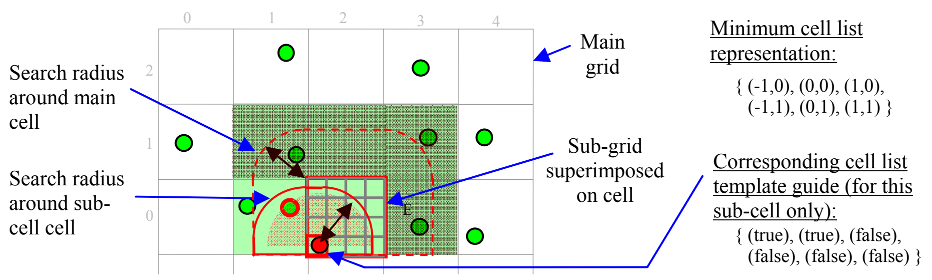

0
1
2
3
4
2
1
0
Search radius around main cell
Search radius around sub-cell cell
Main grid
Sub-grid superimposed on cell
E
Minimum cell list representation:
{ (-1,0), (0,0), (1,0), (-1,1), (0,1), (1,1) }
Corresponding cell list template guide (for this sub-cell only):
{ (true), (true), (false), (false), (false), (false) }

Figure 3.4. Use of sub-grid to refine search.

The reason for using a sub-grid instead of just using a finer main grid is primarily due to storage requirements. Although making a grid with 100 cells per side sounds trivial, it means storing 1 million (100^3) cells. Note that the optimal number of cells per side is much higher if points are highly skewed (Section 2.11). Results of using sub-grid cell list template guides are investigated in Section 5.9.

3.5 Minimum Bounding Rectangles in Cells

R-trees are a hierarchical spatial data structure built using MBRs (Definition 3). However it was realised that the use of MBRs can also be applied to fixed grid. Specifically, a MBR can be kept for each cell and updated whenever a point moves within that cell. Figure 3.5 shows the use of MBRs in cells for an evenly and skewed distributed set of points. *Minimum bounding spheres (MBS)* could also be kept and used, but these would be more expensive to compute [33].

For each range search (on every atom), there will always be a considerable proportion of cells only just tipped/overlapped by the cutoff radius (Figure 3.5.a) – depending on the number of *cellSidesPerRs*. Such cells can be indicated, and for these cells, it is possible to check if the distance from the root atom to the MBR is greater than the cutoff radius, in which case the exhaustive checking of these atoms could be bypassed.

This method can be combined with the minimum cell list and sub-grid method. One idea is that each sub-cell could store a pre-computed array of values, whereby each value corresponds to an adjacent cell and indicates what percentage of the cell overlaps the cutoff boundaries. All cells below a certain value could have their MBRs checked.

However, whether or not this method can produce a timesaving depends on many factors. If there are too few particles/points per cell, it will be quicker to check the root atom against each of these particles, rather than their MBR. The other factors are the fraction of cells just tipped, and the number of these with MBRs out of range. Furthermore, if the points have a skewed distribution, MBR are likely to be smaller (on average). In liquid, where particle distribution is very even, each MBR is likely to occupy a large volume of each cell, depending on the number of atoms per cell. Results of using MBRs in cells are investigated in Section 5.10.

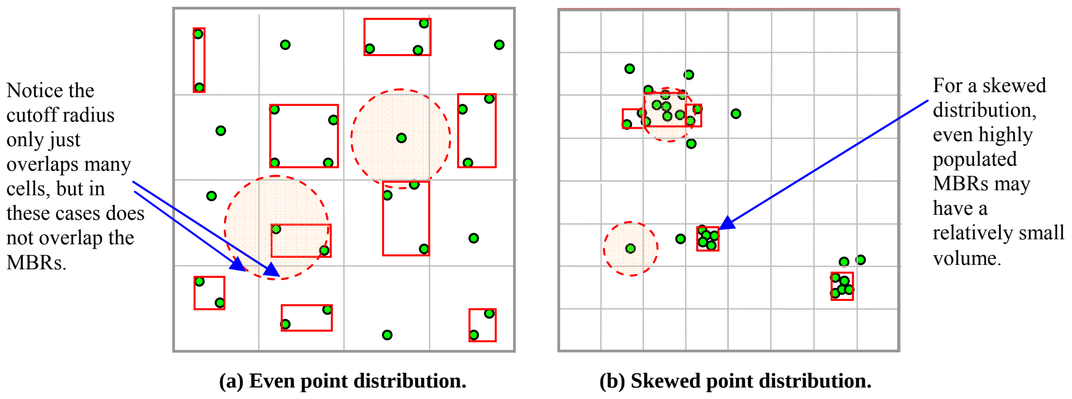

Notice the cutoff radius only just overlaps many cells, but in these cases does not overlap the MBRs.
For a skewed distribution, even highly populated MBRs may have a relatively small volume.
(a) Even point distribution.
(b) Skewed point distribution.

Figure 3.5 The use of MBR in grid cells for two different data sets.

3.6 *Selective Checking of Verlet Neighbours*

The verlet neighbour list (Section 2.8) is effective, because the cost of updating a list is much cheaper than rebuilding it. To reduce the cost of updating the neighbour list further, this thesis proposes two methods. Firstly, it was thought that, rather than check displacement of all atoms every timestep, a more efficient algorithm could measure the number of iterations between rebuilds, and only check displacements at sensible intervals, which would decrease towards once per iteration as the sum of the two largest displacements approaches r_v - r_c (Section 2.8). For example, if it takes twenty iterations before a rebuild is needed, chances are it will take about twenty iterations before the next rebuild. Instead of checking every timestep, the algorithm might check after the tenth iteration, and depending on the maximum displacements, might decide that it is safe to delay the next check for another four timesteps.

Moreover, a similar approach could be used to avoid checking neighbours which are within the verlet shell radius r_v, but not within r_c of each other, every timestep. In Figure 2.5 for example, atom 3 is just within r_v of atom 1, and will presumably take several timesteps before it can move within range of r_c. This technique could be successful because the number of atoms which are within r_v but outside r_c, can be significant, especially if a large r_v is used.

Definition 14. *Let **fractNeigOutsideRc** be the fraction of all neighbours in a verlet list which are not within the cutoff radius distance of each other, which can be approximated as:*

$$fractNeiOutsideRc = \frac{4/3\,\pi \times r_v^3 - 4/3\,\pi \times r_c^3}{4/3\,\pi \times r_v^3} = \frac{r_v^3 - r_c^3}{r_v^3} = \frac{(r_v/r_c)^3 - 1}{(r_v/r_c)^3}$$

Definition 15. *Let **safeDist** be the minimum distance two atoms must travel before they can become in range of each other, as given by:*

$$safeDist = distance\ between\ atoms - r_c$$

Instead of calculating the distance between all neighbours each timestep, the *safeDist* can be used to determine how long to delay checking the distance between any neighbours outside r_c. To decide when the distance between these atoms should be checked again a number of schemes may be appropriate and two of these are proposed below.

The first scheme may be to record the maximum velocity of any particle in the system each timestep, and use this to determine exactly how many iterations to wait before checking the distance between those neighbours, as shown in Figure 3.6 and the following equation:

$$iterations\ before\ next\ check\ of\ neighbours\ pair = \frac{safeDist}{2 \times (max\ velocity \times timestep)}$$

```
determine max velocity of any atom for this iteration
For each verlet neighbour:
    If (iterations before next check > 0):
        decrement iterations before next check
    Else:
        update distance between neighbours                    // (expensive step)
        safeDist = distance between neighbours - $r_c$
        iterations before next check = safeDist / (2 $\times$ max velocity $\times$ timestep)
```

Figure 3.6. Pseudo code for selective checking of neighbours using maximum velocity.

A more sophisticated scheme would be to record the *safeDist* for each neighbour pair, and decrement this each iteration. When the *safeDist* for any pair becomes less than zero, the distance between neighbours would be computed again. Initially it was thought the velocity of both atoms in a neighbour pair could be used to decrement the *safeDist* each timestep, however many interaction models also take acceleration into account when moving atoms, or may not even want to record velocity at all. For this reason, using the displacement of atoms between timesteps is much safer. Each timestep, the displacement of all atoms since the last timestep would be calculated and recorded. The verlet neighbour list would then be updated, such that any neighbour pair with a *safeDist* greater than zero would have its *safeDist* value decremented by the sum of the displacement of both atoms since the last timestep. Finally, if the *safeDist* is less than zero, the distance between neighbours and *safeDist* would be re-calculated. Figure 3.7 helps illustrate this process using pseudo code. Results of using these selective neighbour checking techniques are investigated in Section 5.15.

```
For each atom:
    calculate displacement since last timestep
For each verlet neighbour:
    If (safeDist > 0):
        safeDist = safeDist – (displacement first atom + displacement second atom)
    If (safeDist <= 0):
        update distance between neighbours                  // (expensive step)
        safeDist = distance between neighbours - rc
```

Figure 3.7. Pseudo code for selective checking of neighbours using atom displacement.

4 Implementation

This chapter gives an overview of how simulations were implemented and results obtained. Test code is available at <http://manning.it.jcu.edu.au/~jc130551/thesis/> [24].

4.1 Simulation Testing Sequence

The following is the sequence of main events involved in each simulation.

1. Initial setup:
    1.1. All atoms were placed into a perfect lattice formation, and given slight random offsets.
    1.2. All atoms were given random velocities (Appendix D).
    1.3. The vector of cells forming the grid was initialised (using some number of cells per side).
    - *NOTE: This step often included generation of a minimum cell lists template, loading of cell lists and several other setup functions.*

2. For each iteration (until the desired number of timesteps had elapsed):
    2.1. A list of all neighbours was compiled.
    - *NOTE: This was achieved by placing all atoms into cells then executing a complete self-spatial join using r_v if verlet was used, or r_c if verlet was not used. For a verlet update, this step involved checking distance between atoms in the verlet neighbours list.*

    2.2. Forces between neighbours were calculated and accumulated on each atom using the neighbours list.
    - *NOTE: Lennard-Jones pair potential was used.*

    2.3. Atoms were moved, which included checking if coordinates fell outside the box, in which case they were wrapped around.
    2.4. Timestep was incremented.

This is standard procedure for molecular dynamics simulations. Typically, the step of building the neighbours list is by far the most time consuming part of a molecular dynamics simulation [11].

4.2 Scientific Testing Process

All experiments were run on the same computer, a 2.6 GHz Pentium 4 machine with 512 MB of RAM and 512kB of L2 cache. The molecular dynamics system tested was implemented as a C++ console application written, compiled and executed using Microsoft Visual C++ 6 [30]. Each graph of results in Chapter 5 represents a series of simulations, executed in batch. The number of timestep/iterations used in different simulations was varied, since simulating over fifty thousand atoms could take several minutes per iteration, but simulating five hundred took less than a second per iteration. For this reason, a time limit of about one minute was set for most simulations, at which point they were terminated, even if the desired number of iterations had not completed. Results of the simulations were appended to a CSV file, and analysed using Microsoft Excel.

Many metrics and tallies, including the number of distance calculations, were recorded, but the main metric of performance presented in results was the average number of CPU tics elapsed per iteration. On the test machine each CPU tic was approximately 0.015 of a second (approximately 67 tics per second). Since the goal of this thesis was to improve overall simulation speed, CPU tics was the most practical metric for comparing different methods. However, while the simulation times lead to reasonably accurate estimates of relative speedup between competing algorithms, the absolute measure of CPU time is dependent on compiler tools, programming language, and the underlying computer hardware used to run simulations. Running the same batch of simulations on a machine with different specifications is likely to produce different results. For example, the performance improvement of space-filling curves (Section 5.12) should be significantly better on a machine with better cache. Furthermore, the best implementation techniques on one compiler might not necessarily yield the best results on a different compiler. All simulations presented were small enough to fit entirely in memory.

5 Experimental Results and Discussion

In this chapter, the performance results of various techniques and implementation options, including all techniques proposed in Chapter 3, are presented. The sections and techniques tested are arranged in a logical sequence, such that the most successful techniques discovered in one section are generally adopted in all subsequent sections. Most unsuccessful techniques were abandoned immediately. Sections 5.2 to 5.13 are focussed on optimising the build time for the neighbours list and do not use the verlet neighbour list technique; only Sections 5.14 and 5.15 use the verlet list technique. The final two sections, Sections 5.16 and 5.17, test accuracy relationships specific to a molecular dynamics simulation using the Lennard-Jones interaction model.

5.1 Guide to Results

Each graph in this chapter shows the values of the main input parameters used in the simulation, hence allowing replication of results. A guide to these variables is provided below. Table 1 shows the main input variables defined at the beginning of each simulation, Table 2 shows several variables which were useful in analysis of results, and Table 3 shows a few of the most important variables calculated at the end of every simulation, many of which represent performance metrics. Many of the names for these variables are used throughout the rest of the thesis. Appendix A details the properties of the data types shown. In all tests atom coordinates and forces are represented as doubles (Appendix C).

Variable Name	Description	Data Type	Comment
numAtoms	Number of atoms used in the simulation.	*integer*	In all simulations atoms were evenly distributed; never clustered.
rc	Cutoff radius	*double*	Cutoff radius was never allowed to exceed *boxLen*/2.
boxLen	Length of the cubic box sides	*double*	
CPS	Cells per side used in the simulation	*integer*	Many graphs show how performance varies as *CPS* varies. Other results attempted to find an optimal *CPS*, and then ran the simulation.
timeStep	Duration of each timestep	*double*	Notice the effective duration of a simulation is equal to $timeStep \times timeStepsToExe$.
timeStepsToExe	Number of timesteps to execute	*integer*	Many simulations were ended early if they exceeded some time limit.
useVerlet	Shows whether or not verlet neighbour list technique was used	*bool*	Note that only the last few sub-sections of results use the verlet list technique.
rv	Verlet radius used is above was true	*double*	

Table 1: Main input variables.

Variable Name	**Description**	**Data Type**	**Value**	**Comment**
cellLen	Length of each cell side	*double*	*boxLen/CPS*	-
boxVol	Volume of the box	*double*	$boxLen^{3}$	-
density	Density of atoms (number of atoms per unit volume)	*double*	*numAtoms/boxVol*	Often *boxLen* was set such that density would stay constant across a set of simulations.
numCells	Number of cells in the grid	*integer*	CPS^{3}	-
avgAtomsPerCell	Average number of atoms per cell.	*double*	*numAtoms/numCells*	-
rs	Represents the search radius used to build the neighbour list, and depends on whether the verlet technique is used.	*double*	*if useVerlet is true* *rs = rc* *else* *rs = rv*	Helps eliminate confusion.
rvDivRc	The verlet radius compared to the cutoff radius	*double*	*rc / rv*	Best way to measure *rv* in results.
avgLatticeSpacing	Distance between adjacent atoms if set in a lattice structure.	*double*	$numAtoms^{1/3}$ / *boxLen*	Useful to visualise rough separation of atoms.
cellSidesPerRs	Number of cell sides covered by rs.	*double*	*cellLen/rs*	-

Table 2: Useful derived variables.

Variable Name	**Description**	**Data Type**	**Comment**
avgTicsPerIt	Average number of tics elapsed during each iteration. Note that every 100 tics took approximately 1.5 seconds.	*double*	The most common metric of performance used in this thesis. Does not include setup time, but does include building the neighbour list, calculating forces and moving atoms etc.
UavgTicsPerIt	Average number of tics for each iteration not including calculating forces and moving atoms.	*double*	Includes building neighbour lists, and updating verlet list if the verlet technique is used.
avgNeiSize	Average size of the neighbours list.	*double*	If verlet neighbour list is used, *avgNeiSize* will typically be greater than the actual average number of true neighbours found each iteration.
avgInRc	Average number of neighbour pairs within *rc* found each iteration.	*double*	If verlet technique is not used, this will be equal to *avgNeiSize*.
avgNeiPerAtom	Average number of neighbours per atom	*double*	Is equal to $2 \times avgInRc/numAtoms$, but can be estimated/approximated before the simulation by $(numAtoms-1) \times \{(4/3\pi \times r_c^{3}) / boxVol\}$

Table 3: Results and performance measures.

5.2 Scalability of Fixed Grid

There is a slight misconception in molecular dynamics simulations that performance of any even distribution simulation, using a fixed grid, scales linearly with the number of particles. Results in Figure 5.1 and Figure 5.2 show this is only true if the density of atoms stays constant relative to r_c. In Figure 5.2, r_c was kept constant, but the box length and number of cells per side increases such that the length of each cell and the average number of atoms per cell stayed constant. Therefore, in reality, this type of molecular dynamics simulation scales *O(N)* whereby $N = numAtoms \times (1 + avgNeiPerAtom)$. This relationship is shown in Figure 5.1 where performance increases roughly proportionally with the volume of the cutoff sphere and therefore the average number of neighbours found each iteration.

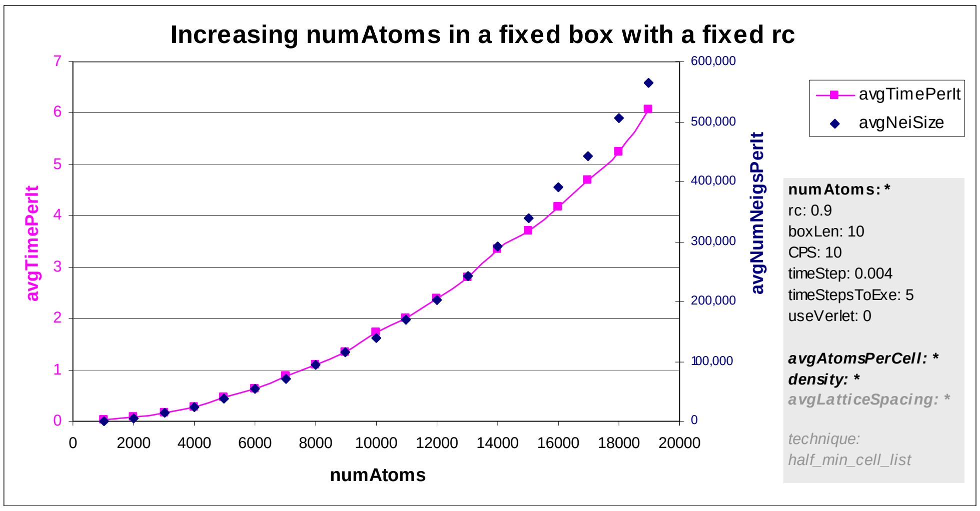


Figure 5.1. Performance when increasing the number of atoms in a fixed size box.

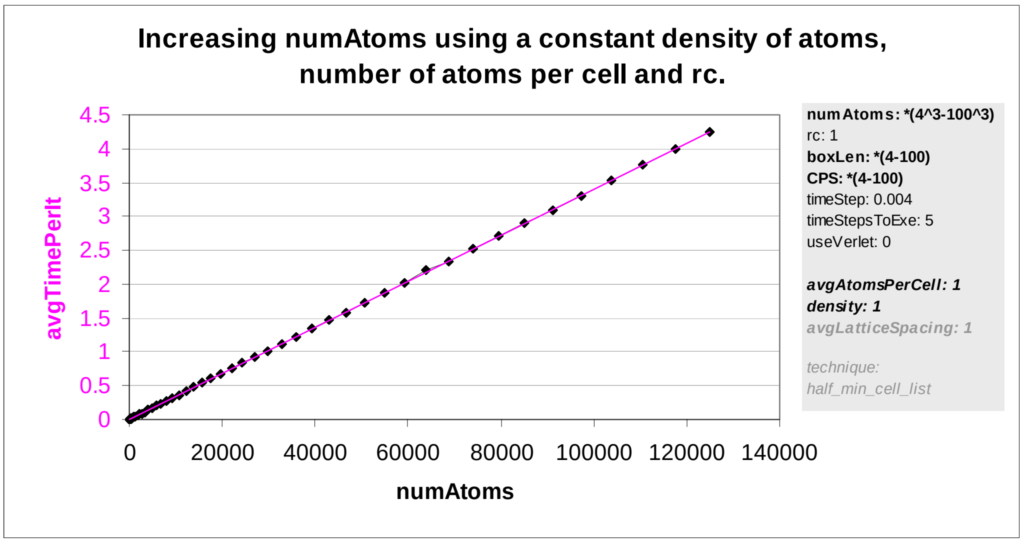


Figure 5.2. Performance when keeping atom density and atoms per cell constant.

5.3 Breakdown of Costs

The following experiments demonstrate the cost of performing each of the steps involved in a single timestep/iteration in the simulation (Section 4.1). Figure 5.3 and Figure 5.4 show the same set of results, whereby 10000 atoms were simulated using the Lennard-Jones pair potential model using a half atom list technique and a range of cutoff radius values. Varying the number of atoms had negligible affect on this performance breakdown, meaning results in Figure 5.4 were representative of simulations with any number of atoms. These results show that time to build the neighbours list took the bulk of processor time. As r_c increases and *avgNeiPerAtom* increases, the proportional cost of placing atoms in cells and repositioning atoms reduces. In these simulations, total build time (which includes putting atoms into cells) took about 85-90% of the total cost per iteration. Calculating forces, velocity and positions for each iteration took the remainder of the time.

Note these results are just a rough guideline; the performance breakdown depends largely on the cost per neighbour of the interaction model used, not to mention the cost of and frequency of any scientific functions which might be used. In these tests a simple implementation of the Lennard-Jones potential algorithm was used to calculate forces, but the performance of this could have been significantly improved by implementing techniques described in Section 2.9.

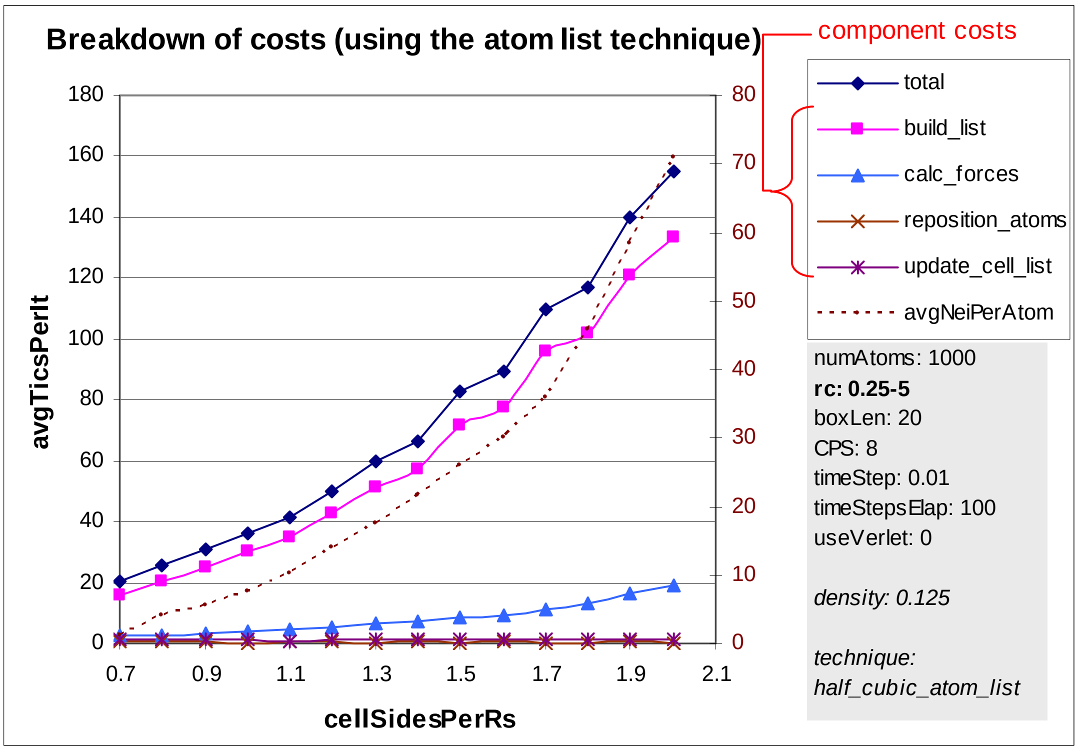


Figure 5.3. Various costs of using atom list.

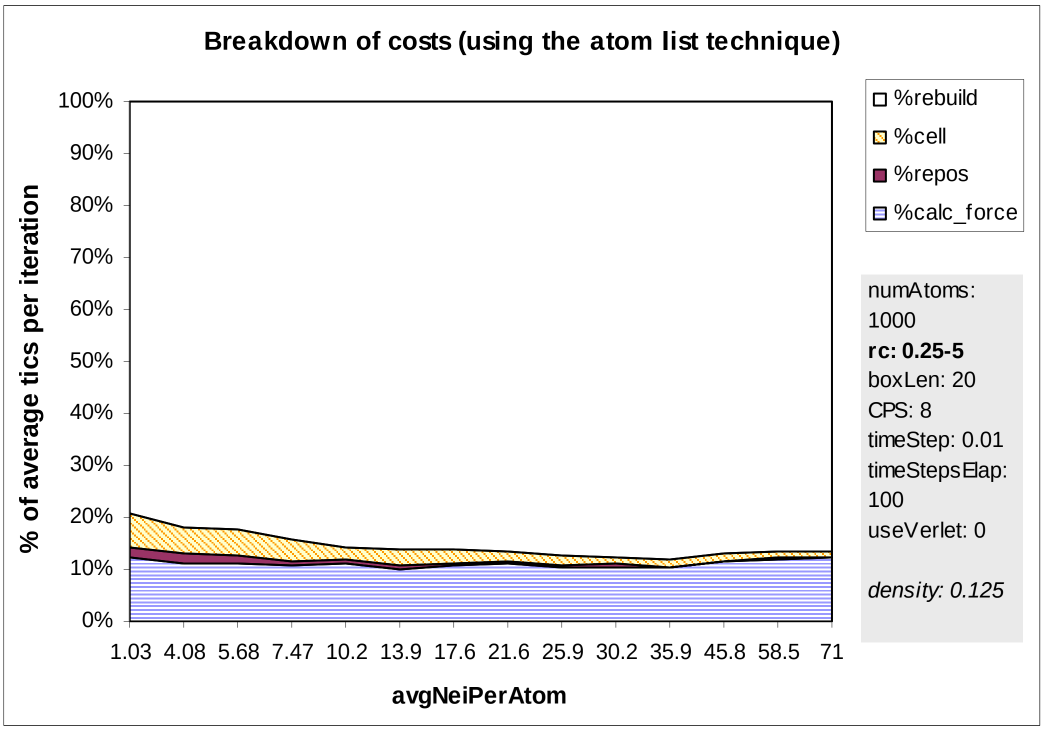


Figure 5.4. Breakdown of costs using atom list.

5.4 Minimum Cell List

The following graphs show the difference in volume (the number of cells) between four different types of cell lists outlined in Sections 3.1 and 3.2. The number of cells was calculated by a small fragment of code, which first generated the cubic cell lists and then eliminated any cell outside of r_c to create the corresponding minimum cell list. Figure 5.5 and Figure 5.6 show the volume of the minimum cell list increases quite smoothly compared to the volume of the cubic cell list. Figure 5.5 shows that the volume of the minimum cell lists compared to the volume of the cubic cell lists slowly approaches 52% as *cellSidesPerRs* increases, although the trend is very jagged. The difference is always greatest just after the number of cells in the cubic cell list increases. Obviously, refining cubic cell lists into minimum cell lists is well worth the effort of coding if *cellSidesPerRs* is greater than one. Figure 5.7 shows the fraction of candidates in range (Definition 7) for each type of cell list. Even for high values of *cellSidesPerRs*, the fraction of candidates not in range is large, thus representing a large detriment to performance.

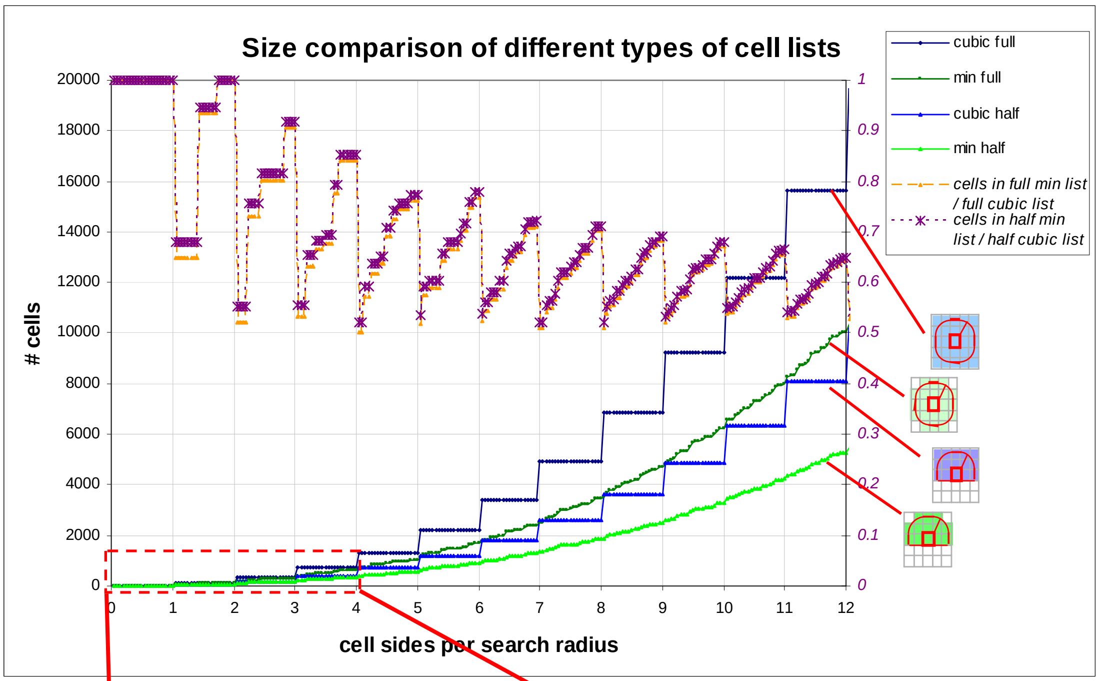


Figure 5.5. Number of cells of minimum cell lists vs. cubic cell lists.

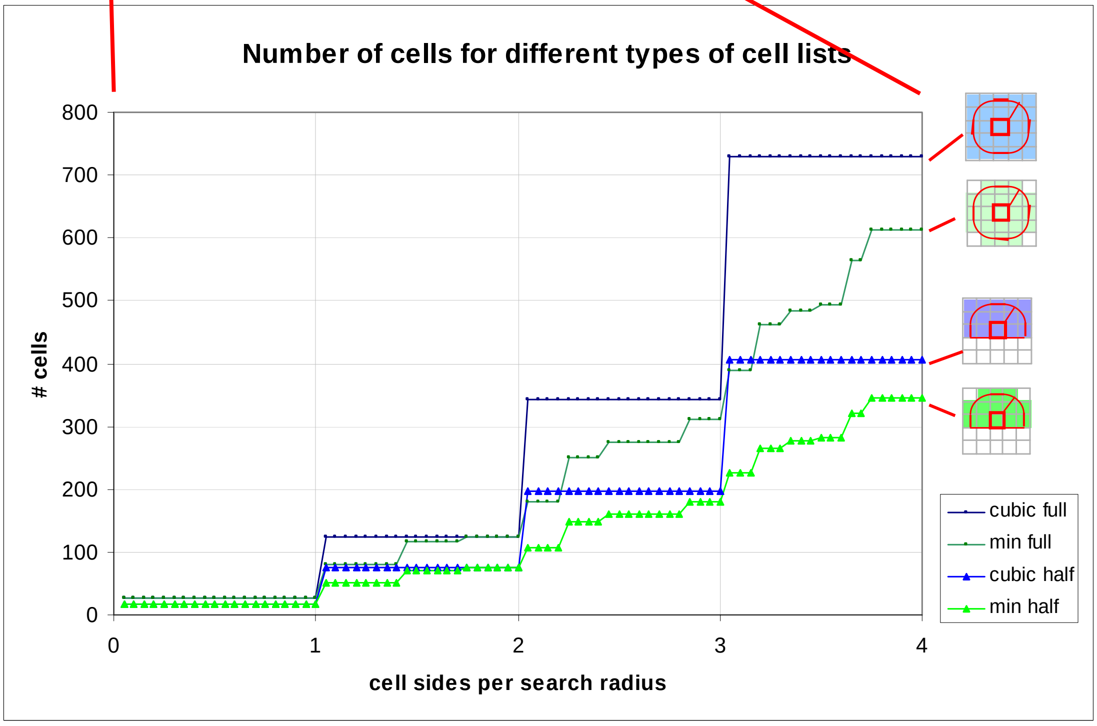


Figure 5.6. Number of cells in different cell lists.

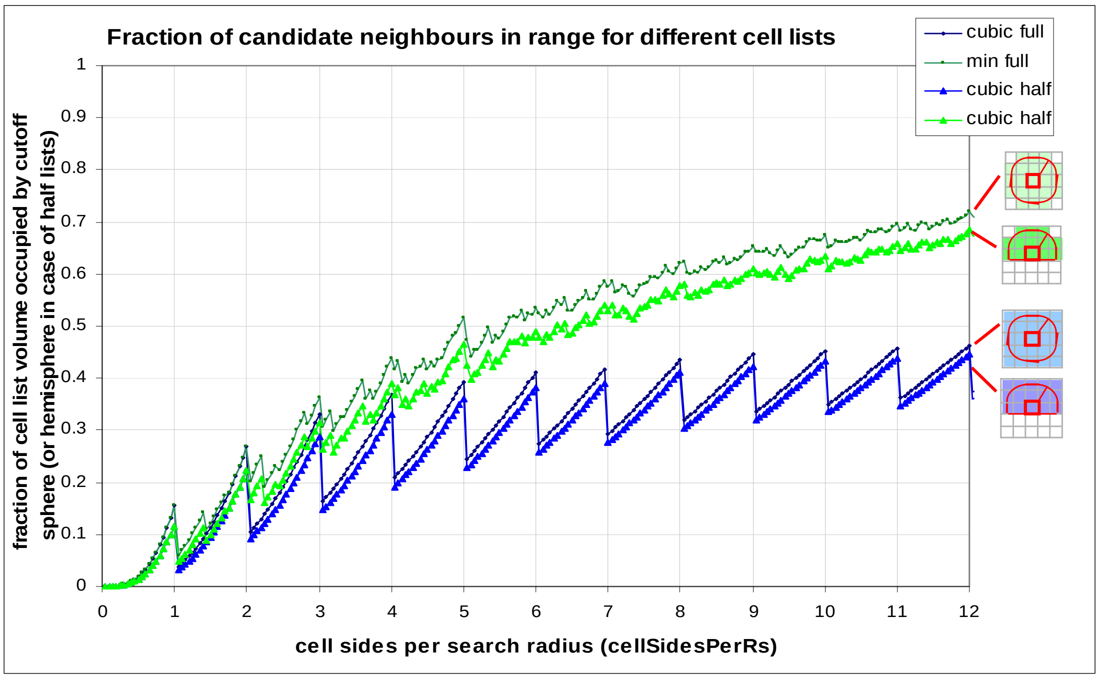


Figure 5.7. Fraction of candidates in range for cell lists.

5.5 Loaded Cell List vs. Unloaded Cell List

In the following experiment, two methods of implementing minimum cell lists (Definition 9) were compared. When considering each cell, the indexes of all other cells in range can be re-calculated (using a single minimum cell list template) for each iteration, or can be calculated and stored for each cell at the start of the simulation before the first iteration. These techniques have been labelled as "*unloaded min cell list*" and "*loaded min cell list*" respectively. Calculating each cell index is not just adding the coordinates of the cell to the cell list template value, but also checking if these values wrap around the boundaries.

Test results show a loaded minimum cell list yields much better performance than an unloaded implementation. Figure 5.8 shows that storing values can reduce cost per iteration significantly – more than halving it for *cellSidesPerRs* values of four or more. The performance saving increases roughly proportional to the number of cells in the cell list itself (Figure 5.6). However, it is worth noting that it often takes a long time to load a cell list and it can also occupy a lot of storage space. For each cell, a cell list must be loaded. Figure 5.5 shows the number of cells in a half minimum cell list. The total number of indexes that must be stored in a grid using the loaded cell list technique has a lower bound of $CPS^3 \times (4/3\ \pi\ cellSidesPerRs^3) / 2$, (if using a half minimum cell list) and therefore does not scale well for large simulations where both these values would be high.

As an example, if *cellSidesPerRs*=1 and *CPS*=11 then only 23,958 (18×11^3) indexes must be stored but if *cellSidesPerRs*=3.15 and *CPS*=35 then 9,646,875 (225×35^3) indexes must be stored. These two examples took 31 tics (half a second) and 7985 tics (2 minutes) respectively (Figure 5.8). Although initial setup cost is usually not considered in these experiments it represents a substantial time penalty (many minutes for larger simulations) and storage penalty for loaded cell lists. Furthermore, if during the simulation either r_c or the number of cells per side were changed, then the cell list would need to be reloaded, and this cost incurred again. The inability to easily and dynamically change these values is a significant disadvantage of the loaded cell list technique.

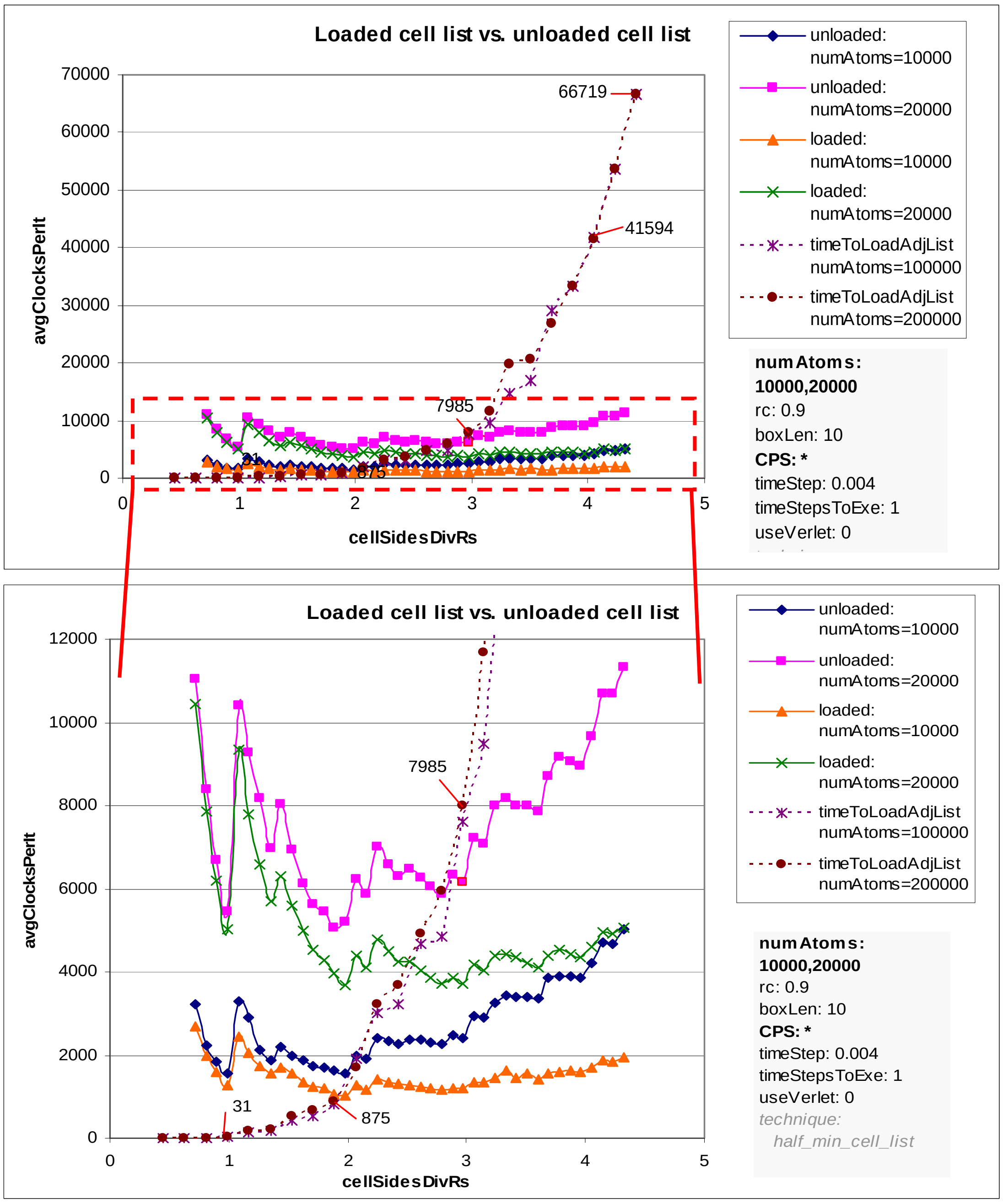


Figure 5.8. Performance improvement of storing adjacent indexes for each cell.

5.6 Atom List vs. Cell List

The following preliminary experiments were intended to show when it is preferable to use an atom list (Section 2.6), calculating which cell indexes are in a rectangular range of each atom individually, rather than a minimum cell list (Section 3.1).

As expected, results showed that for small values of *cellSidesPerRs* the half atom list performs much better than the half cell list, because only cells within a minimum bounding rectangle of each atom itself are considered (Figure 2.3), whereas, a minimum cell list always searches at least eighteen cells. All these simulations were run a number of times using a range of cell per side values, so that the optimal speed could be determined (Figure 5.9). Due to the nature of cell lists, the optimal number of cells per side for a cell list would typically be such that the length of cells was some multiple of r_c. Atom lists on the other hand exhibited a much smoother trend.

Using an optimal number of cells per side for each technique, the loaded cell list usually performed better than the atom list. However improvement margins were much less than anticipated. In some cases the loaded cell list performed worse; even though the atom list requires each cell index to be calculated and searches a cubic area, whilst the cell list searches a roughly semi-spherical area. In Figure 5.10, using 50,000 atoms and a cutoff radius about 10% of the box length, the minimum loaded cell list only produced a performance improvement of about 5.2% over the atom list.

Note that the value of r_s with respect to the box length is of critical importance. The larger the optimal *cellSidesPerRs* is likely to be, the more the shape of the minimum cell list approximates a sphere, and eliminates unnecessary cells (Figure 5.5). The optimal number of cells per side and performance difference between using a loaded cell list and atom list also fluctuated significantly if r_c and the box length remained constant but the number of atoms was changed. A more detailed analysis of optimal number of cells per side is provided in Section 5.13.

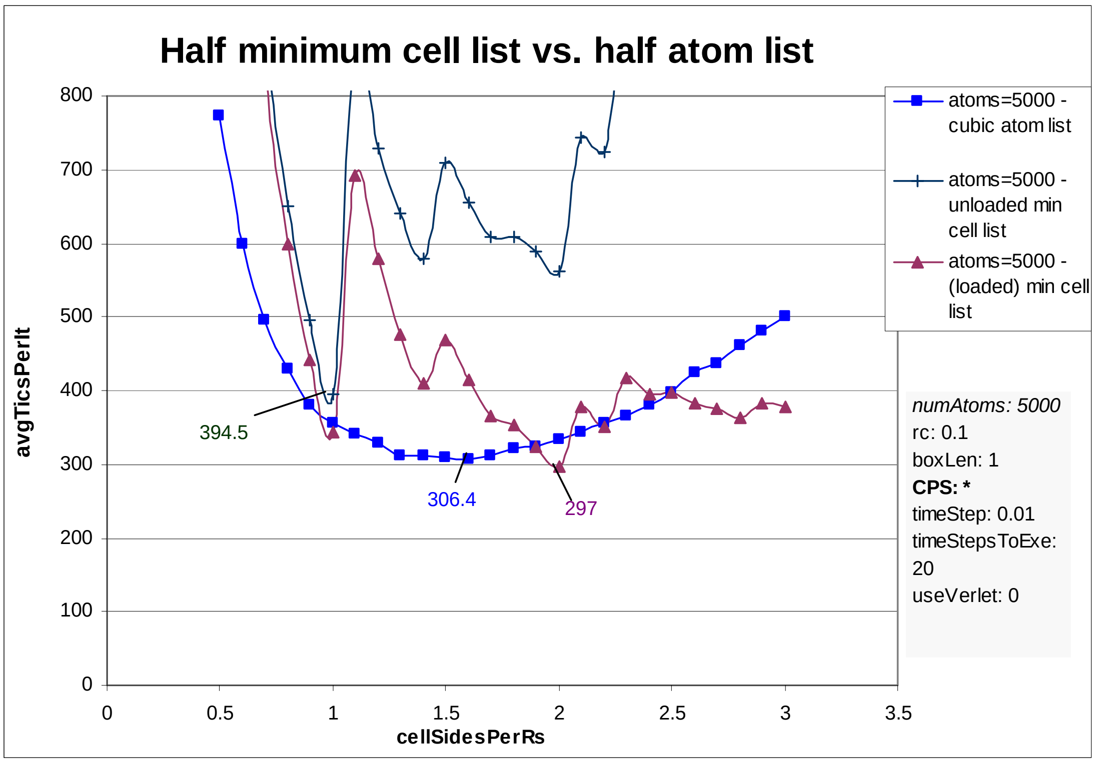


Figure 5.9. Performance of minimum cell list vs. half cubic atom list.

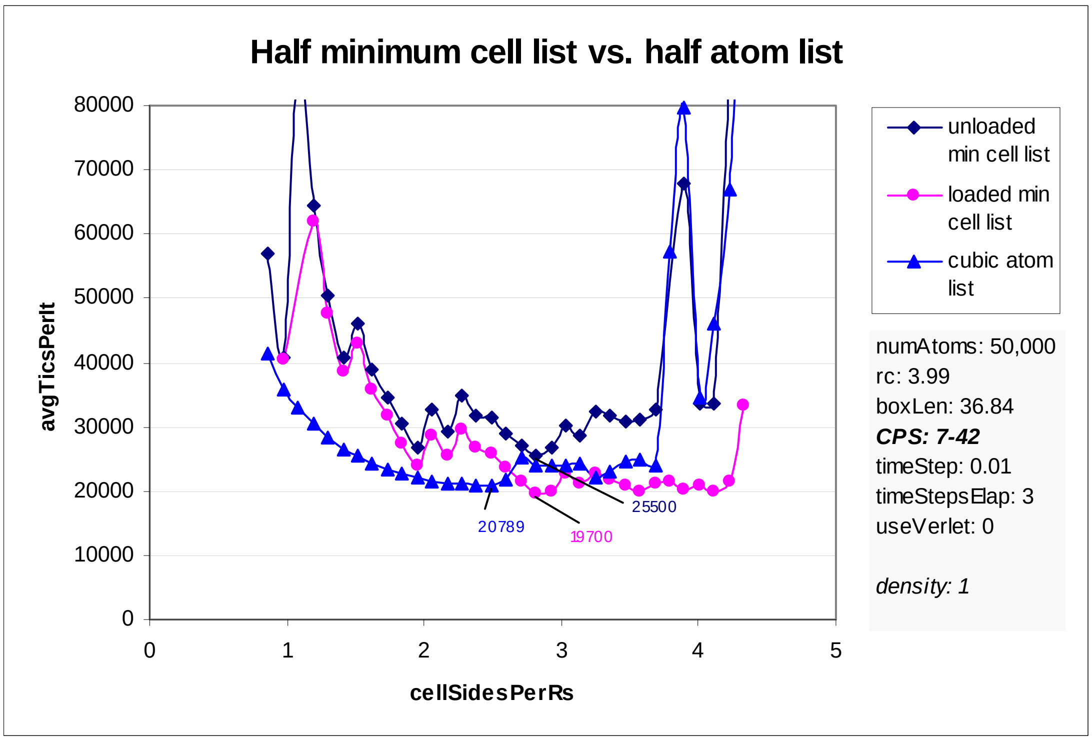


Figure 5.10. Performance of minimum cell list vs. cubic atom list.

5.7 Half Range Searches

The following experiments compare the performance of simulations using the half range search technique proposed in Section 3.2 to a normal range search technique. Each atom was given a unique integer id. For the full range search (using a minimum cell list), candidate neighbour atoms were rejected if their id was greater than the atom being considered. For the half range search (Definition 10), no cells to the left of the root cell were searched, and candidate neighbours atoms were immediately rejected if they had a smaller x coordinate than the atom in question. If both atoms had the same x coordinate, atom id was used as a tiebreaker; the neighbour atom was rejected if it had a greater atom id. Both techniques ensured each neighbour pair was captured only once.

Figure 5.11 shows the performance of a cubic atom list using full range search and half range search techniques. In this particular simulation, the half range search clearly outperforms a full range search technique – using optimal performance the full list is about 65% slower in this case. Notice the full range search attains its best performance with a lower number of cells per side than the half range search. This was consistent with most collected results for cell lists. As *cellSidesPerRs* increases so too does the performance difference between full and half range search. A higher *cellSidesPerRs* value means more cells outside of the atom's cell (in every direction) will be captured during each search, and the number of cells captured by a full range search will slowly approach double that captured by the half range search. The difference in the number of cells between the half and full cell lists dictated improvements (Figure 5.6).

Figure 5.12 shows similar results, but using a minimum cell list instead of atom list. Improvements of this technique for cell lists proved to be slightly less than improvements for the atom list, but still significant – using optimal performance, the full list is about 25% slower in this simulation.

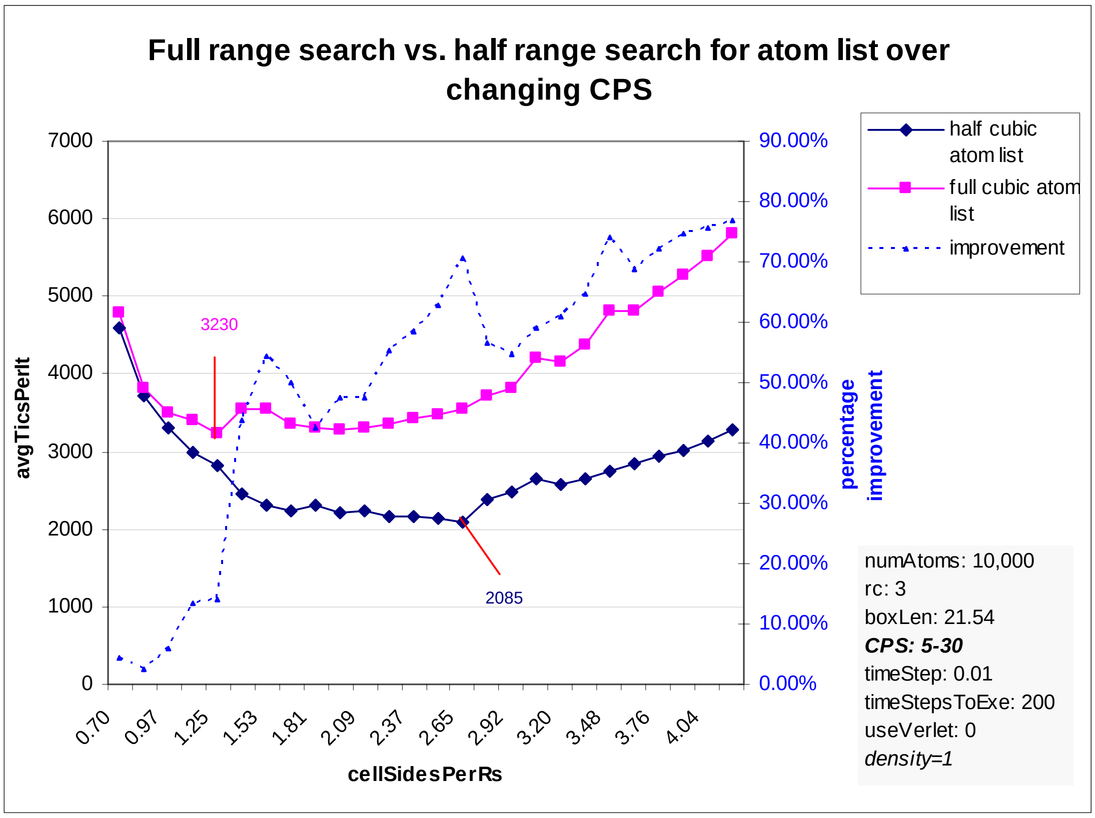


Figure 5.11. Performance of full and half range search using atom lists.

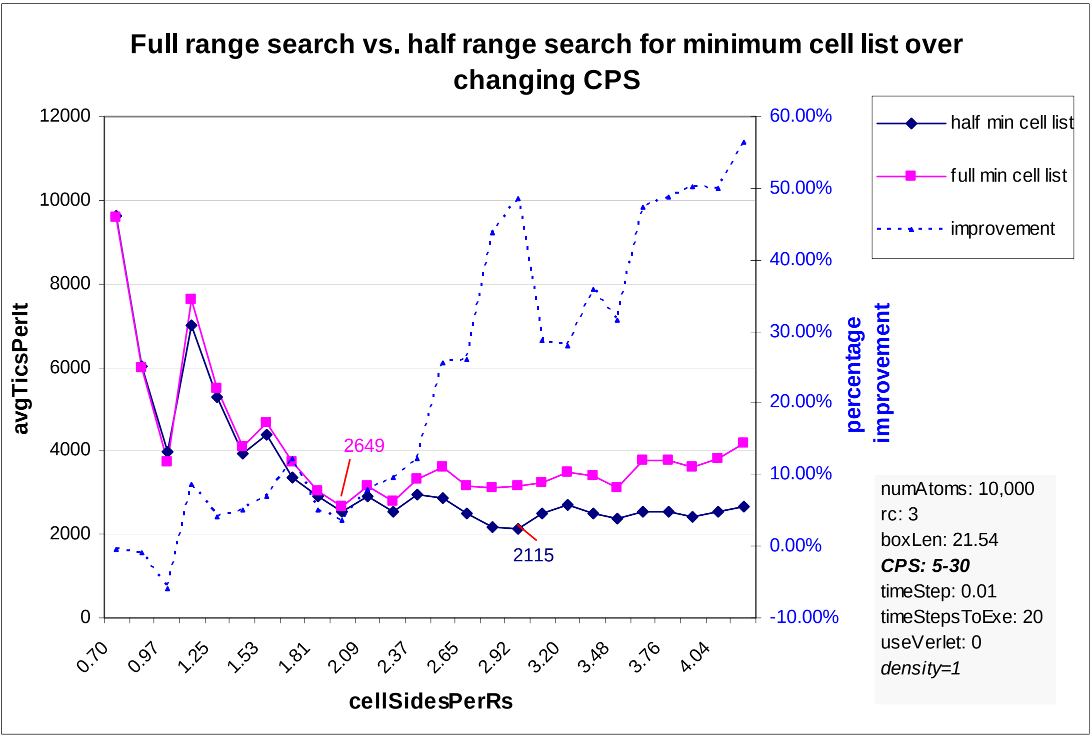


Figure 5.12. Performance of full and half range search using minimum cell lists.

5.8 *Early Elimination of Non-Neighbours*

Figure 5.13 shows performance of using the different early elimination techniques described in Section 3.3 using a half minimum cell list. Results show that using early elimination (checking the distance between each atom along each axis is less than the cutoff, before calculating total distance) gave negligible performance improvements, and in some cases cost more. Although it was proved a high fraction of candidate neighbours are subject to early elimination, it was not worth the cost of checking for early elimination, implemented in code. Avoiding the use of square root gave an efficiency increase of up to approximately 3%. The author recommends that the early elimination technique is not worthwhile, especially if using a half cell list. However, avoiding square root unless/until required is still recommended.

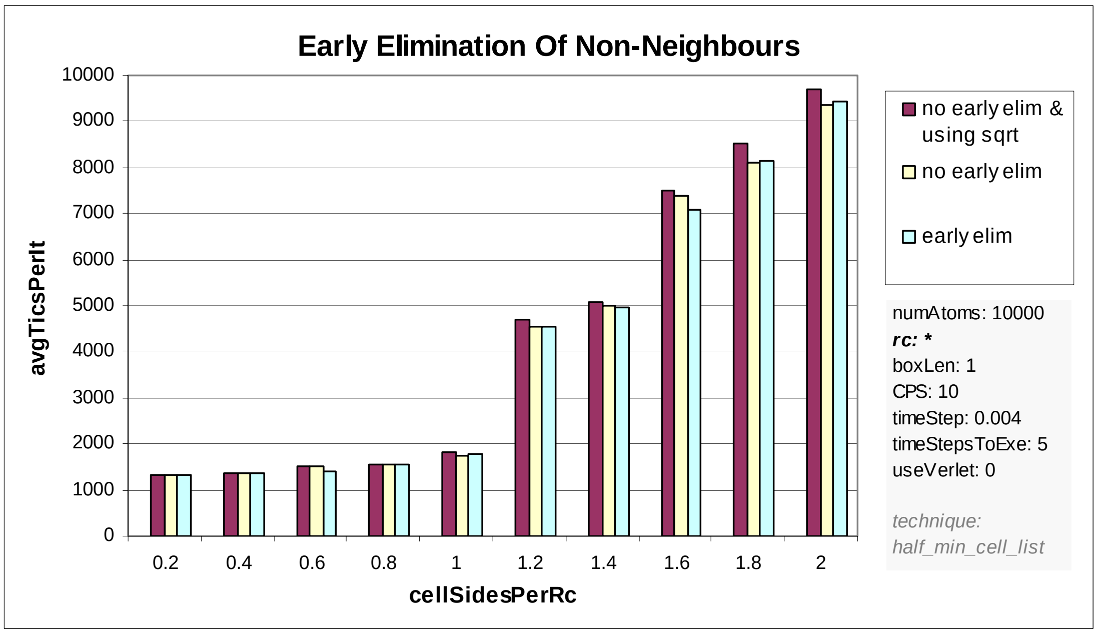


Figure 5.13. Performance improvement of early elimination and using distance squared.

5.9 *Sub-grids and Cell List Template Guides*

The sub-grid technique described in Section 3.4 was implemented as follows. A half minimum cell list template was generated and used to calculate and store the indexes of the corresponding adjacent cells of each cell. A "sub-grid template guide" was generated such that each sub-cell identified which cells in the cell list template were not in range, using a vector of integers (since this is more efficient than using a vector of booleans in C++ [20]).

For each half range search, the atom was placed into a sub-cell, and only cells for which the matching integer value was true were searched.

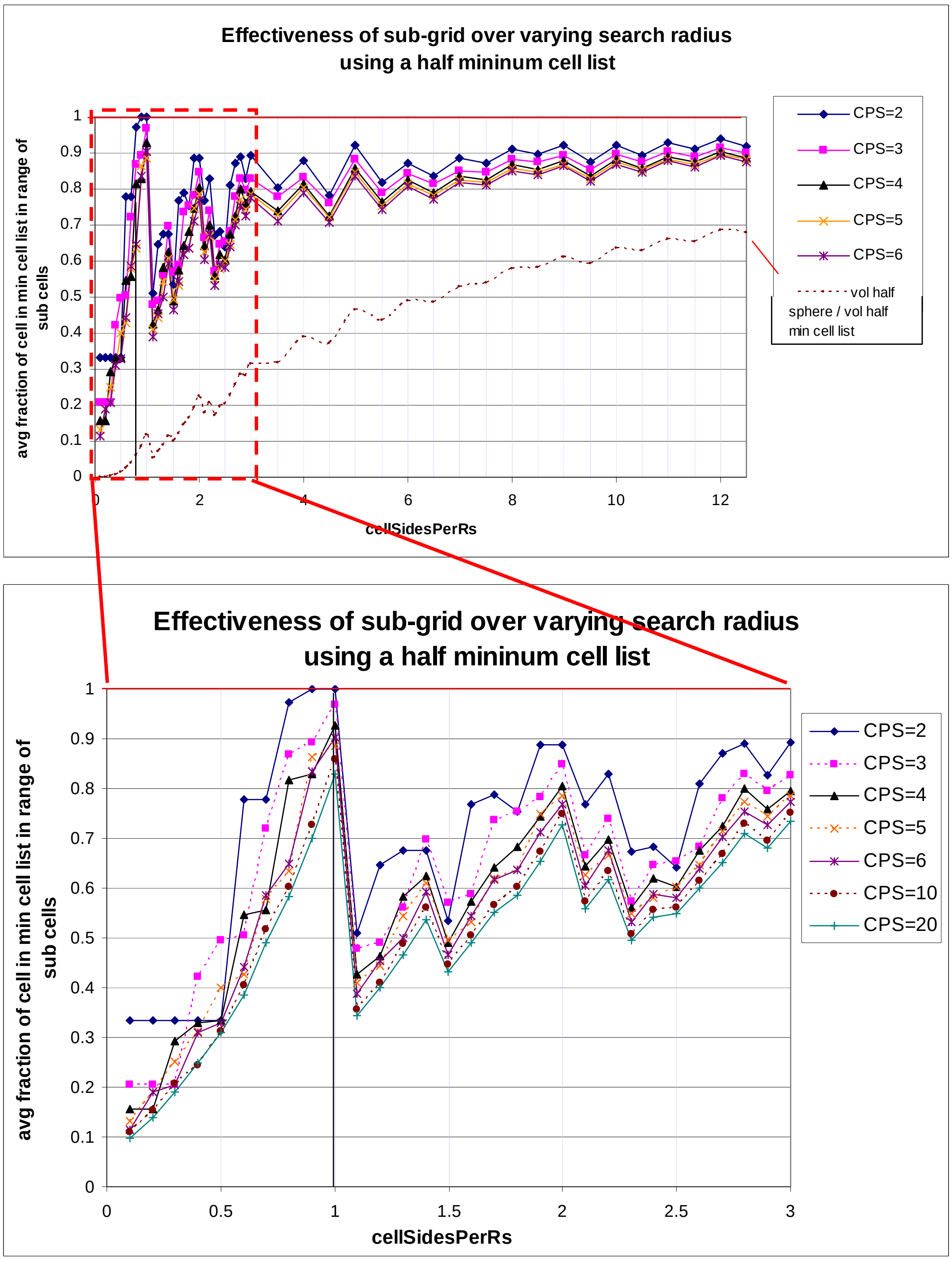


Figure 5.14. Sub-grid effectiveness over varying *cellSidesPerRs*.

Figure 5.14 shows the average fraction of cells in a half minimum cell list which are included for randomly placed points. Results for several sub-grids were generated whereby each sub-grid used a different number of cells per side. These results show that the sub-grid should be most beneficial where the cell length is less than r_c; which can often happen in MD simulations where the length of the box will not divide evenly into r_c. However if r_c does divide evenly, or almost evenly, into cell length, this approach in unlikely to be effective. Using four to six cells per side in the sub-grid appears sensible, since any more than this only gives negligible improvement, but will cost exponentially more storage. Note that the amount of storage will be at least *(number cells in cell list × number of cells per side in the subgrid^3)* booleans. For example, a grid with a *cellSidesPerRs* value of 4, and 20 cells per side for the sub-grid, would require approximately 4 million (514×20^3) integers.

Figure 5.15 illustrates how the sub-grid technique has the effect of smoothing bumps in performance of an ordinary loaded cell list when the number of cells in the cell list steps up a notch. Figure 5.16 shows that, using a sub-grid with just 4 cells per side can be quite effective in these molecular dynamics simulations, but only for systems with over 3 atoms per cell. Notice that results are poor when *cellSidesPerRs* is equal to one (Figure 5.14).

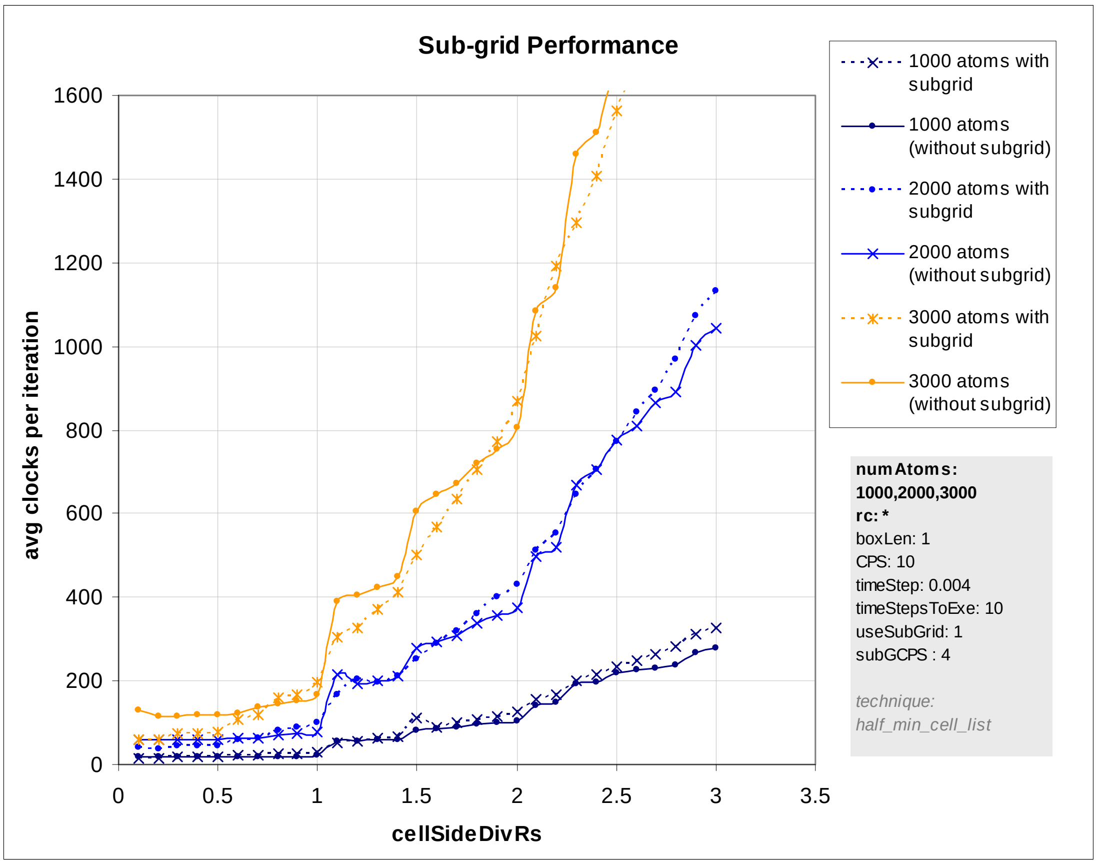


Figure 5.15. Sub-grid technique performance.

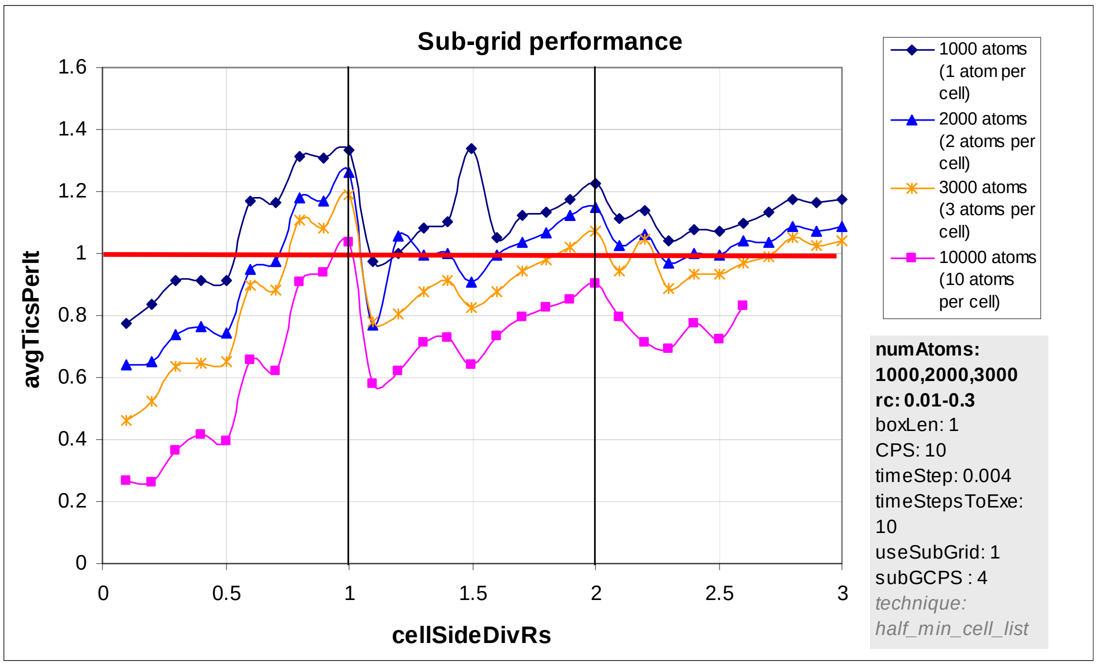


Figure 5.16. Sub-grid technique improvements for different numbers of atoms per cell.

This author would recommend experimenting with the sub-grid technique if a cell list is already established, there is a high number of atoms per cell and *cellSidesPerRs* is not ideal (Figure 5.14). However, it is later shown that the optimal grid usually has a low number of atoms per cell, making it advisable to avoid this idea altogether. This sub-grid approach could work well in an application such as a computer game, whereby a fixed master grid is already set up, but a series of small range searches with a relatively small fixed radius (eg: less than half the cell length) are required.

5.10 Minimum Bounding Rectangles in Cells

Figure 5.17 shows that, for 1000 atoms, using MBRs in each cell was less efficient than using minimum atom lists. Although, MBRs would occasionally allow the skipping of a cell which was in range of the minimum atom list, the cost of building MBRs for each cell was greater.

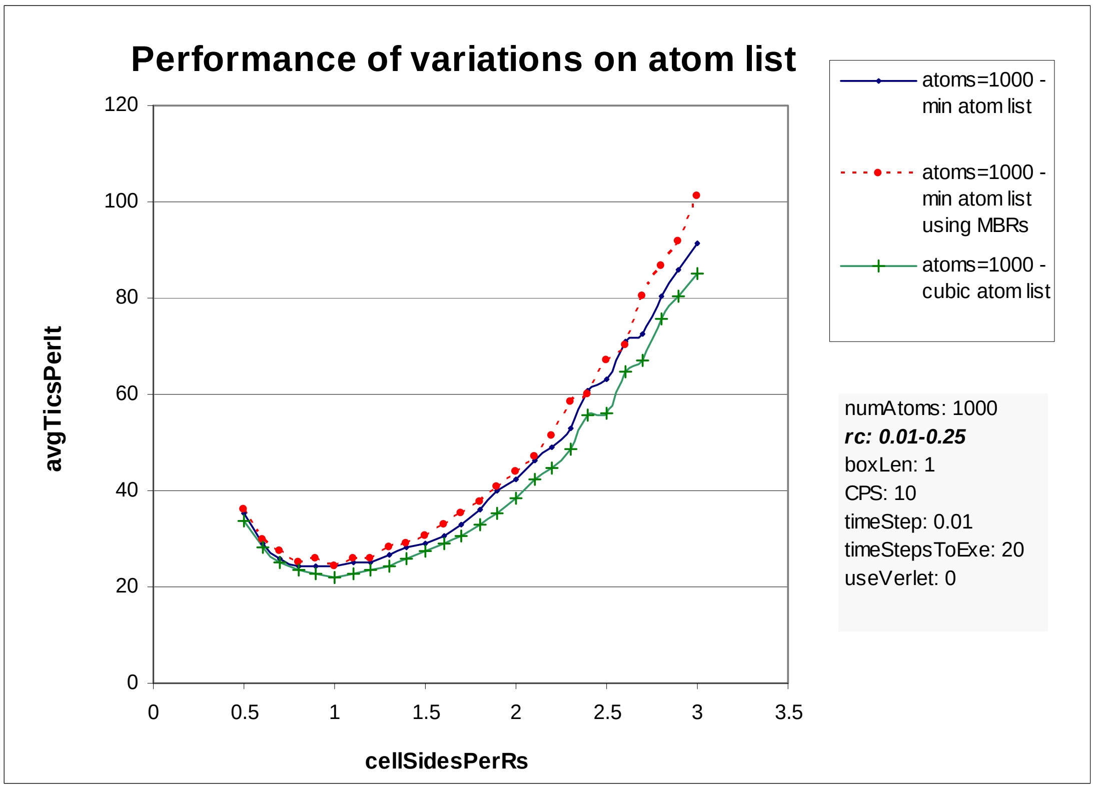


Figure 5.17. Performance of using MBRs and minimum atom list vs. cubic atom list.

More unexpected, however, was that a cubic atom list performed better than a minimum atom list. In a minimum atom list, for each atom searched, each cell in the cubic range is first tested to see if it is in range of the atom. The computational savings of using a minimum atom list over a cell list can be approximated as follows:

$$\begin{aligned} \textit{comp savings} = \{&(\textit{avgAtomsPerCell} \times \textit{time to check candidate pair} \times \textit{fraction cells eliminated}) \\ &- \textit{time to check cell against atom}\} \times \textit{numAtoms} \end{aligned}$$

Checking a cell is outside of r_c of an atom is relatively inexpensive. Nevertheless, if only a small proportion of cells in the cubic atom list fall outside r_c and there are not many atoms per cell, then this cost can outweigh the potential savings.

Figure 5.18 shows the average fraction of cells in a cubic atom list which are omitted in the case of a minimum atom list. For *cellSidesPerRs*=1 this is about 20%. However, since most optimal simulations have only between 1 and 2 atoms per cell, it is cheaper to check these atoms against the root atom, rather than each cell. Checking the distance between two atoms is always cheaper than checking the minimum distance between an atom and a cell or MBR, because the latter requires determining which side of the box is closest to the atom along each dimension [27].

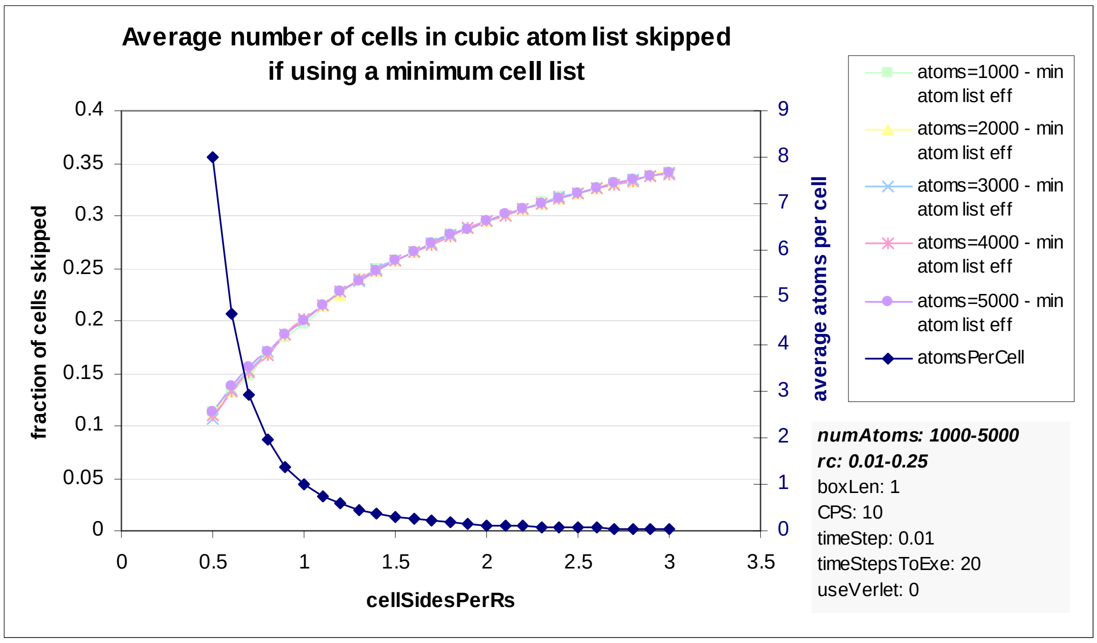


Figure 5.18. Fraction of cells in cubic atom list skipped when using minimum atom list.

It is reasoned that, if a skewed data set were used, the number of atoms in each cell could be used to decide how to process that cell. There would be a large number of atoms in certain cells and none in others (Figure 3.5). If the number of atoms in a candidate cell is above some threshold, it would be worthwhile to check if that cell was in range of the atom being considered, because if the cell is out of range checking all atoms individually could be avoided. If the number of atoms in the candidate cell was below the set threshold, checking the few atoms individually would work out cheaper.

It should also be mentioned that the self-spatial join algorithm was implemented such that, each cell was visited, and the range search executed for each atom in that cell. However, if a high proportion of cells are empty, as is the case for skewed distribution, iterating over these cells would result in wasted processor time. In this case it may be faster to execute range searches for each atom in the order in which it appears in the atom array, despite the fact searching atoms in order of cells exhibits better spatial locality.

5.11 Improving Spatial Locality: Single Atom Object vs. Separate Position Array

An important implementation decision was whether to store location of particles in a separate array from other atom attributes. Storing locations separately means more points should fit into cache at a time, therefore it seems reasonable that cache hits would improve while building neighbours lists. However, results showed that separating the positions resulted in worse overall CPU performance (Figure 5.19). Seemingly, the step of moving atoms, which involves adjusting each atom's position based on accumulated force and velocity, would result in many cache misses. If molecular dynamics testing functions were added to this simulation, this cost would become even greater, because it is likely that a single atom's position and other attributes would be accessed in close succession. Having a single particle object, containing (x,y,z) position, velocity, forces and perhaps a few other attributes, is not only more object-oriented and easier to code, but exhibits the best spatial locality.

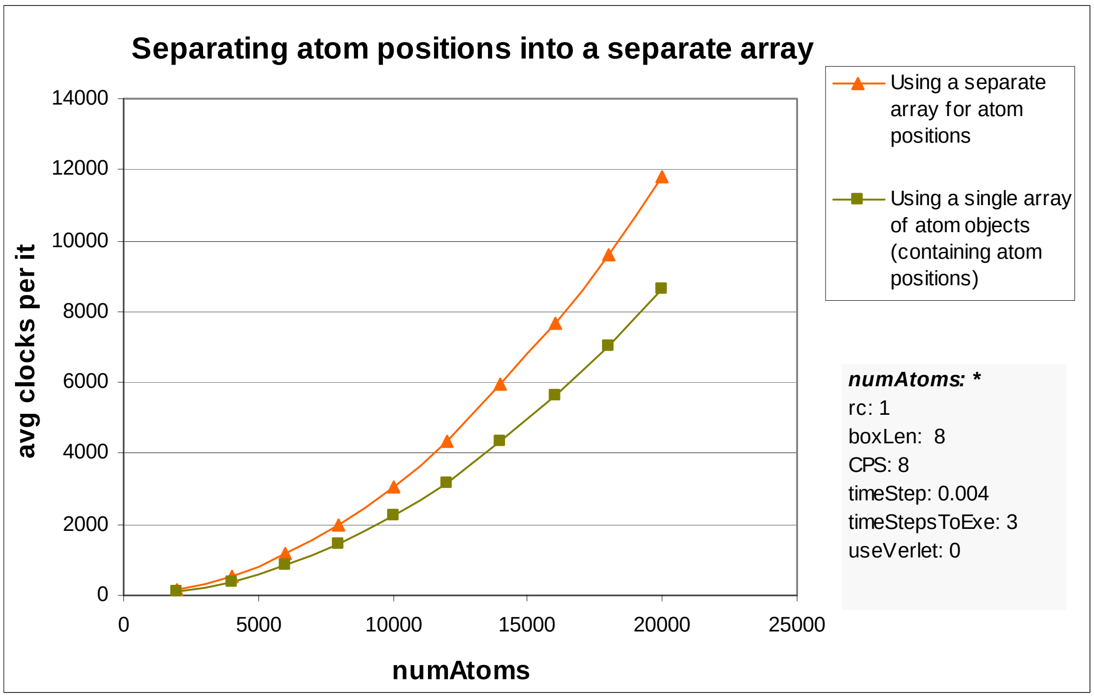


Figure 5.19. Result of keeping atom location in a separate array.

5.12 *Improving Spatial Locality using Space-filling Curves*

This section tests the performance of using several of the space-filling curves described in Section 2.10. A space-filling curve only needs to be generated once to establish an order for the grid cells. Ideally, the number of cells per side is some practical value of 2^n, or else the pattern of the curve must be broken. Figure 5.20 shows the performance improvements of Hilbert curve and Z-order curve compared to random atom distribution. The cost of reordering the points is not included in this set of results. For up to 20,000 atoms, the improvements were low – compared to a totally random order, Z-order performed 2.32% better and Hilbert curve performed 2.45% better on average.

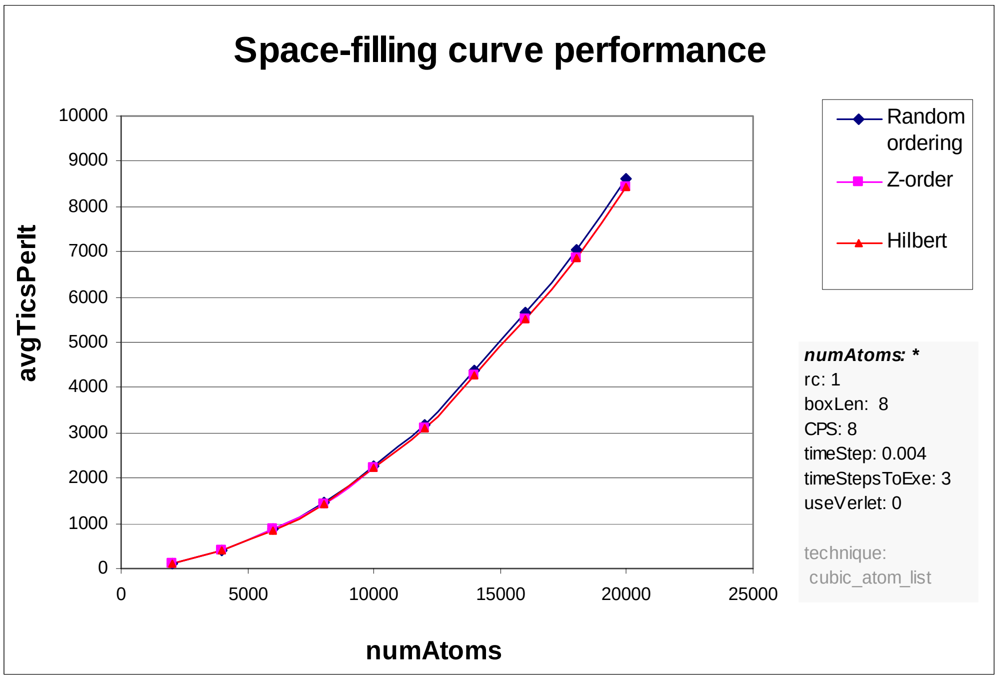


Figure 5.20. Ordering atoms using space-filling curves vs. random ordering.

Later tests showed that when using much larger numbers of atoms, more than about 200,000 atoms, performance improvements gained using space filling curves were dramatic. Compared to a random order, Z-order performed almost twice as fast and row ordering performed almost as well. It is conjectured that this sizeable difference in performance was due to virtual paging in the virtual memory system when simulations became too large to fit completely in main memory. Unfortunately, it was also found that, in order to get accurate results for these larger atom sets, numerous iterations were required, and to collect a more extensive set of data would have taken many days.

The above simulations only measure time to build the neighbour list, and do not include the time to re-order the atom list using the curve. Furthermore, the velocity of each particle was set to almost zero so that atoms were not likely to float out of their original cells during the simulation. If the velocity was higher, atoms would move from one cell to the next, and the performance improvements would reduce faster because the atoms would quickly become out of order again (Section 2.10). Note that, if a verlet neighbour list was used, rebuilds would not occur every iteration. This means particles would move much further between rebuilds of the neighbour list (many will have moved by as much as $r_v - r_c/2$) therefore performance improvements would degrade rapidly. This would make it necessary to re-order the atom list very frequently – possibly forcing a rebuild every second rebuild – in order to prevent the order of atoms in the atom array degrading back to a random ordering. Determining the best moments to reorder points in order to optimise performance is non-trivial, since the re-ordering itself also bears a cost. Ultimately, the best time to reorder points will depend on the velocity and pattern of movement of atoms. One possible method is to establish a threshold value such that, when performance falls below this value, the list of atoms is re-ordered.

Two methods of re-ordering the atoms array were tested. The first simply made a temporary copy of the entire atom array, and then copied atoms from the temporary copy back into the original array, following the order of cells specified by the space-filling curve. The second method used much less space, by first determining the order of atoms, and then selectively swapping atoms in and out, until the entire array was in order. Results showed the first method was almost twice as fast.

The time to re-order atoms scales linearly with the number of atoms. It was initially thought this cost would be substantial; but results showed that re-ordering only took approximately 254 clock tics for 1 million atoms, and approximately 2.2 clock tics to re-order 10,000 atoms; negligible compared to the total cost of each iteration. To put this in perspective, in the simulation with 10,000 atoms in Figure 5.20, the performance improvement between random ordering and Hilbert curve ordering was approximately 50 clock tics (2520 clock tics per iteration × 2.45%). Note also that cost of iteration goes up about linearly with the average number of neighbours per atom, and this value was only 4 (unrealistically low for MD simulations) in the previous experiment. For such a low cost, re-ordering of atoms could be done every time the neighbours list is rebuilt, but for a 2-5% performance increase, this author believes that it is not worth the extra code and complexity which is required.

5.13 Choosing Optimal Cells per Side

One of the most effective ways to optimise the building of neighbour lists is to find the optimal number of cells per side in the fixed grid. It was discovered that finding an optimal value was deceptively non-trivial, and may vary significantly from one machine or compiler to the next. Initially it was thought that setting up the grid such that there would be a certain number of atoms per cell would help performance. Figure 5.21 shows results from a batch of simulations using a cell list, whereby r_c and the box length stayed constant. The optimal number of average atoms per cell appears to increase almost linearly as the density of atoms increases.

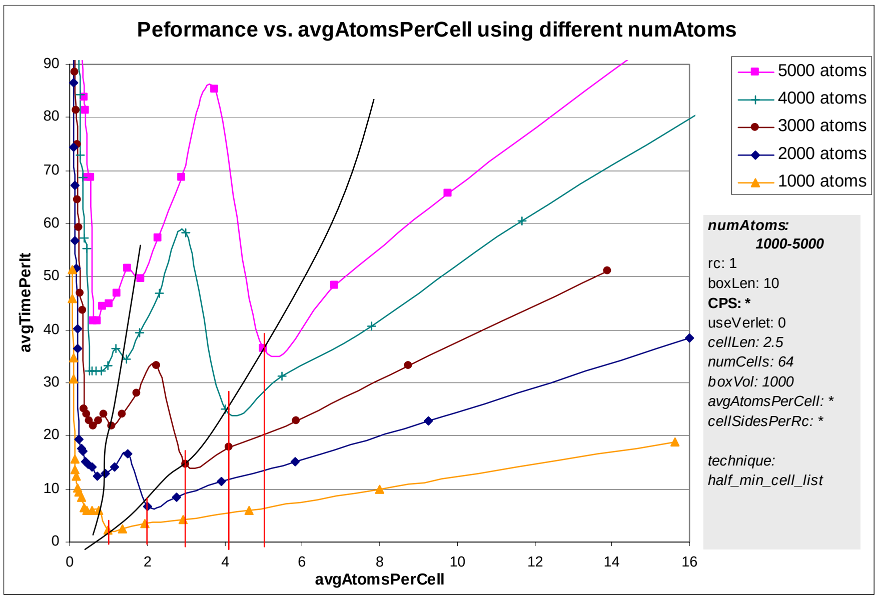


Figure 5.21. Optimal atoms per cell for a cell list vs. average atoms per cell.

In the case where a cell list is used, *cellSidesPerRs* is a crucial value to measure, since it dictates the volume of each search. Figure 5.22 shows the same results in Figure 5.21 graphed against *cellSidesPerRs*. As the density of atoms increases, the optimal value of *cellSidesPerRs* shifts from 1 towards 2.

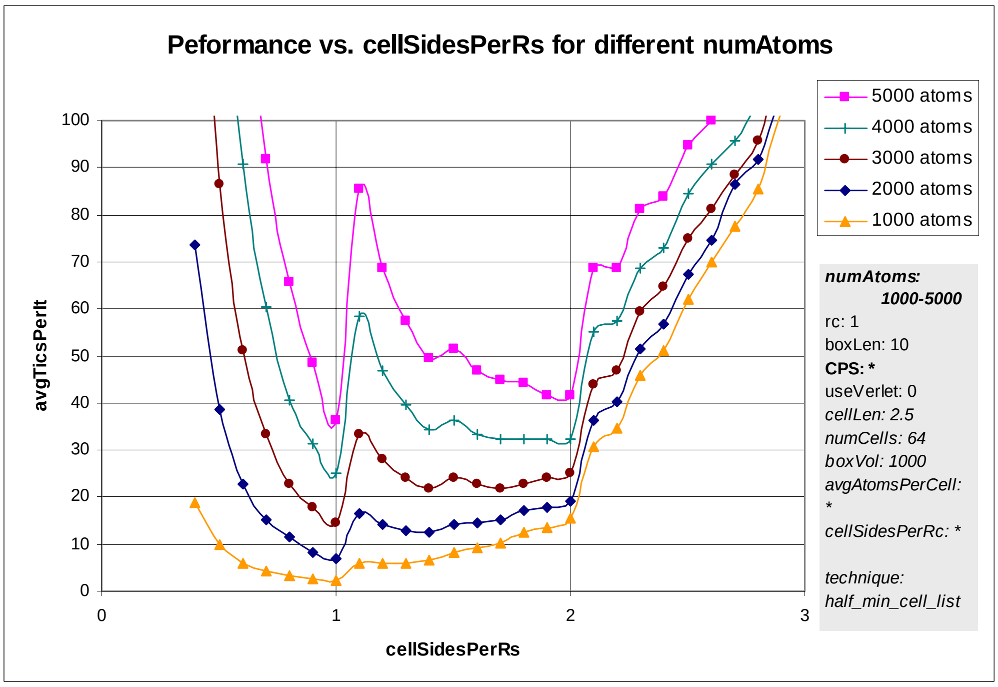


Figure 5.22. Performance costs vs. *cellSidesPerRs* using cell list.

Using an atom list instead of a cell list yielded much different results. Figure 5.23 shows that the optimal number of cells per side for a given simulation for an atom list is much more dependent on the number of atoms than the cell list (Figure 5.21). This shows that by setting the number of cells per side so that the average number of atoms per cell is just above one achieves a near-optimal value, for each of these simulations.

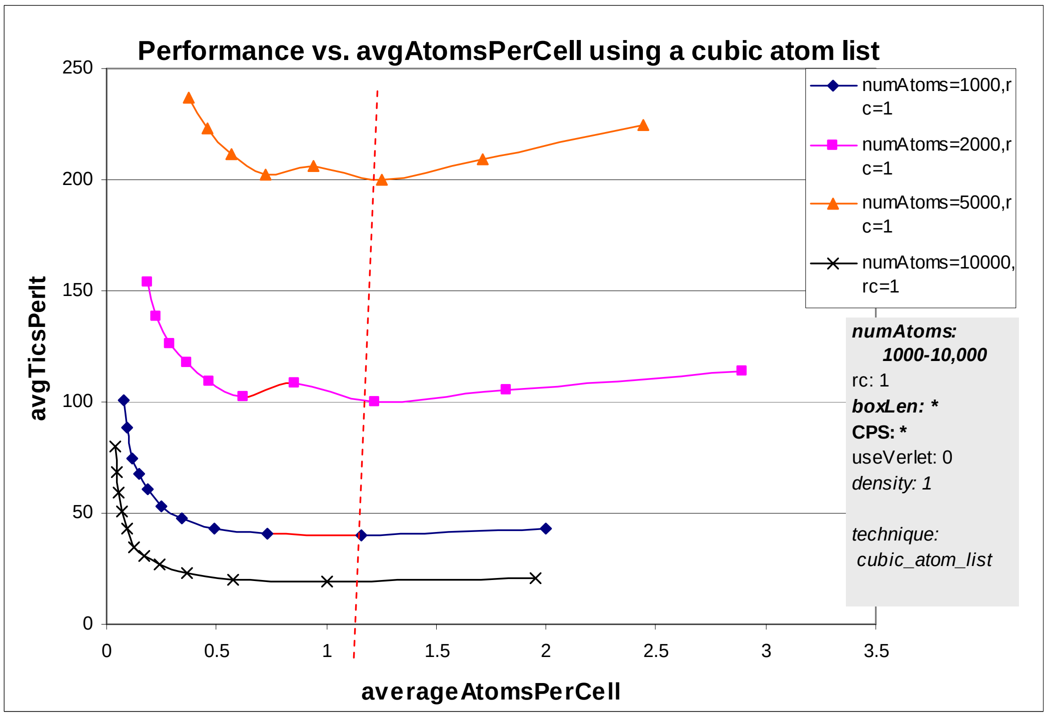


Figure 5.23. Performance vs. average atoms per cell using a cubic atom list.

As observed in previous sections, the performance trend for an atom list is much smoother than for a cell list, and unlike the cell list, there is no penalty to change the number of cells per side dynamically. For this reason, it may be beneficial for long simulations to change the number of cells per side until the optimal value is found.

A simple algorithm was proposed to dynamically find the optimal cells per side for any given simulation. At the start of the simulation, a decent first estimate of cell per side is set such that it obtains (approximately) some default number of atoms per cell.

$$\mathit{cellsPerSide} = \left\lfloor \sqrt[3]{\text{numAtoms} \div \text{desiredAtomsPerCell}} \right\rfloor$$

Values of between 1 and 2 atoms per cell worked best in the tested simulations. As the simulation executes, the number of clock tics elapsed during each rebuild of the neighbours list was recorded. Only the algorithms responsible for placing atoms in cells and generating the neighbours list were timed. After each rebuild the performance of the rebuild just executed and the rebuild before that were compared, and then the number of cells per side was incremented or decremented by one, depending on the result. If an improvement was found, the number of cells per side was changed in the same direction as the previous change; otherwise, it was changed in the opposite direction. When the number of cells reaches an optimal value, the cells per side fluctuates up and back from that value, and when this occurs it was assumed that the optimal number of cells per side had been found, and stayed fixed at that value for the remainder of the simulation.

This worked well for large simulations, but for simulations with a small total number of neighbours per iteration it was found that each rebuild only took a few tics, so accuracy was poor. The solution implemented was simply to group together several rebuilds and calculate an average number of tics. Rebuilds were only grouped and cells per side changed when the total number of tics for rebuilds since the last change exceeded some specified threshold.

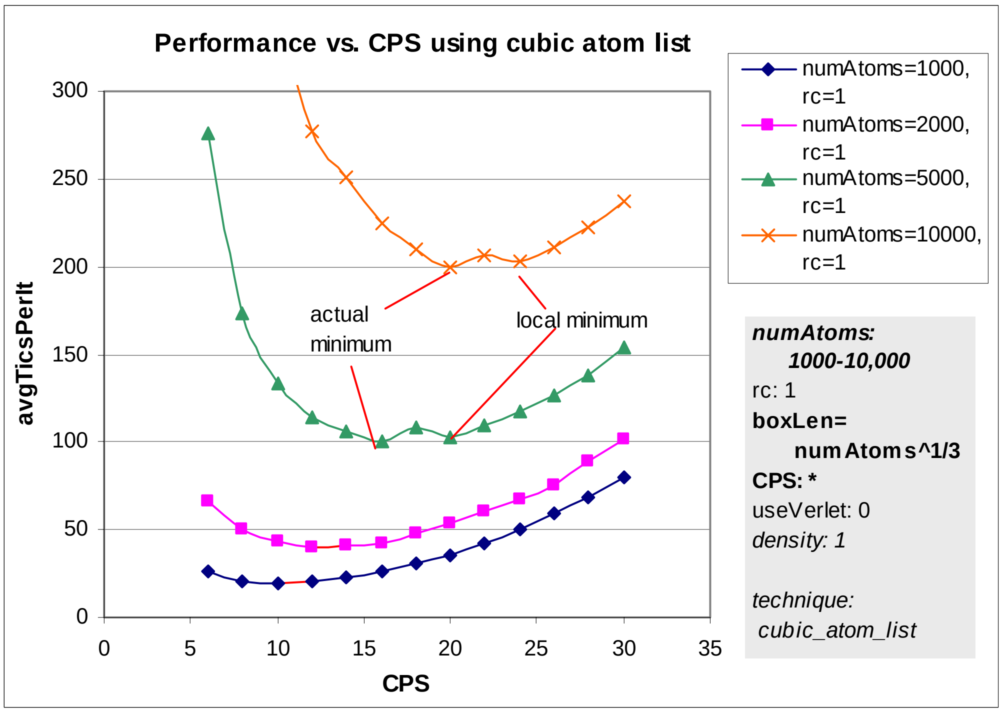


Figure 5.24. Finding the optimal cells per side using a cubic atom list.

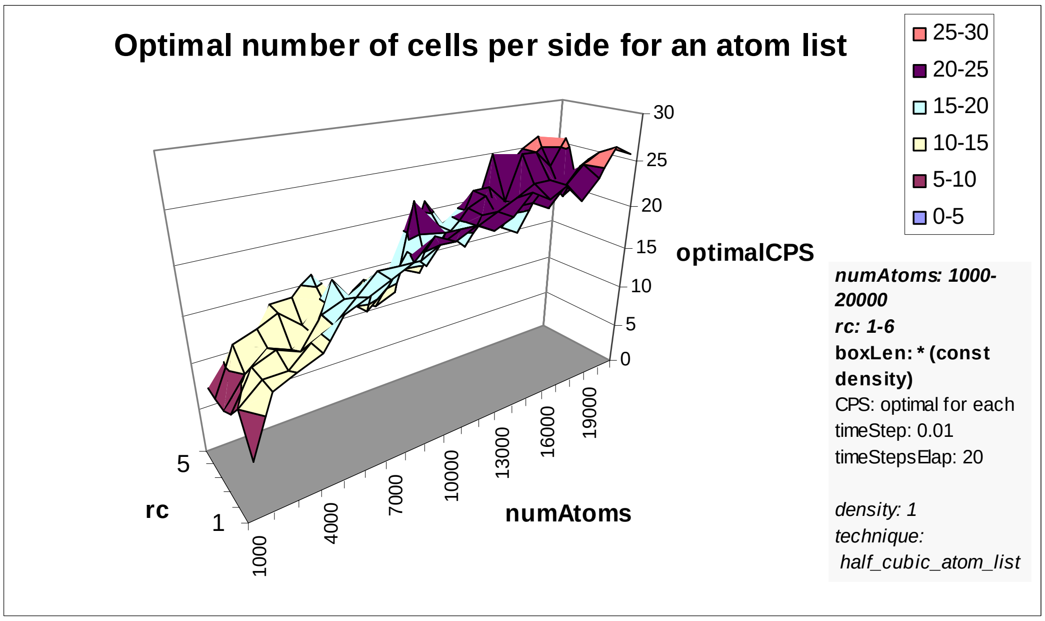


Figure 5.25. Optimal number of *CPS* for varying number of atoms and cutoff radius.

However, even with high accuracy this scheme did not necessarily work perfectly. As demonstrated in Figure 5.24, a local maximum and minimum often formed a few increments away from the actual minimum, and it is not fully understood why this is. If the algorithm were to start on the wrong side of the local maximum and single increments were used, this local minimum would be chosen as the optimal cells per side, instead of the actual minimum. For this reason, choosing a starting point carefully is important, and it was found that choosing about 1.5 atoms per side and accuracy threshold of several hundred tics, the algorithm rarely found an incorrect optimal value.

Figure 5.25 shows the optimal number of cells per side has little dependence on the cutoff radius. This set of results was generated using the optimal number of cells per side finding algorithm described in this section. As expected, the optimal number of cells per side increases roughly logarithmically as the number of atoms increases.

5.14 *Choosing Optimal Verlet Radius*

A larger verlet radius means atoms must travel a larger distance and therefore a longer time before the neighbours list must be rebuilt (Figure 2.5). However, a larger verlet radius also means the neighbours list will be longer and therefore each update will require checking more neighbours, which is more expensive. Finding the optimal verlet radius is ultimately dependent on finding an ideal balance between these two component expenses. In order to test the performance of verlet radius, simulations were set up where all atoms had the same constant fixed velocity, and the verlet radius was changed.

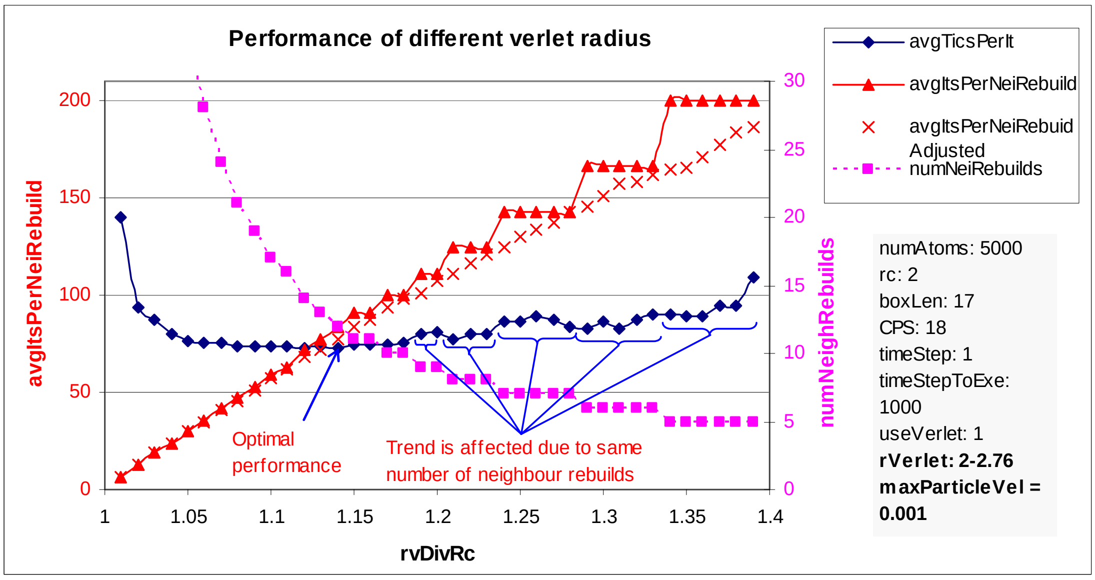


Figure 5.26. Verlet performance with respect to finite number of rebuilds.

Figure 5.26 demonstrates that, whenever a small finite number of iterations occur, even if this is as high as 1000, large anomalies can occur in the performance trend. This is due to the fact that the same simulation run with two different verlet radii might both undergo the same number of neighbour list rebuilds, but one may stop just after a rebuild is executed, and the other (with a slightly smaller verlet radius) may stop before a rebuild is needed. Figure 5.26

illustrates these relationships. To allow more accurate results to be obtained, one method used was to allow a fixed number of rebuilds to occur such that the simulation was stopped just after it has been determined that another rebuild was needed. This method allows the average number of iterations per rebuild to more accurately reflect results if the simulation were run over an infinite timeframe, and therefore was used in many of the following result sets.

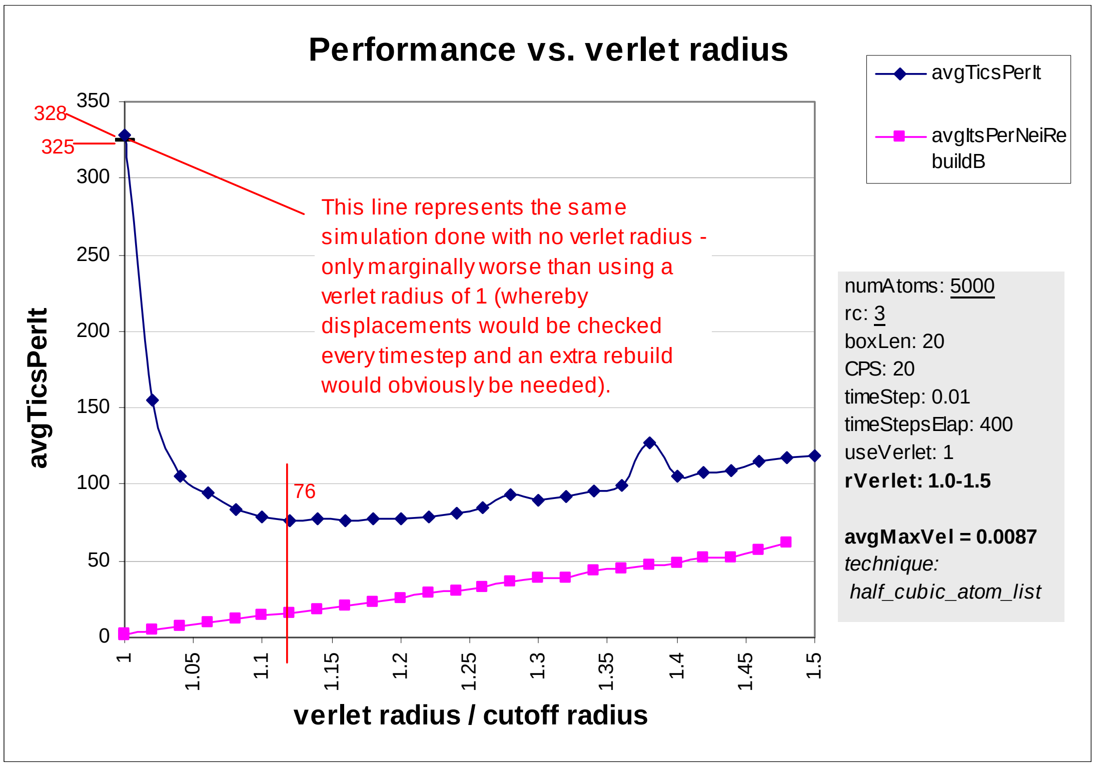


Figure 5.27. Performance of using different verlet radius.

Figure 5.27 shows that the performance of simulations exhibits a smooth trend as the verlet radius is increased. In this simulation the optimal verlet radius is approximately 1.12 times the cutoff radius and is over 4 times faster than executing the simulation without using a verlet radius.

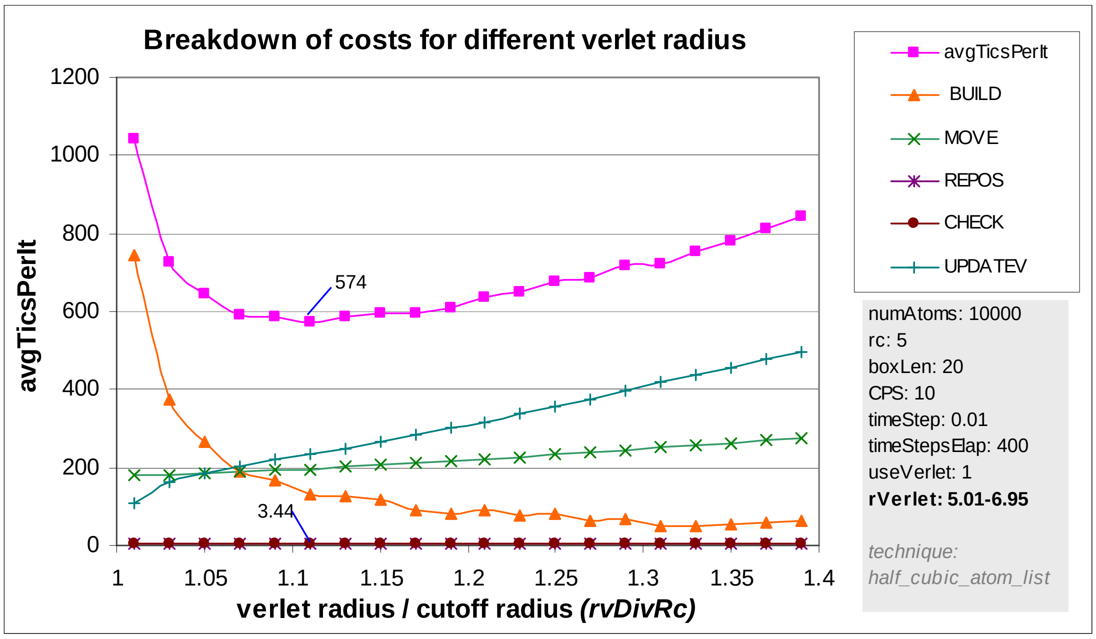


Figure 5.28. Performance of using different verlet radius.

Figure 5.28 shows similar results for a simulation with 10000 atoms, and also shows the breakdown of total costs per iteration as the verlet radius is increased. As *rvDivRc* and therefore the number of iterations between rebuilds increase linearly, the total number and cost of rebuilding decreases rapidly, and the cost of updating the verlet list increases linearly. Significantly, the cost of checking atom displacements to check if the verlet list needs updating (Section 3.6) is negligible; only 0.6% of the total cost per iteration for the optimal verlet radius in the simulation shown.

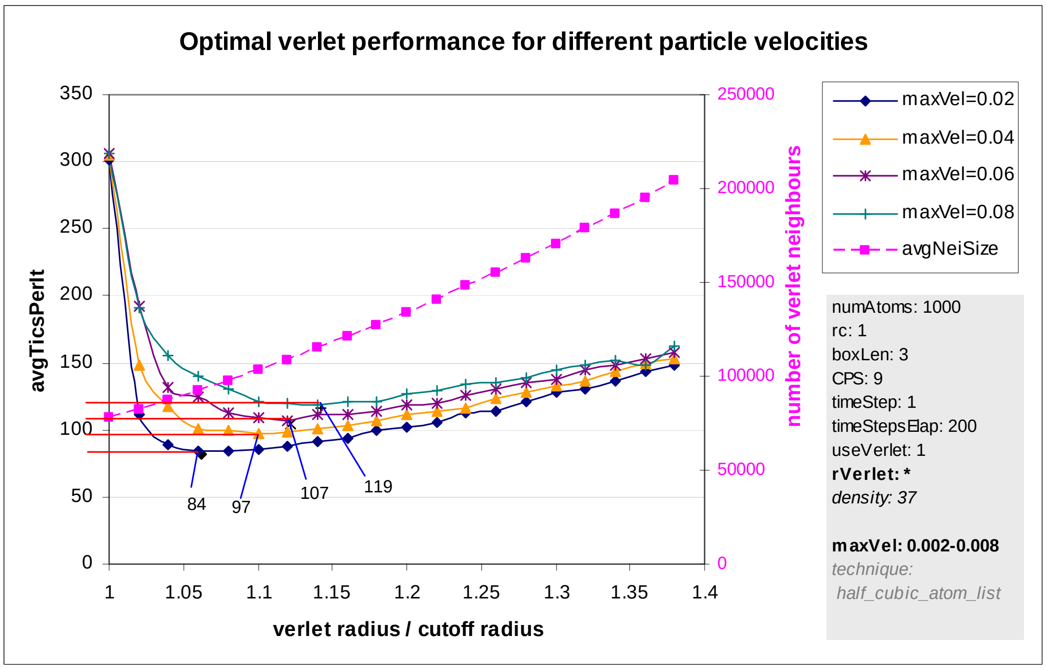


Figure 5.29. Optimal performances for different particle velocities.

Figure 5.29 shows how the optimal verlet radius and performance improvements are dependent on the maximum velocity of particles each timestep. The faster particles move, the worse the optimal performance and greater the optimal verlet radius.

Since the trend of performance against *rcDivRc* was smooth in all recorded results, a similar algorithm to the one described in Section 5.13 was proposed to find the optimal verlet radius in any given simulation. At the beginning of the simulation, the initial starting value of r_v was set to 1.2 times r_c, since this appears to be a common choice for a verlet radius [11]. A more sophisticated scheme might also take the starting velocities of particles into account.

The number of clock tics elapsed during iteration was recorded, for both rebuilds and verlet updates. Prior to each neighbour rebuild, the average tics per iteration starting from the previous rebuild was calculated as the number of tics for the previous rebuild plus all subsequent verlet updates, divided by the number of iterations this includes.

$$\textbf{i.e.:}\ \ avgTicsPerItSince\,\mathrm{Re}\,b \ = \ \frac{\mathrm{ticsForPrevRebuild} + \mathrm{totalTicsForSubsequentVerletUpdates}}{\mathrm{numUpdatesSinceRebuild}+1}$$

A history of these times was kept, and the verlet radius was changed up or down depending if *avgTicsPerItSinceReb* for the set of iterations just completed was an improvement on the *avgTicsPerItSinceReb* for the previous set. Each iteration, the verlet radius was changed up or down a default increment value of 0.01 times the value of r_c. If the performance settled at a particular point (determined when r_v fluctuates up one then down one from some value) the increment value was reduced by half, so that, eventually, a more precise value for the optimal verlet radius would be found.

However, most of these fluid simulations are dynamic in nature, and particles often speed up or slow down. Often particles start with negligible velocity, and then gradually speed up to some state of equilibrium and in other simulations, the user may wish to dynamically heat or freeze the particles to see the effects. For this reason, it made sense to not allow r_v to settle towards or stop at one given point, but instead be allowed to change up and down throughout the simulation. There was little investigation into an ideal increment value, but ideally the simulation should not fluctuate far enough from the optimal r_v to have any significant influence on performance, nor should it change so slowly that it cannot respond to changes of state within the system. An increment value of 0.01 performed quite well. Figure 5.30 shows the performance obtained by using this algorithm over 1400 timesteps, and shows the order in which *rvDivRc* was changed. Notice that the performance obtained using the algorithm was only slightly worse than choosing the optimal *rvDivRc* value outright.

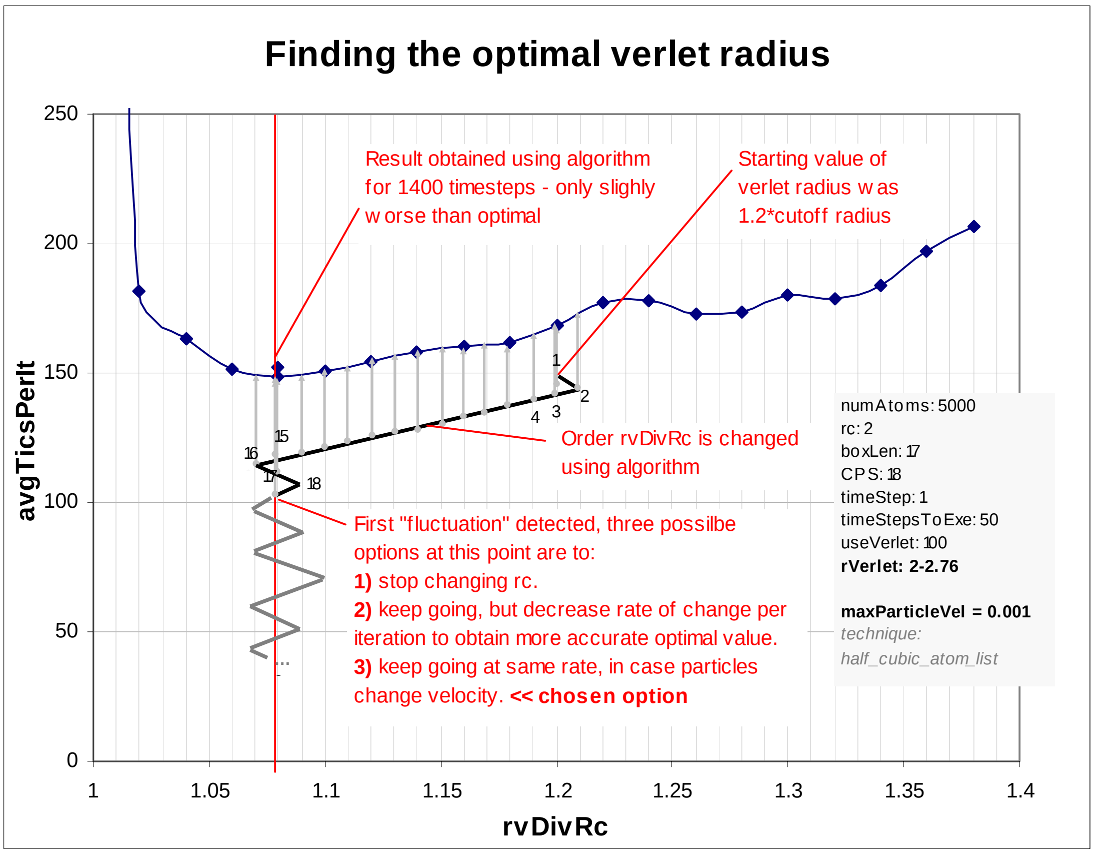


Figure 5.30. Results of simple optimal verlet radius algorithm.

Ideally, this algorithm for finding the optimal verlet radius and the algorithm for finding the optimal number of cells per side are complementarity. The use of both algorithms is not tested thoroughly enough to present results, but note that the algorithms may affect the effectiveness of each other. The optimal verlet radius finding algorithm changes *rvDivRc* just prior to each rebuild, while the optimal cells per side find algorithm measures the performance of the rebuild, and changes the number of cells per side after the rebuild. A potential problem with this is that the search radius will be changed slightly each rebuild, therefore affecting the size of the neighbour list and changing the optimal value for the number of cells per side. This conflict could easily cause one to select an incorrect value. Ideally, the two algorithms would have awareness of each other and interact intelligently. One simple idea is to let the algorithms take turns such that the verlet radius settles at a fixed value, then the number of cells per side is adjusted, then the verlet radius adjusts to reflect this, and so on. The author believes that anything more sophisticated than this may be unwarranted and only slight performance improvements may result.

5.15 Selective Checking of Verlet Neighbours

This section reports on the techniques to reduce the cost of checking neighbours proposed in Section 3.6. As shown in Figure 5.28 the time to check displacement is small, and ranges between about 0.3% and 0.8% of the total cost for most simulations tested during this thesis. For this reason, the idea of checking displacement at safe intervals is not worthwhile. However, the time to update the verlet list is large; taking 40% of the total cost for the optimal simulation in Figure 5.28. Figure 5.31 graphs the *fractNeiOutsideRc* (Definition 14) for different values of r_v. These values are considerable, for instance if r_v is 1.1 times r_c then 25% of neighbours are outside r_c, and if r_v is 1.2 times r_c then 42% of neighbours are outside r_c.

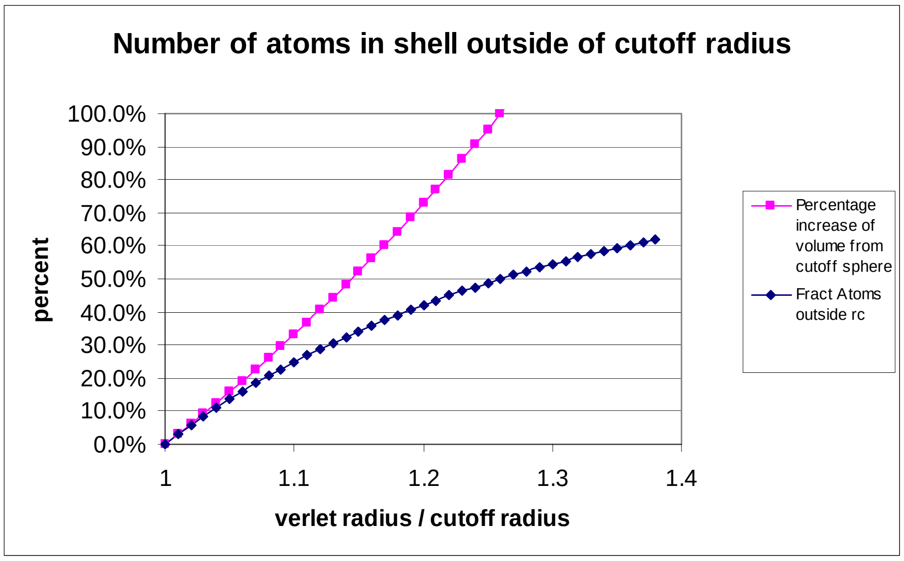


Figure 5.31. Fraction of atoms not in range for different verlet radius.

Figure 5.32 shows performance results for the two techniques for selective checking (Section 3.6) using the Lennard-Jones interaction model. The average cost of updating the list itself is also shown. The performance improvement of the selective checking using maximum velocity technique is minimal; but, selective checking using the displacement of individual atoms proved quite successful. Most interestingly, this technique reduces the gradient of the cost of updating the neighbour list as the verlet radius is increased. For this simulation, the performance improvement offered by selective checking using displacement of atoms each timestep improved performance by about 7%.

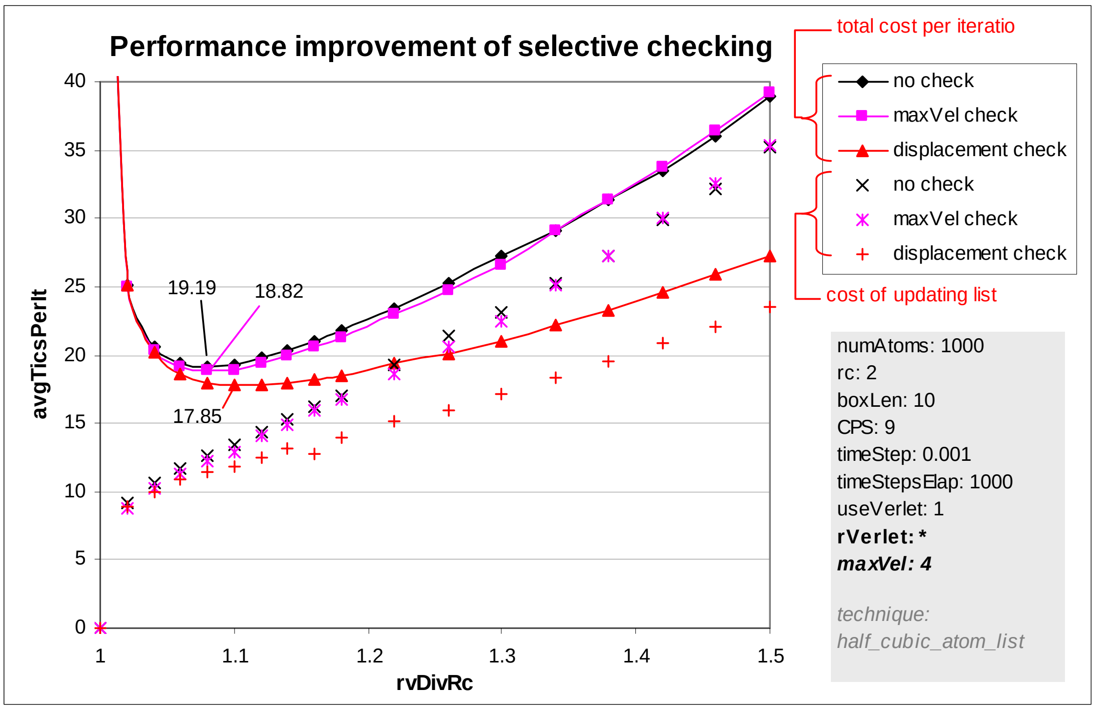


Figure 5.32. Performance improvement using selective checking of verlet neighbours.

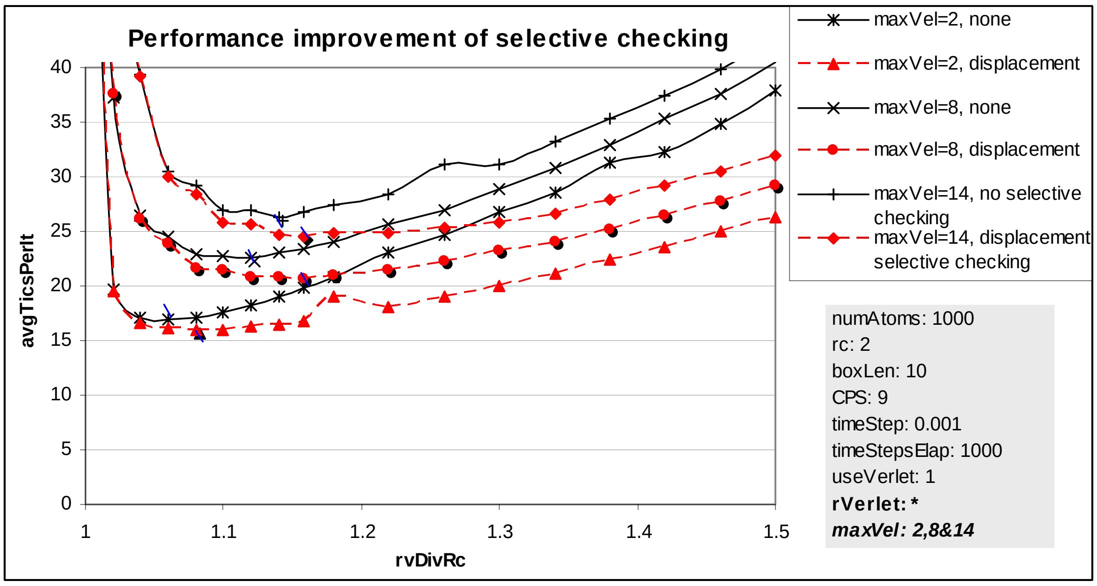


Figure 5.33. Improvements of selective checking at different atom velocities.

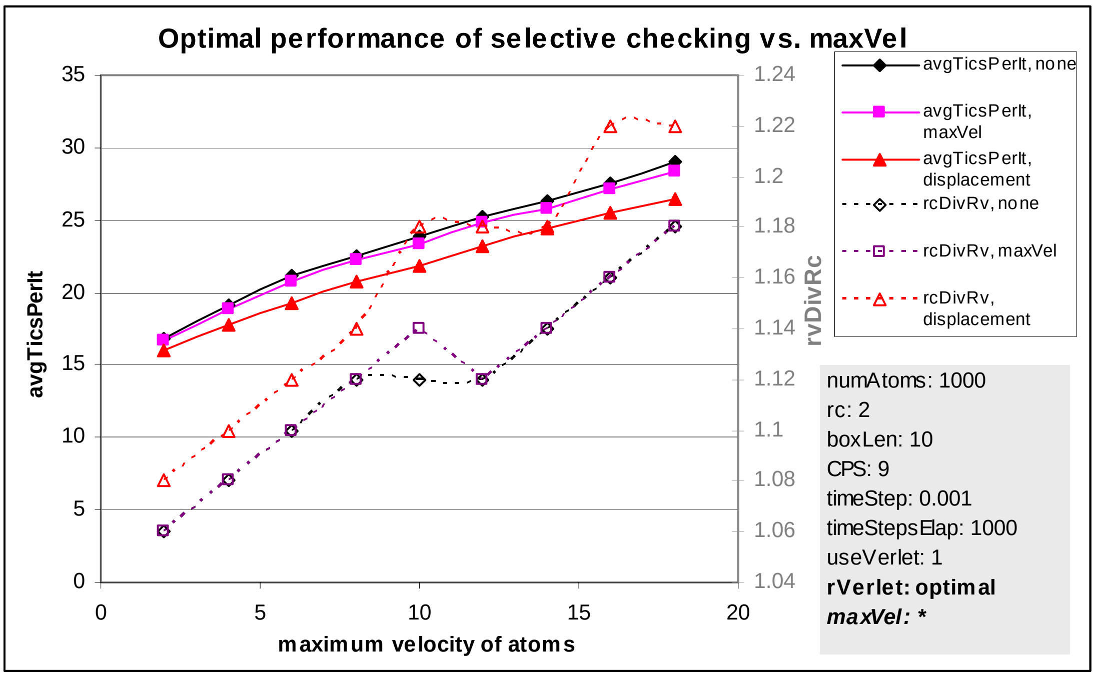


Figure 5.34. Optimal performance of selective checking of verlet neighbours.

Figure 5.33 shows similar results for three more sets of simulations, each with different particle velocities. Notice that the optimal performance of the selective checking of neighbours using displacement technique usually has a higher optimal *rvDivRc* value than the normal technique whereby the distance of all neighbours is computed every timestep. This relationship is also shown in Figure 5.34, which only graphs the results for the optimal performance of each technique for a range of different particle velocities. The average performance improvement over all these simulations was 1.8% for the maximum velocity checking technique, and 8.3% for the displacement checking technique. Moreover, another advantage is that, if non-optimal verlet radius is chosen, the loss in performance is not as great if the latter technique is used (Figure 5.33), since the curve increases slower. This author would recommend exploring this technique in future implementations of a molecular dynamics simulation, especially since it is not complicated to code (Figure 3.7). Note though, that improvements are liable to also depend on the size of the atom list compared to the size of the neighbour list.

5.16 Choosing Cutoff Radius

Choosing an appropriate cutoff radius is an important decision to consider in molecular dynamics simulations. The smaller the cutoff radius, the faster the simulation rate, but the worse the accuracy of results. Ideally, cutoff radius is large enough that only atoms far enough apart to have only negligible influence on each other are out of range (Figure 2.1). It is claimed that 2.5σ and 3.2σ are two most commonly used cutoff radius values, where σ is a characteristic distance in pair potential which fits the pair potential curve for a specific material [8].

Figure 5.35 shows the accuracy and performance relationship as cutoff radius is increased. A very small timestep was used, so timestep would have negligible affect on accuracy. The chosen measure of inaccuracy used in the following experiments was the absolute deviation of the position and velocity of all particles, compared to an equivalent control simulation using a cutoff radius of 5 units. The experiment and control were run over the same time frame and deviations averaged over all particles.

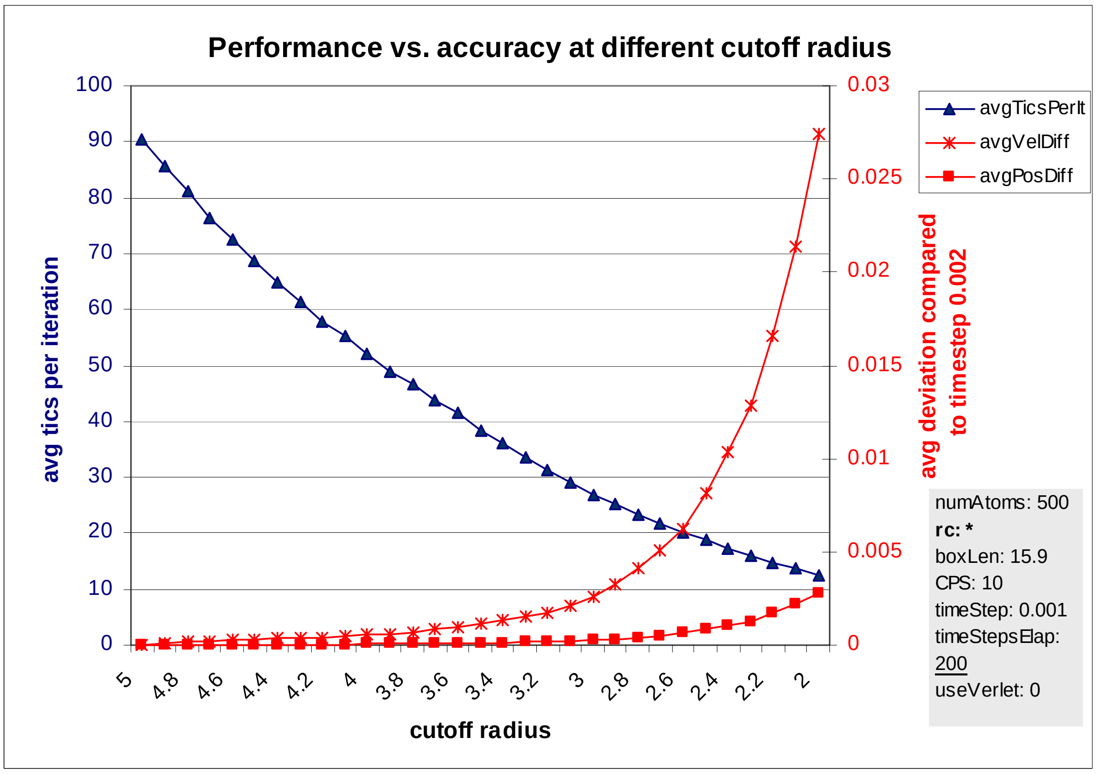


Figure 5.35. Performance and accuracy vs. cutoff radius.

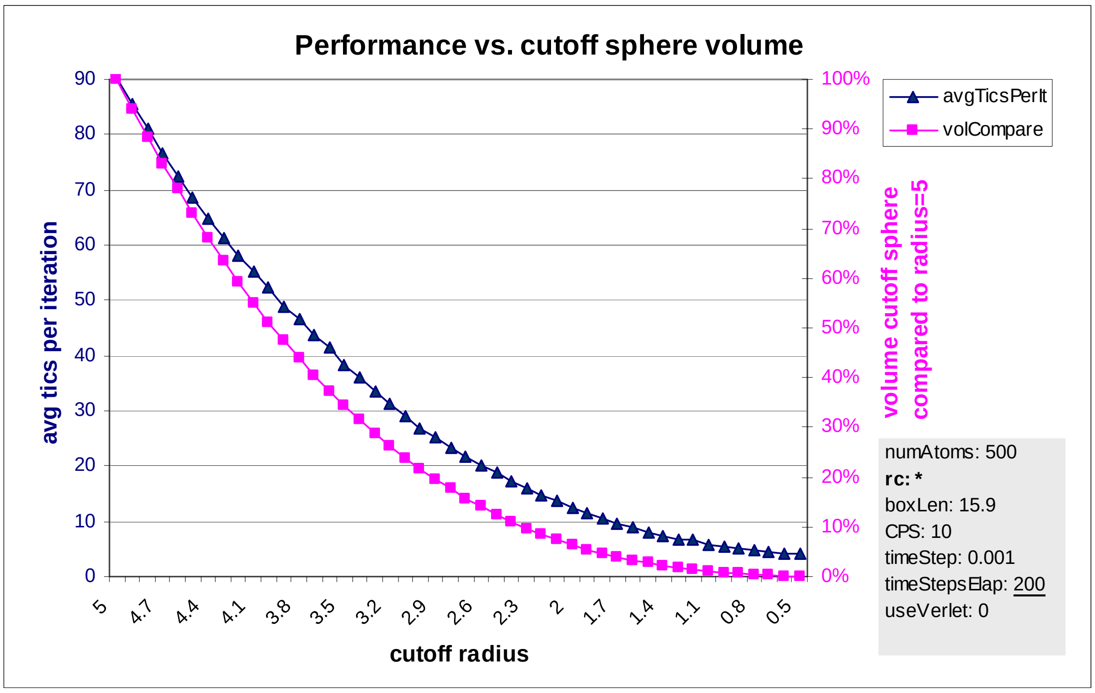


Figure 5.36. Performance-volume relationship

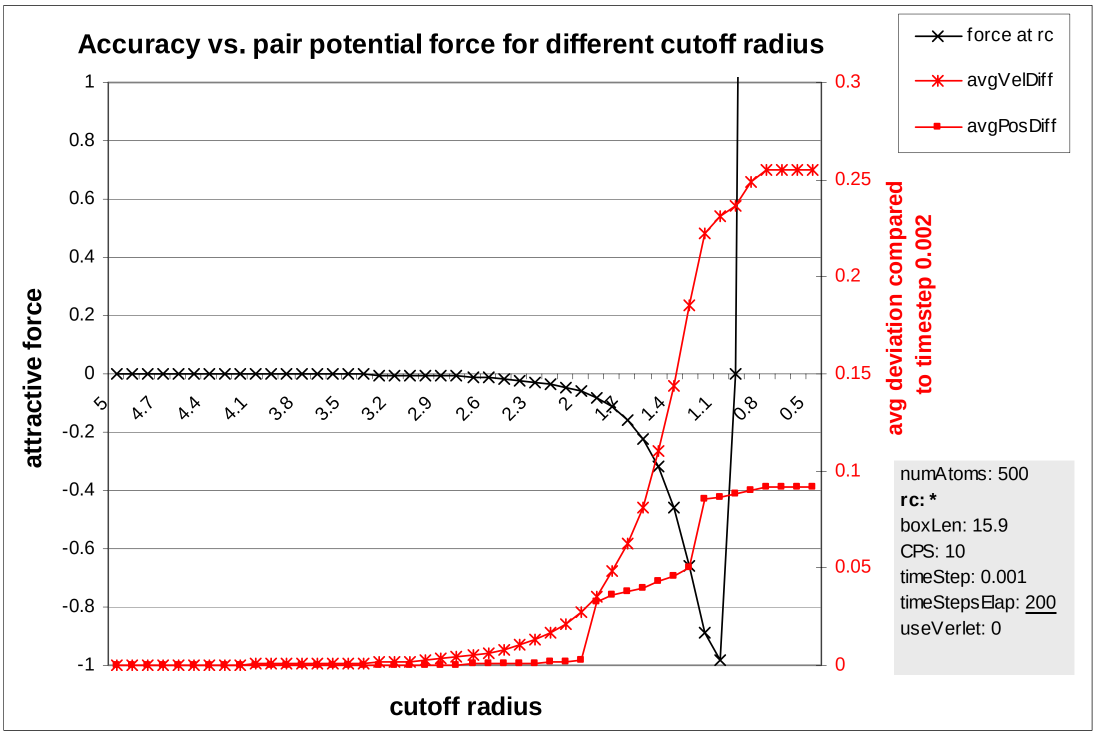


Figure 5.37. Accuracy vs. pair potential force.

Notice performance improves by a rate of approximately ($4/3\ \pi\ r_c^3$), as expected (Figure 5.36). The rate of error increases exponentially. In each iteration, atoms may only deviate a small distance from where they are located in the control experiment, but this affects all future force calculations, thus the error compounds quickly. Figure 5.37 shows the rate of error graphed against plotted Lennard-Jones pair potential forces. At a radius of about 2.3 the deviation in velocities is no longer negligible, and starts increasing faster, roughly in proportion to the changes in attractive force. At the point at which forces start to repel, difference in velocity stops because if the simulation was run any longer, the particles would get so close to each other they would repel with such a force that the program terminates due to arithmetic overflow. It is recommended that a good molecular dynamics simulation program should check any cutoff radius specified by the user, calculate the force at that distance of separation, and warn the user if that cutoff radius is likely to result in very poor results or even a program crash.

5.17 Choosing Timestep

Ideally, timestep is as small as possible. Molecular dynamics is always an approximate science where the longer the timestep, the less accurate the results. In the worse case scenario, the timestep will allow atoms to move too far between single iterations, allowing atoms to get closer together than they ever could in a real liquid. This usually causes an incorrect “chain reaction”, whereby two close particles repel at a much faster speed than normal causing them to bump even closer other atoms, which are repelled at an even greater velocity. This effect compounds until all atom are moving at unrealistic speeds and eventually arithmetic overflows will occur.

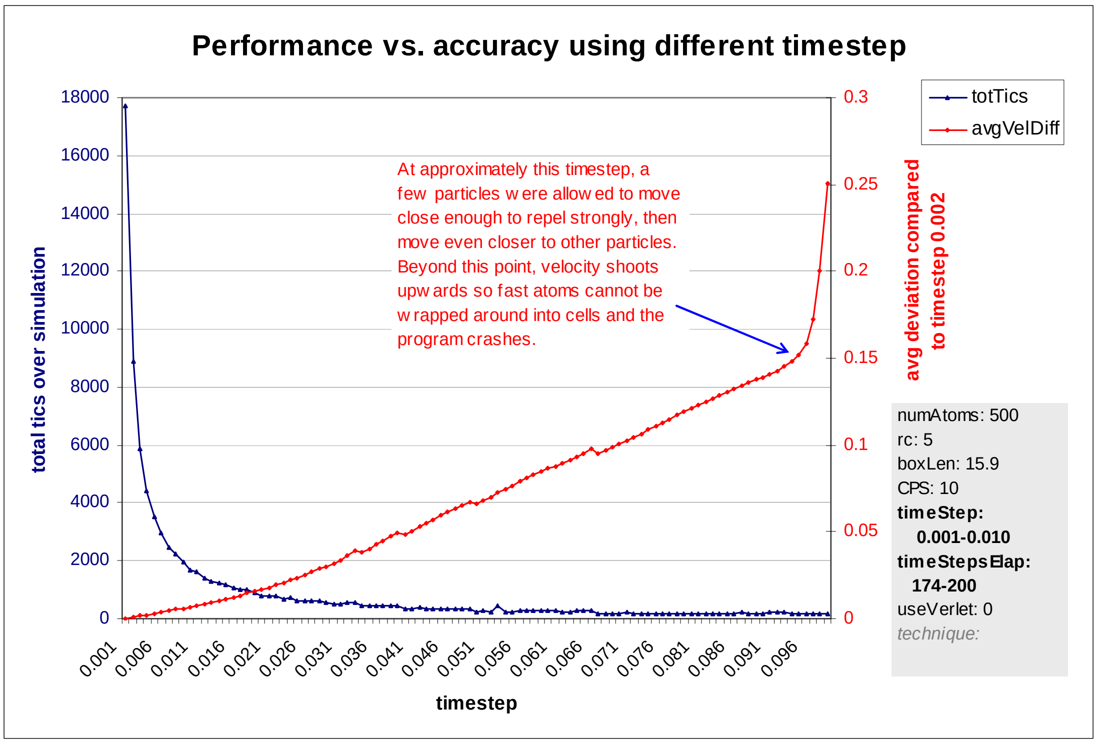


Figure 5.38. Performance and accuracy vs. timestep.

In Figure 5.38, several different timestep simulations were run and compared against an equivalent control simulation using a small timestep of 0.001. Each simulation is run through a fixed timeframe (0.480 seconds). A larger timestep requires fewer iterations and therefore the overall performance as total number of tics (not tics per iteration) should improve roughly proportional with the increase in timestep; which it does. The results also show increasing timestep decreases accuracy roughly linearly, however, at a certain point when the particles get too close together, an incorrect chain reaction occurs, and deviations immediately shoot towards infinity (which is difficult to depict in the graphs). In practice, it pays to be wary of this, and warn or prevent the user from entering a timestep which is too large. Ultimately, finding an appropriate balance between performance and accuracy depends on the requirements of the simulation.

Figure 5.39 shows how performance degrades over time. A simulation using a timestep of 0.01 was compared to a control simulation using 0.0001, and deviations compared at constant timeframe intervals. The average velocity of all particles in the tested simulation is also shown. In the given simulation (Figure 5.39), atoms start off with random velocities, speed up slightly (as shown by the blue line), and eventually the fluid reaches a fairly constant state whereby atoms are vibrating about, but the total entropy of the system remains about constant. Results show that the deviation in total velocities of particles starts off slowly increasing, and then increases faster, but eventually this difference in velocity approaches some constant. In other words, in the simulation with the larger timestep, particles speed up slightly faster and will, on average, remain travelling slightly faster for the whole experiment. Meanwhile, the deviation of particles' positions is also slow to start and shows correlations with the velocity trend. Eventually, just after the deviation in velocity settles down, the trend of total position deviation appears to become linear, but then the rate of increase reduces slightly. It is conjectured that this is due to the fact particles are not travelling as straight at this time, and many particles even wobble back over their own paths.

In summary, the choice of timestep has a big impact on accuracy of simulations. It is recommended that, to avoid the incorrect "chain reaction" phenomena, if two atoms get unrealistically close, the user should be warned that the timestep should be decreased and be given the option to terminate the program since the results are already effectively useless. A more advanced program might provide warnings if the user enters an unrealistically large timestep before the simulation is allowed to start.

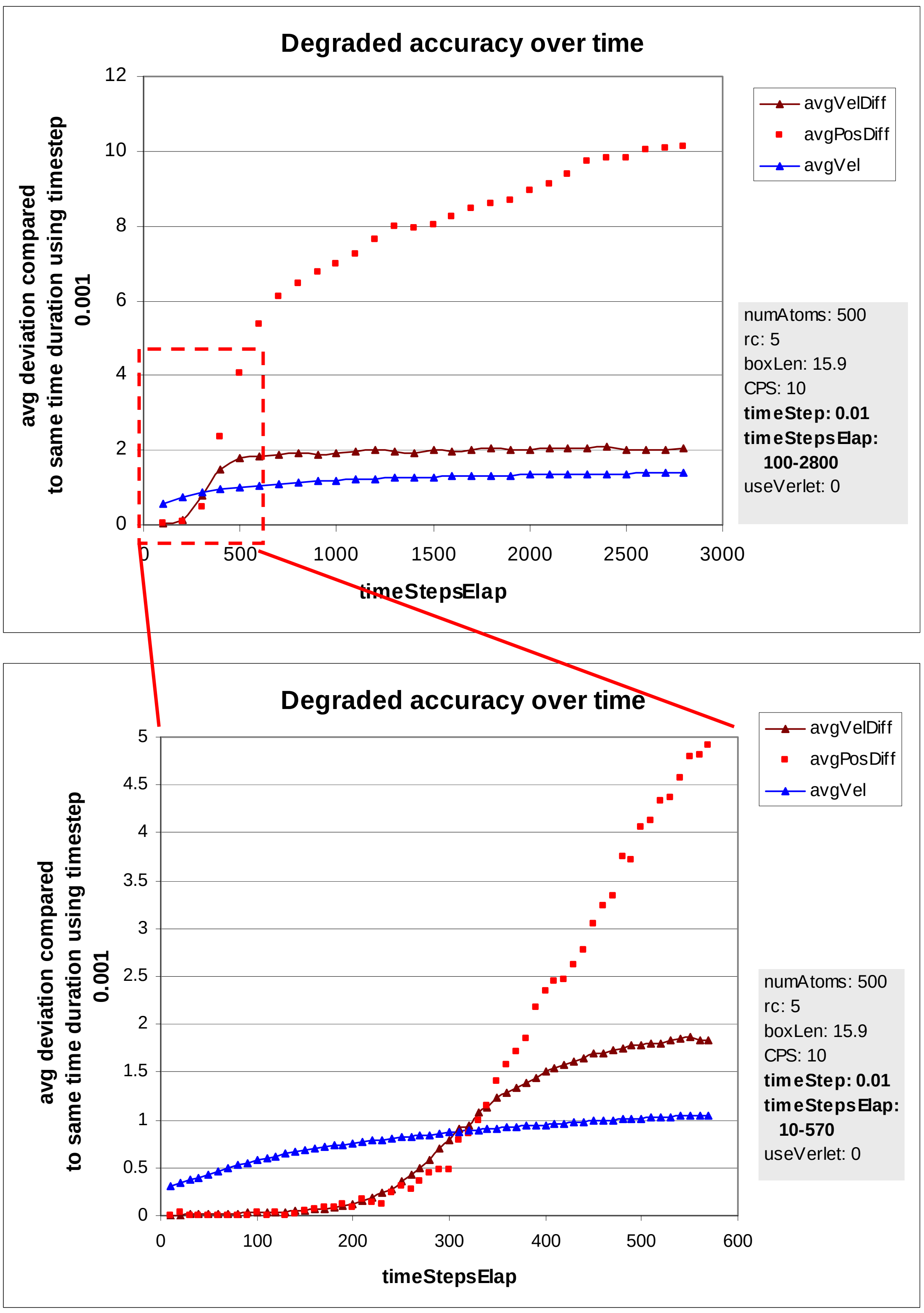


Figure 5.39. Effect of degrading accuracy over time.

6 Conclusion

This chapter presents a summary of results and suggests direction for further work. Many of the results provide examples of traditional trade-offs between computation time and storage space and also the need for temporal and spatial locality to increase cache effectiveness.

6.1 Summary of Results

In this thesis, several existing and proposed methods for optimising a common molecular dynamics problem were evaluated. It was established that building a neighbours list occupied the bulk of processor time. Improving the performance of this step was the focus of most of this work. Arguably the most significant discovery of this thesis was that half range searches yielded significant performance improvements over the traditional full range search technique, especially where the search radius spanned a large number of cell sides. The value of the cutoff radius divided by the length of the cell side, denoted *cellSidesPerRs*, proved to be a critical value throughout the thesis.

Results demonstrated that loading a minimum cell list was much more effective than re-calculating adjacent cells each iteration, although the former approach can consume a large amount of storage and take a significant amount of time to load at the start of the simulation. It was found the loaded cell list technique often performed slightly better than the atom list technique. However the atom list technique requires no loading phase and the performance when varying the number of cells per side exhibited a much smoother trend.

Several additional optimisation methods were attempted with limited success. A technique for early elimination of neighbours was investigated, as was a technique for using integer arithmetic instead of floating point arithmetic in construction of neighbour lists, but both methods resulted in worse performance. Of the three types of space-filling curves investigated, the most effective was the Hilbert curve, although all curves yielded much smaller improvements (compared to a completely random ordering of atoms) than expected, unless the number of atoms was in the hundreds of thousands. It was also found that using a single atom object to contain all atom data exhibited significantly better spatial locality and performance than creating a separate array for the position of atoms.

A technique of using sub-grids to refine the searching of a loaded cell list proved effective, but only if the number of atoms per cell was high, as did a minimum atom list technique and the use of minimum bounding rectangles in cells. However, a significant discovery was that atom list technique typically performed best when the number of cells per side was adjusted so that there were only one or two atoms per cell. For a cell list, the optimal number of cells per side was very much dependent on *cellSidesPerRs*, since this dictated the number of cells in the minimum cell list, rather than the average number of atoms per cell.

The author speculates that, a cubic atom list is preferable to a loaded minimum cell list, because it is easy to program and allows the dynamic changing of the number of cells per side, or even the cutoff radius itself, with minimal affect on performance. The thesis also analysed the optimal number of cells per side and proposed an algorithm able to dynamically find the optimal number of cells per side for an atom list. This algorithm was found to yield close to optimal performance. An almost identical algorithm was proposed to find the optimal verlet radius to use in the verlet neighbour list technique. As expected, the verlet list technique, yielded significant performance improvement compared to rebuilding each iteration, depending on the velocity of particles. Further performance improvements were obtained by a new method of selective checking of the neighbours list using the displacement of atoms, which significantly reduces the cost of updating the verlet list.

Several of the methods presented could be used in a wide range of applications which execute spatial searches, not just molecular dynamics simulations. For those implementing molecular dynamic simulations, this author strongly recommends using the half range search technique (rather than a full range search) and using an atom list technique (rather than the traditional cell list). Using the verlet neighbours technique and the selective checking of the neighbour list using atom displacements technique is also advisable. Space-filling curves may also yield large improvement, but these improvements are likely small and not worth the effort of the extra code and complexity unless simulating vast numbers of particles. Moreover, finding the optimal number of cells per side and verlet radius is critical to optimising these simulations, so using an algorithm to dynamically find these values (or at least provide a good estimate), similar to the algorithms described, is essential. By implementing all these complimentary techniques, the performance of any spatial join simulations can be greatly improved, and better than halved. Optimisation of this magnitude is surely welcome in large molecular dynamics simulations which often take many days to process.

6.2 Future Work

The particular molecular dynamics fluid problem investigated is well understood, but there remains an opportunity for much future work on detailed analysing of and optimising the performance of such simulations. This is important, since similar problems of range searches are used in many scientific applications and computer experiments, and scientists are constantly trying to increase the number of particles and amount of data in such experiments. Due to time constraints, no results for simulations for more than 200 thousand atoms were presented, but scientists are naturally interested in making larger simulations with many millions of particles.

One of the main foci of future research will be a better analysis of finding the optimal parameters for a given simulation, including more advanced means of finding the optimal number of cells per side and verlet radius for certain simulations. Some suggestion to extend this work are listed below:

- Investigate various algorithms to find the minimum point on the performance curve (representing the optimal number of cells per side or verlet radius). As observed, the simple algorithms proposed and implemented in this thesis have a high likelihood of resolving to a local minimum.
- Investigate ways to find the optimal verlet radius and number of cells per side simultaneously. The two current algorithms are not designed to co-operate in any way, and the effectiveness of running both at once has not been properly tested.
- Investigate the use of Hidden Markov models to determine best number of cells per side, given the main input parameters.
- Experiment with the performance of various techniques in this thesis using different compilers and platforms. Currently performance was tested using a single programming language and a single machine, so it should be especially interesting to test performance differences between different languages.
- Thoroughly test the performance of space-filling curves for larger numbers of atoms travelling at different velocities, with and without using a verlet radius. This author believes the curves should yield significant improvements for two-dimensional simulations, but had insufficient time to test this theory. Furthermore, if the dataset were too large to fit in memory and instead resided on disk, space-filling curves would be expected yield huge performance improvements.
- Testing of the accuracy of various statistical functions for different timestep and cutoff radius values. This thesis compared accuracy of simulations by comparing the position

and velocity of atoms; but it would be interesting to see performance relationships by comparing temperature, energy, and numerous other measurements, since these are the important outputs from molecular dynamics simulations.

- Investigate the usefulness of techniques proposed in this thesis for skewed data sets, particularly use of sub-grids minimum bounding rectangles.
- Investigate the performance of techniques proposed in this thesis using other data structures such as R-trees, quad-trees and various types of grid files.
- Write a more comprehensive guide to the performance benefits of the half range search.
- Adapt cell lists so that some cells are always searched exhaustively, but certain outer cells (or their MBRs) in the adjacent cell list are first checked to determine if they are within range of each atom. This technique is more sophisticated than checking every cell, and could be implemented by splitting the cell list into two separate lists. However this approach is still only likely to yield improvements in simulations with high number of atoms per cell, since it is cheaper to compare atoms to each other than calculate the minimum distance from an atom to a cell.

This work is most likely to be continued by future students in TOMSK [19], if not by this author.

7 Appendices

Appendix A: Common C++ Data Types

Table 4 shows some of the most commonly used data types in C++ [21]. The int data type is system dependent, but on most systems, including the tested machine, is equivalent to __int16.

Data-type name(s)	Bytes	Decimal digits	Exponent range	Range of Values
int	*	*	*	*System dependent, but usually equivalent to __int16*
__int16	2	5	N/A	*–32,768 to 32,767*
long (__int32)	4	10	N/A	*–2,147,483,648 to 2,147,483,647*
__int64	8	19	N/A	*–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807*
float	4	7	38 to 38	*3.4E +/- 38*
double *	8	15	-308 to 308	*1.7E +/- 308*
long double	10	19	-4932 to 4932	*1.2E +/- 4932*
bool	1	N/A	N/A	*false or true*

Table 4: Relevant C++ data types ranges.

Appendix B: Performance of Various Operations on Chosen Platform

The following results were used to aid implementation decisions during the coding of the simulation. Results were obtained by writing a simple testing platform which timed and executed a single operation numerous times (500 million or more) in a loop, and compared this with a baseline of running the same loop with no operation, in order to obtain a rough average number of clock tics for the given operation or method.

$$avg\ tics\ per\ operation = \frac{tics\ elaped\ during\ occupied\ loop - tics\ elapsed\ during\ empty\ loop}{number\ of\ iterations\ in\ loop}$$

Results were outputted to a CSV file and analysed in Excel, since this method was found easier than using a profiler. Tests were run on the same test computer; 2.6 GHz Pentium 4 machine with 512 MB of RAM and 512kB of L2 cache; using the default debug configuration mode so loops would run exactly as coded.

Figure 7.1 shows the speed of basic arithmetic operations for the common C++ data types. Results show multiplication, addition and multination is very cheap for long, int and float, but more expensive for __int64 and more expensive again for double. Division is expensive for all data types, but actually more so for the integer-based data types, presumably because they must be converted to floating-point numbers before division occurs. Figure 7.2 shows some of the many other operations which were tested. If statements were inexpensive, so too is the right shift operation, which is approximately four times less expensive than division using doubles, depending on it's configuration. Right shift only works for integer-based data types. Certain typecasting operations, such as converting long to a __int64 were more expensive than expected.

Finally, Figure 7.3 shows some common functions from the C++ maths library. The sqrt() operation was approximately 64 times more expensive than double multiplication. Results also show that using the pow() function is inadvisable if it's possible to multiply the numbers by themselves instead (if trying to obtain the square or cube of a number for instance). Interestingly, the fmod() operation, which performs the modulus operation for two floating point inputs, was more expensive to call than an equivalent function fMod() written as a simple inline function with if statements. Such speedup can easily add up, for example the modulus operation was called frequently to put any atoms which had wandered outside the box boundaries back into the box.

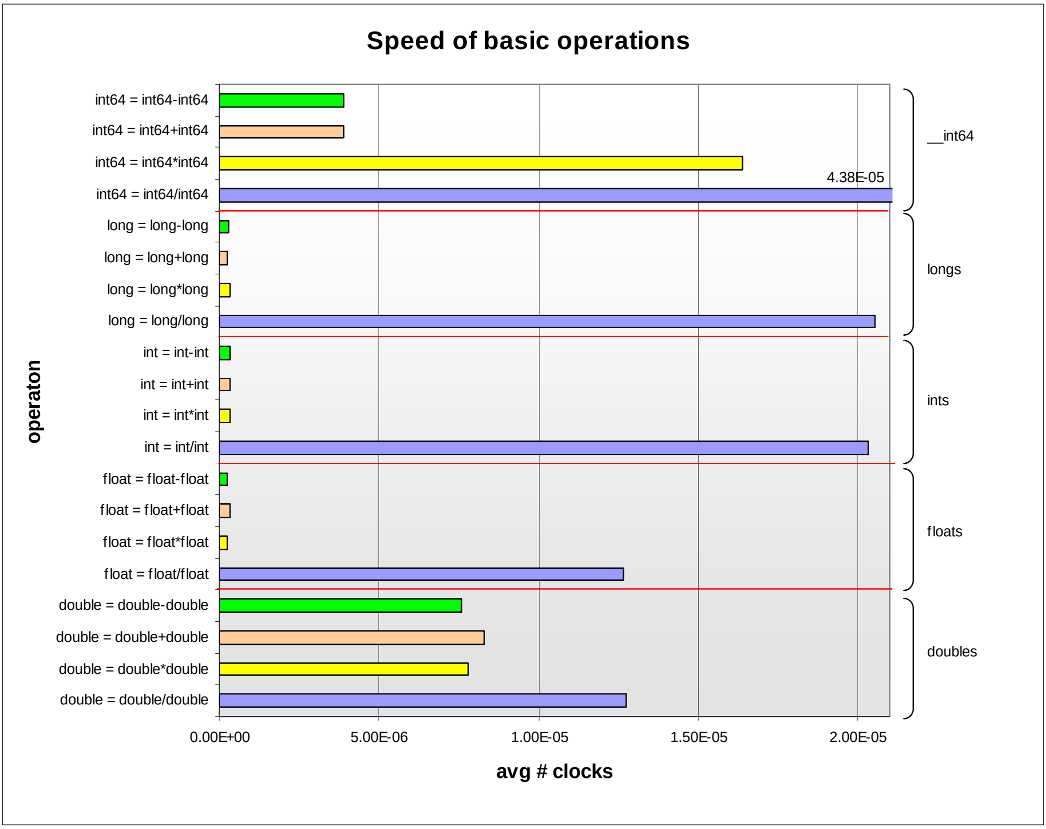


Figure 7.1. Speed of basic arithmetic operations.

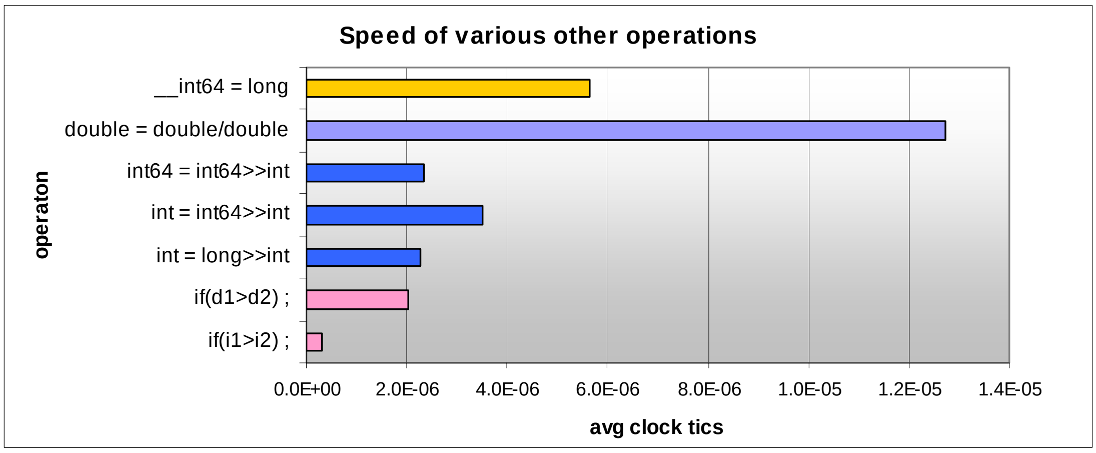


Figure 7.2. Speed of other relevant operations.

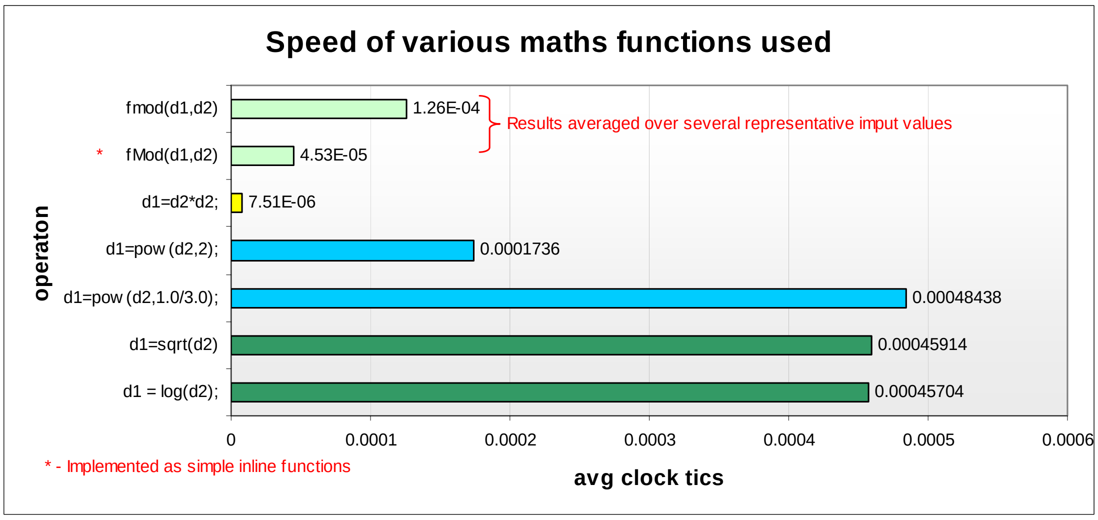


Figure 7.3. Speed of relevant functions.

Appendix C: Choice of Data-type: Double or Long

In the initial code, the doubles were used to store all atom positions, whereby each coordinate would be between 0 and *boxLen*. An alternate idea was to store atom positions as longs, whereby each coordinate would be between 0 and some large long value *l_boxLen*, because integer-based arithmetic is characteristically faster than floating point arithmetic (Figure 7.1). Division is an expensive operation for all data-types; however the bitwise right shift operation (written as ">>"), which only works for integer-based data-types, can be used to divide a number by any exponent of two and is much cheaper (Figure 7.2).

$v>>i \equiv v/2^i$, where v and i are both integers

Indexing an atom to a cell is a frequent operation, and requires the atoms position to be divided by the cell length for each axis. If a long is used, the same results can be achieved by right shifting the coordinate so that only the bits representing the index position for the cell remain. Initially it was thought this technique would limit the values of cell length and cells per side which could be used in right shift, however it was discovered that, by carefully configuring *l_boxLen* and using several other parameters, this approach can work for any number of cells per side. Table 5 shows how parameters were set up.

Variable Name	**Value Assigned**	**Data Type**	**Purpose/Description**
BIT_MAX_LONG	32	*integer*	
maxLong	$2^{\text{BIT\_MAX\_LONG}}$	*integer*	Represents the maximum possible value long can be set to (Table 4).
bitsCPS	$\lceil \log_2 \text{CPS} \rceil$	*integer*	Represents the minimum number of bits needed to represent the CPS.
rShiftAtomToCell	bitsMaxLong – bitsCPS	*integer*	Is the number of bits which will be used to represent values between 0 and cellLen. By right shifting an atom's position by this amount, only the bits representing the cell's index along that dimension will remain.
l_cellLen	$2^{\text{rShiftAtomToCell}}$	*long*	Represents the cell length as a long.
l_boxLen	$2^{\text{rShiftAtomToCell}} \times \text{CPS}$	*long*	Becomes the total box length as a long.
longToDouble	boxLen/l_boxLen	*double*	Used to covert any doubles to longs when needed. ie: $atomPosDouble = atomPosLong \times longToDouble$
doubleToLong	l_boxLen/boxLen	*double*	Opposite of above.

Table 5. Parameters used to allow right shift operation.

To establish the cell an atom belongs to can now be done by calling:

$cellIndex_{axis} = atomPosLong_{axis} >> rShiftAtomToCell$

This technique was ideal for placing atoms into cells; however, calculating distances was now a problem. In order to calculate distance squared, the distance along each axis must be squared and all these added together. This is straightforward for floating-point numbers which use exponents, but for integer-based numbers, squaring an integer means the number of bits required to represent that number is effectively doubled, meaning a long had to be typecast to an __int64 (Table 4) in order to safely square its value. Accuracy is an extremely important factor in molecular dynamics, and notice in Table 4 that a long only has 10 decimal digits of accuracy whereas a double has 15 significant figures.

Using the proposed methods of longs was tested, but although the building of atom lists became faster, the cost of comparing distances did not improve, and overall performance was about 5% worse in the results collected. A possible method would be to use both doubles and integers to store atoms' positions, to gain the advantage of both data-types, but this would result in further code complexity and storage requirements. The user of a molecular dynamics simulation will naturally want any results to be outputted as floating-point numbers, not longs, meaning any performance improvements provided by integer-based data types would probably be overshadowed by the need to frequently convert one data-type to the other (possibly loosing accuracy in the process). The recommendation of this thesis is to keep things simple by sticking with doubles or floats if implementing molecular dynamics simulations. Possibly for other applications the use of longs as described above would yield more substantial advantages.

Appendix D: Efficient Generation of Random Directions in Three-Dimensional Space

Generating a random direction for two dimensions is a simple matter of generating a single random number between 0 and 180 degrees. However, generating a non-biased random direction for three dimensions using angles is more complicated, so the author proposed and used a simple approach to solve the problem. A random point in cube was generated by generating a random coordinate between -1 and 1 along each axis. This point was then checked to see if it was within a distance of 1 from the origin. If so, the vector from the origin to the point provided a random direction, if not, the process of generating points would repeat until successful. Since a sphere occupies 52% of its bounding cube, about one in every two attempts will be successful. This technique was used in the generation of random velocities and random position offsets at the start of each simulation.

Bibliography

1. D. J. Abel and M. D. M., *A comparative analysis of some two-dimensional orderings*, Int. J. Geograph. Inf. Syst 4 (1), p.21-31 (1990).
2. M. P. Allen and D. J. Tildesley, *Computer simulation of liquids*, Oxford University Press, New York, 1987.
3. J. E. Barnes and P. Hut, *A hierarchical o(nlogn) force calculation algorithm*, Nature 324 (4), p.446-449 (1986).
4. J. L. Bentley and J. H. Friedman, *Data structures for range searching*, ACM Computing Surveys (CSUR) archive 11 (4), p.397-409 (1979).
5. G. Blelloch and G. Narlikar, *A practical comparison of n-body algorithms*, In Parallel Algorithms, Series in Discrete Mathematics and Theoretical Computer Science. (1997).
6. E. Chavez, G. Navarro, R. Baeza-Yates and J. Marroquin, *Searching in metric spaces*, Technical Report TR/DCC-99-3, Dept. of Computer Science, Univ. of Chile (1999) To appear in ACM Computing Surveys (1999).
7. M. T. Dickerson and D. Eppstein, *Algorithms for proximity problems in higher dimensions*, Computational Geometry: Theory and Applications 5, p.277-291 (1996).
8. F. Ercolessi, *A molecular dynamics primer*, http://www.fisica.uniud.it/~ercolessi/md/md/, Date accessed 01-07-2004.
9. C. F. Fischer, *The hartree-fock method for atoms*, John Wiley & Sons Inc, New York, 1977.
10. J. H. Freidman, J. L. Bentley and R. A. Finkel, *An algorithm for finding best matches in logarithmic expected time*, ACM Transactions on Mathematical Software (TOMS) 3 (3), p.209-226 (1977).
11. D. Frenkel and B. Smit, *Understanding molecular simulation: From algorithms to applications*, Academic Press, 1996.
12. V. Gaede and O. Günther, *Multidimensional access methods*, Source ACM Computing Surveys (CSUR) archive (June 1998) 30 (2), p.170-231 (1998).
13. L. Greengard and V. Rokhlin, *A fast algorithm for particle simulations*, Journal of Computational Physics (December 1987) 73 (2), p.325-348 (1987).
14. A. Guttman, *R-trees: A dynamic index structure for spatial searching*, In proceedings of the ACM SIGMOD International Conference on Management of Data, p.47-57 (1984).
15. R. W. Hockney and J. W. Eastwood, *Computer simulation using particles*, Publisher Taylor & Francis, Inc., Bristol, PA, USA, 1988.
16. V. Jain and B. Shneiderman, *Data structures for dynamic queries: An analytical and experimental evaluation*, Proceedings of the workshop on Advanced visual interfaces, p.1-11 (1994).
17. D. V. Kalashnikov, S. Prabhakar and S. E. Hambrusch, *Main memory evaluation of monitoring queries over moving objects*, Distributed and Parallel Databases archive (March 2004) 15 (2), p.117-135 (2004).
18. D. V. Kalashnikov, S. Prabhakar, S. E. Hambrusch and W. G. Aref, *Efficient evaluation of continuous range queries on moving objects*, Proceedings of the 13th International Conference on Database and Expert Systems Applications, p.731-740 (2002).
19. D. Konovalov, *Tomsk group*, http://www.it.jcu.edu.au/~dmitry/tomsk/, Date accessed 9/2004.
20. S. Meyers, *Effective c++*, Addison-Wesley, Indianapolis, 2003.
21. Microsoft, *C++ language reference - data type ranges*, http://msdn.microsoft.com/library/en-us/vclang/html/_langref_data_type_ranges.asp, Date accessed 8/11/2004.
22. B. Moon, H. v. Jagadish, C. Faloutsos and J. H. Saltz, *Analysis of the clustering properties of the hilbert space-filling curve*, IEEE Transactions on Knowledge and Data Engineering 13 (1), p.124-141 (2001).
23. J. Nievergelt, H. Hinterberger and K. C. Sevcik, *The grid file: An adaptable, symmetric multikey file structure*, ACM Transactions on Database Systems (TODS) 9 (1), p.38-71 (1984).
24. A. Noske, *Molecular dynamics simulation and other self-spatial join - an engine layer and performance testing platform*, http://manning.it.jcu.edu.au/~jc130551/thesis/, Date accessed 26/11/2004.
25. A. Papadopoulos, P. Rigaux and M. Scholl, *A performance evaluation of spatial join processing strategies*, Proceedings of the 6th International Symposium on Advances in Spatial Databases, p.286-307 (1999).

26. S. Prabhakar, Y. Xia, D. V. V. Kalashnikov, W. G. G. Aref and S. E. E. Hambrusch, *Query indexing and velocity constrained indexing: Scalable techniques for continuous queries on moving objects*, IEEE Transactions on Computers archive 51 (10), p.1124-1140 (2002).
27. N. Roussopoulos, S. Kelley and F. Vincent, *Nearest neighbor queries*, Proceedings of ACM Sigmod (May 1995) (1995).
28. H. Samet, *The quadtree and related hierarchical data structures*, ACM Computing Surveys (CSUR) archive (June 1984) 16 (2), p.187-260 (1984).
29. H. Samet, *The design and analysis of spatial data structures*, Addison-Wesley Longman Publishing Co., Inc, Boston, MA, USA, 1990.
30. B. Stroustrup, *The c++ programming language (special edition)*, Pearson Education, Indianapolis, 2000.
31. Y. Tao and D. Papadias, *Spatial queries in dynamic environments*, ACM Transactions on Database Systems (TODS) 28 (2), p.101-139 (2003).
32. L. Verlet, *Computer "experiments" on classical fluids. I. Thermodynamical properties of lennard-jones molecules*, Physical Review 159 (1968).
33. S. Wang, J. M. Hellerstein and I. Lipkind, *Near-neighbor query performance in search trees*, (1998).