Australian National University

RESEARCH SCHOOL OF COMPUTER SCIENCE

COLLEGE OF ENGINEERING AND
COMPUTER SCIENCE

The Fast Multipole Algorithm vs. the Particle Mesh Ewald method

Joshua Nelson - u4850020

COMP3006 - COMPUTER SCIENCE RESEARCH PROJECT

Supervisor: Dr Eric MCCREATH

October 26, 2012

Abstract

The N body problem is common across the fields of physics, biology and chemistry. The classic solution to this problem has an inhibitive complexity in the class $O(n^2)$. Two alternative methods were examined: The Fast Multipole Algorithm, and the Particle Mesh Ewald Method, with better complexities of $O(n)$ and $O(n\log(n))$, respectively. These algorithms were implemented in Java, and their efficiencies were discussed and compared. The algorithms were run over typical molecular dynamics simulations to determine the most efficient algorithm for the N-body problem.

Contents

Chapter 1

Introduction

This chapter introduces the N body problem, approaches that have been made in the past, and gives background on related literature.

1.1 The N body problem

1.1.1 Problem summary

Suppose we have a collection of n bodies in some space, that interact with each other. Each body interacts with every other body in the system in a pairwise way. Often this pairwise interaction is a function of the distance between the bodies, and their properties, such as mass or electric charge. The task is to calculate the total effect on each body from every other body.

The N body problem is key to the simulation of many different scientific environments. The bodies may be astrophysical objects, such as planets or galaxies, interacting based on distance and body mass [8], or atoms in a molecular dynamics simulation, based on distance and particle charge.[14]. During the report, we will mostly discuss the N body problem in regards to Molecular dynamics, however the approaches can be generalised to other fiields.

1.1.2 History of the problem

The N body problem was first formulated mathematically in Isaac Newton's Principia Mathematica. [10]. Later, H. Poincarè showed the impossibility of an analytical solution using first order integrals for $N \geq 3$. [12]. This led to the development of several approximation methods. These methods can be roughly classified as either tree methods, or mesh methods.

Tree methods are methods that break the simulation space into cells, and use a tree structure to organise these cells, allowing close interactions to be calculated directly, and far ones approximated. The Fast Multipole Algorithm, first described by L. Greengard and V. Rokhlin [6], is such a method, using multipole expansions to approximate far cells.

Mesh methods are ones that discretize the simulation space onto a mesh, and the potential is then solved over the mesh. The Particle Mesh Ewald method, developed by T. Darden et al. [2] is such a method, and is based on Ewald summation, developed by P. Ewald [5]. The variation of this known as the Smooth Particle Mesh Ewald method was developed by U. Essman et al. [4]

1.1.3 Related work

Comparisons of the Fast Multipole Algorithm and the Particle Mesh Ewald method that have been made previously have found that the Particle Mesh Ewald method outperforms the Fast Multipole Algorithm in

the range 10^4 - 10^5. (H. Petersen, [11]). A. Toukmaji et al. [1] claim that the mesh methods are faster in the range 10^3 - 10^4, and the multipole tree methods faster for $N > 10^5$.

1.1.4 Contribution

The contribution to this section of work that this paper provides includes a basic Java implementation of the Particle Mesh Ewald method and the Fast Multipole Algorithm. This software allows direct comparison of the two algorithms in terms of speed and accuracy. The code was written with readibility in mind, rather than focusing on excessive optimisation, which obscures the algorithm the code implements.

The report seeks to give insight into the approaches from a Computer Science perspective, while giving the background necessary to understand it from a Computational Chemistry perspective.

Chapter 2

Algorithms for the N body problem

This chapter describes the mathematics and the implementation of the main algorithms discussed in this report: The basic algorithm, the Fast Multipole Algorithm. and the Particle Mesh Ewald method. For each of these methods, we examine their running times and accuracy individually, and discuss the details of their implementations in Java.

2.1 The $O(n^2)$ solution

2.1.1 The algorithm

The simplest solution to the N body problem is the basic $O(n^2)$ approach of calculating each interaction directly (See Figure 2.1). Pseudocode for this soltuion is given in Algorithm 1.

Data: r_i: particle positions, q_i: particle charges, N: number of particles, Q: output array of charges
for *i=0 to N* **do**
  for *j=i to N* **do**
    if $i \neq j$ **then**
      $d := |r_i - r_j|$;
      $Q[i] := q_i * q_j/d$;
    end
  end
end

Algorithm 1: The basic approach to the N body problem

The advantages to the $O(n^2)$ approach are it's simplicity, ease of implementation, and it's low initialisation overhead. This low overhead can be seen in Figure 2.2. However, it's primary disadvantage is that it is limited to small numbers of particles by it's $O(n^2)$ complexity. These advantages and disadvantages are further discussed in Chapter 3

2.1.2 Implementation analysis

The $O(n^2)$ algorithm was run on increasing system sizes for one time step, and the time taken to calculate the potential at each atom's position was calculated. It is easy to turn this process into a dynamic simulation, using the relationship between potential energy, a particle's charge, and force.

We can see in Figure 2.2 that the running time increases porportionally to N^2. We can also see the low overhead when N is in the range $N < 1000$, where the total computation time ≈ 100ms.

Details of the computer these measurements were taken on can be found in Appendix A. By sampling Figure 2.2 at various points, we find that the rate of computation for the implementation is approximately 38.9 Mflops.

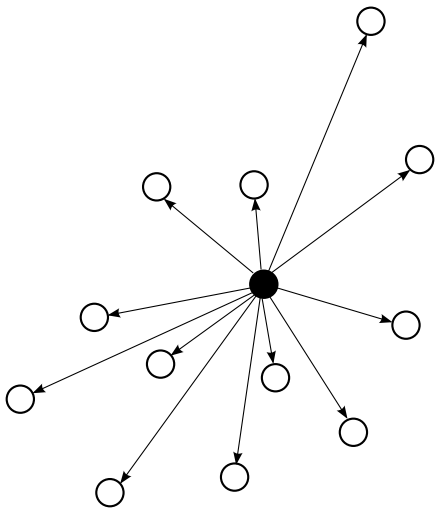


Figure 2.1: The naïve approach to the n body problem - calculate each interaction for each particle

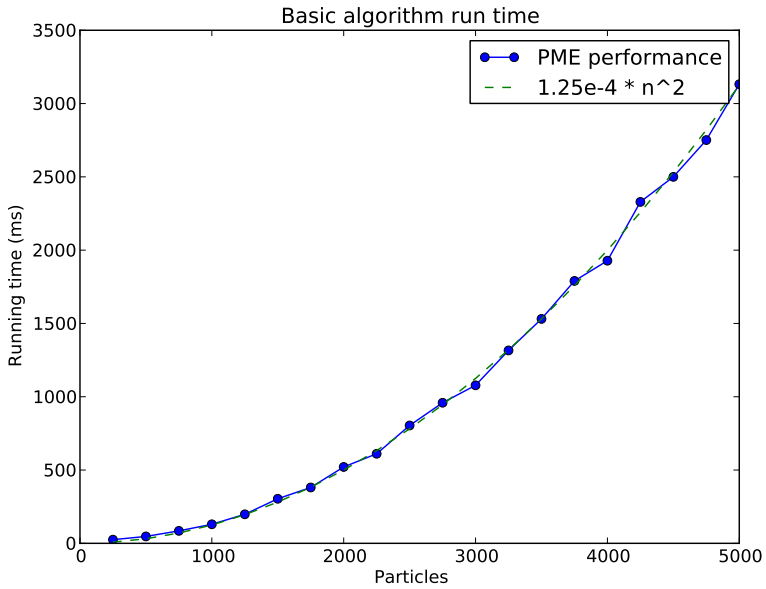


Figure 2.2: The basic algorithm's run time compared to a function of N^2. It is easy to see that the basic algorithms run time grows proportionally to N^2, as we would expect from it's complexity class $O(n^2)$

2.2 The particle mesh ewald method

2.2.1 Background

Potential vs. Energy

Energy and potential are related but different concepts. They both describe effects charged particles have on their surrounding environment, however, *Energy* is a property of a particle, and *Potential* is a property of a field. *Energy* is defined in terms of a particle-particle interaction, while *Potential* describes a field from one charged particle.
Coulomb's law describes the potential at r_2 from a charge q_1 at r_1 as

$$V = \frac{1}{4\pi\epsilon_0}\frac{q_1}{|r_2 - r_1|}$$

And energy from a particle with charge q_2 at r_2 as $E = q_2 V$.
For our purposes, the constant term $\frac{1}{4\pi\epsilon_0}$ is material dependent, and is discarded.

Ewald summation

The key concept behind the Particle Mesh Ewald method is that of *Ewald Summation*. Ewald Summation splits the potential ϕ at a position r into two components, the long range potential and the short range potential. [11].

$$\phi(r) = \phi_{\text{sr}}(r) + \phi_{\text{lr}}(r)$$

The advantage of doing this is that ϕ_{sr}, the short range term, converges quickly in real space (effects from charges far away from r quickly decrease), while ϕ_{lr} converges quickly in reciprocal space. The Particle Mesh Ewald method takes advantage of this by calculating the ϕ_{sr} term by calculating potential directly for nearby particles, and uses a grid based direct fourier transformation to calculate the long range component.

Real space computation

The short range potential at a point can be computed by considering only particles within a small radius of the point. We can keep the radius small as ϕ_{sr} converges quickly in real space. This is also important, as we need to keep the number of particles considered less than N, in order to reduce the algorithm from $O(n^2)$ to $O(n)$ complexity. This is the case though, as the radius we consider is constant and less than the size of the simulation cell width, and we assume the particles are distributed randomly within the simulation cell.

Reciprocal space computation

The reciprocal space is the long range part of the potential computation, which converges slowly in real space. However, in reciprocal space, it converges quickly, so we use discrete fourier transformations to calculate this part of the sum.

Discrete fourier transformations require a discrete space to transform, however, our real space is continuous. So for this, we need to discretise the charges onto a grid. The approach taken in the original paper was to use Lagrangian interpolation to achieve this. Close mesh cells receive most of the charge from each particle, and this amount decreases to zero at some point, depending on the interpolation order. This is described in detail in Section 2.2.2.

An alternative to using lagrange interpolation is to use cardinal B splines. This modification is known as the *Smooth Particle Mesh Ewald* method, and is advantageous in terms of accuracy, and is also easily differentiable, which is important if the forces are required as well as the potentials. [4] This is the method that was implemented in this paper.

Periodic boundary conditions

One feature of the Particle Mesh Ewald method is the ability to simulate *Periodic Boundary Conditions* [4]. Periodic Boundary Conditions have a wrap around behaviour - that is, instead of treating the walls of

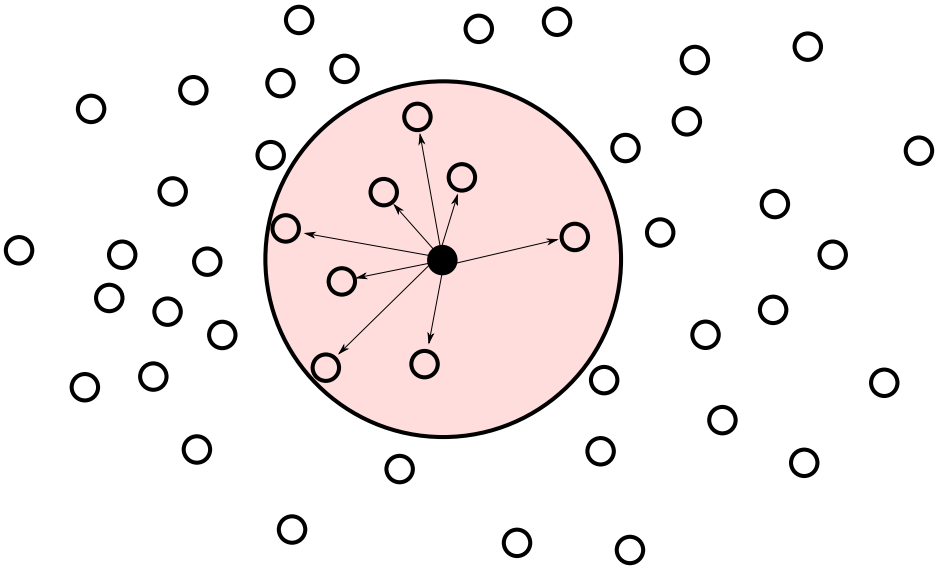


Figure 2.3: A particle and the particles that we consider it's interactions with, based on the cutoff distance

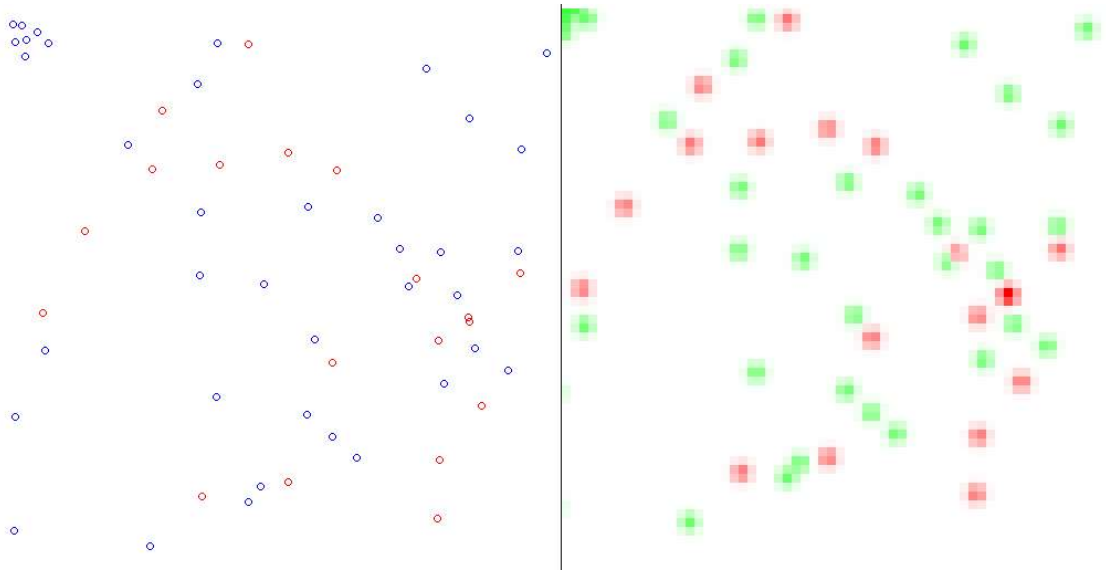


Figure 2.4: A distribution of continuous charges, and an interpretation of them as discrete charges

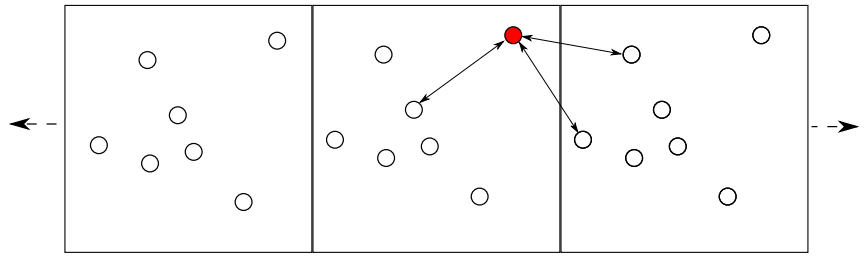


Figure 2.5: A particle interacting with it's nearest neighbours, with periodic boundary conditions in one dimension (x axis)

the simulation unit as empty space, we can allow them to loop around to the other side of the box again, effectively allowing infinite replication of the simulation unit (See Figure 2.5) This is helpful practically, as many applications are interested in the effects on a small portion of a large system [1] (For example, simulation of water at a molecular scale). Simulating the entire system would be time consuming if not impossible, and simulating only a small portion without Periodic Boundary Conditions would produce unrealistic results (Molecules may disperse into empty space over time). While useful practically, Periodic Boundary Conditions do not change the complexity or performance, and as our interest is in a comparison with the Fast Multipole Algorithm, we reduce the effects of Periodic Boundary Conditions by placing all particles within a larger, mostly empty, simulation cell. This extra padding mostly negates the effects of the Periodic Boundary Conditions.

2.2.2 Mathematical description

Interpolating the charges to the Q array

We first present the formulation of the Q array based on lagrangian interpolation.

$$Q(k_x, k_y) = \sum_{i=1}^{N} q_i W_{2p}(u_{xi} - k_x) * W_{2p}(u_{yi} - k_y) \tag{2.1}$$

Where,

- N is the number of particles,
- n_1, n_2 are integers $< N$,
- W_{2p} is a Lagrangian polynomial of order p with a value in the range $[0, 1]$,
- K is the number of cells we split the grid into,
- u_{xi}, u_{yi} are scaled fractional coordinates of particle i (Scaled fractional coordinate coordinates meaning $u_{xi} = K * (r_{xi}/(\text{simulation width}))$, r_{xi} is the particle i's x coordiante, so $0 \leq u_{xi} \leq K$).

More information can be found in [4].

We can replace W_{2p} in the above with M_n, a cardinal B spline of order n, and the formulation of the Q array is the same. This modification is known as the Smooth Particle Mesh Ewald method. From this point forward, we will use cardinal B spline interpolation. The mathematics of B spline interpolation is described in Appendix B.1.2

Calculating electrostatic potential from the Q array

With this Q array we can calculate the long range contribution to the electrostatic potential in reciprocal space.

$$E_{\text{rec}} = \frac{1}{2} \sum_{m_1=0}^{K} \sum_{m_2=0}^{K} Q(m_1, m_2) * (\theta_{\text{rec}} \star Q)(m_1, m_2) \tag{2.2}$$

With $\theta_{\text{rec}} = F(B * C)$, and so $(\theta_{\text{rec}} \star Q)(m_1, m_2) = F(B * C * F^{-1}(Q))$
Where C is the matrix for the original exponential term from Equation B.1, that is,

$$C(m_1, m_2) = \frac{1}{\pi V} \frac{\exp(-\pi^2 m^2/\beta^2)}{m^2} \text{ for } m \neq 0, C(0,0) = 0$$

Details on the derivation of this equation can be found in Appendix B.1.1, [4], [9].

Interpolating energies back from the mesh in real space

If we have a point r in real space that we wish to calculate the potential for, we can interpolate back from a mesh in the same way we did while creating the discrete mesh. That is, with $r = (r_x, r_y)$, which has scaled fractional coordinates (u_x, u_y) (see Section 2.2.2), at point r we have

$$E(r_x, r_y) = \sum_{i,j=-n}^{n} Q(\lfloor u_{x+i} \rfloor, \lfloor u_{y+j} \rfloor) * M_n(u_x - \lfloor u_{x+i} \rfloor) * M_n(u_y - \lfloor u_{y+j} \rfloor) \tag{2.3}$$

This equation iterates over cells within our interpolation order n, and calculates the weighting we ascribe to the cell ($M_n(u_x - \lfloor u_{x+i} \rfloor) * M_n(u_y - \lfloor u_{y+j} \rfloor$, a real number in the range $[0, 1]$). Summing the proportion of the grid charge assignments, we produce B Spline interpolated values from the grid.

The ewald coefficient

The ewald coefficient, β, is a number describing the ratio between the real space and the reciprocal space contributions to the calculation of the total energy. In practice, it depends on the tolerance ϵ_{tol}, and our desired cutoff distance r_{cut} in the following way, [2] [4]

$$\frac{\text{erfc}(\beta r_{\text{cut}})}{r_{\text{cut}}} \leq \epsilon_{\text{tol}} \tag{2.4}$$

Where erfc is the complimentary error function, a function that tends quickly to zero, depending on it's argument. The relation means that for $r > r_{cut}$, we have $\frac{\text{erfc}(\beta r_{\text{cut}})}{r_{\text{cut}}} \leq \epsilon_{\text{tol}}$, and for all $r > r_{\text{cut}}$ we can ignore the direct energy contribution, given by equation 2.5 [4]

$$E_{\text{dir}} = \frac{1}{2} \sum_{n} \sum_{i,j=1}^{N} \frac{q_i q_j \text{erfc} \beta |r_j - r_i|}{|r_j - r_i|} \tag{2.5}$$

Which becomes approximately

$$E_{\text{dir}} \approx \frac{1}{2} \sum_{n} \sum_{i,j=1}^{\overset{*}{N}} \frac{q_i q_j}{|r_j - r_i|} \tag{2.6}$$

Where the $*$ indicates terms with $|r_j - r_i| > r_{\text{cut}}$ are left out of the sum

2.2.3 The algorithm

Main particle mesh ewald flow

The basic flow of the algorithm is given in Algorithm 2

Data: Q: Charge assignment matrix, r_{cut}: Cutoff distance, r: position at which we calculate the potential
Initialise the ewald coefficient (Equation 2.4, algorithm in Appendix B.1.4)
Calculate the B spline coefficients (Details in [4] [9])
Allocate particles to their cells (Using a Verlet list, see section B.1.3)
Initialise the Q matrix (Equation 2.1)
Calculate reciprocal energy (Equation 2.2)
Calculate direct energy (Equation 2.5)
Interpolate reciprocal energies back to desired coordinates (Section 2.2.2)
for *Every particle p within r_{cut} of r (Calculate using Verlet list, (Section B.1.3)* **do**
  Caclulate and sum the direct potential from p at r.
end
Combine the interpolated energy and directly computed energy

Algorithm 2: The Particle Mesh Ewald Method

2.2.4 The implementation

The Particle Mesh Ewald method was implemented in Java, with a GUI front end. In this section, we break down the Particle Mesh Ewald method, and look at the interesting data structures and methods used in initialisation and potential calculations.

The Cardinal B Spline

The Cardinal B Spline was implemented by means of recursion. The class `BSpline.java` contains the methods required for evaluating the B Spline and it's derivative, as well as the Euler Exponential splines (Described in Eq. B.1). This recursion is performed many times, and is costly at high accuracies (high B Spline orders). This makes it a target for optimisation (See section 2.2.5)

Fast Fourier Transformations

The chosen library for Fast Fourier Transformations (Required for calculation of the reciprocal space potential) was the JTransforms library[1]. This library has the advantage of being implemented in pure Java code, allowing simpler integration and debugging. The Fourier Transformation operation is the core of the run time for the Particle Mesh Ewald method, and so and optimised library is essential for performance reasons.

2.2.5 Running time analysis

Cardinal B spline

With a basic implementation of the B spline, implemented directly from the recursive definition given, we notice a dramatically decreased run time. Computing the B spline coefficients is costly in terms of time, and must be computed repeatedly. A method to reduce this running time was implemented, with a HashMap saving values that had already been evaluated. This hash table had a hit rate of approximately 98%, as we evaluate fractional coordinates for the same particles often in the Particle Mesh Ewald method. The hash function converted the double keys to integer keys, which provides some collisions, but the B spline function is smooth, so similar x_1, x_2 values produce similar $M_n(x_1)$ and $M_n(x_2)$ values. The goal of this was to reduce the running time, as hashing full doubles is expensive.

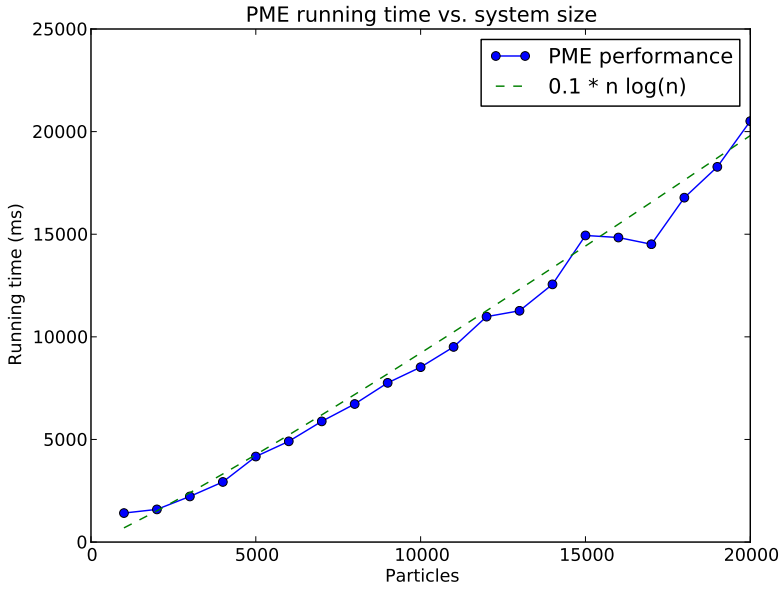


Figure 2.6: The running time of the Particle Mesh Ewald method with increasing system size

Algorithm complexity

Figure 2.6 shows the running time of the Particle Mesh Ewald method, with the hash table functions implemented. We can see that the rate at which the running time increases is proportional to $n\log(n)$. It would be easy to fit a linear function to this data also, as in this range, $\log(n)$ is a slowly growing factor compared to n.

Method analysis

We also present an analysis of the time taken by various functions in the Particle Mesh Ewald method using VisualVM[2]. (Table 2.1)

Method	Proportion of running time	Method description
`pme.initQMatrix()`	53.2%	Assigns charges to the grid using BSplines
`math.Complex.ln()`	43.6%	Calculates $\log(c)$
`math.initProximityList()`	3.2%	Compiles in range particle lists

Table 2.1: The proportion of the running time spent in each method for the Fast Multipole Algorithmwhile benchmarking.

2.2.6 Accuracy analysis

The accuracy of the Particle Mesh Ewald method was far lower than expected in practice. Total energies calculated for systems had errors ranging from 1% to 10%, depending on the system. Studying these systems led to the conclusion that the implementation of the reciprocal space calculation contains an error. It is likely that this error occurs in the generalisation from 3-Dimensions to 2-Dimensions, or in the exclusion of the self energy term [4], which were compromises that had to be made to enable a truly direct comparison between the Particle Mesh Ewald method and the Fast Multipole Algorithm. Figure **??** shows a visual representation of this discrepancy.

[1] JTransforms FFT Library, `http://sourceforge.net/projects/jtransforms`
[2] VisualVM, `http://visualvm.java.net/`

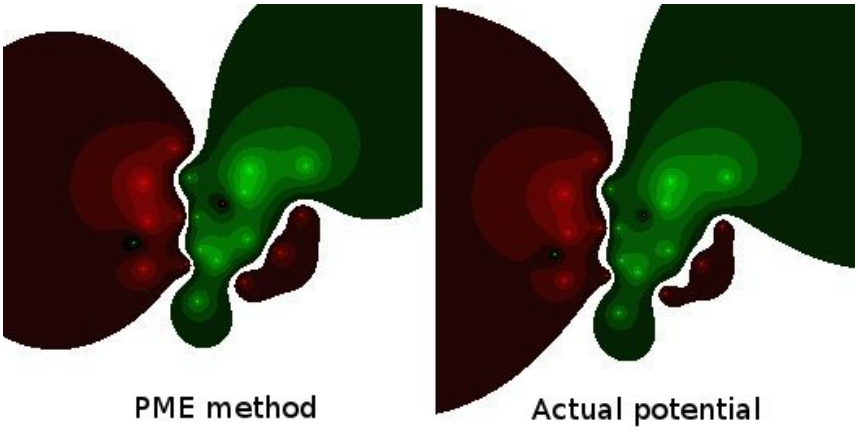


Figure 2.7: A visual comparison of the error of the Particle Mesh Ewald method. The different contours represent levels of potential.

2.3 The Fast multipole algorithm

2.3.1 Background

The Fast Multipole Algorithm is similar to the Particle Mesh Ewald method in that both use a grid structure, and bounded approximations, to speed up computation. However, they are different in the ways they use the mesh, and the way approximations are made.

Complex plane

It should be noted that for this algorithm, it is simplest to implement in two dimensions. It is possible to to implement in three dimensions, with Spherical Harmonics [7], however this is not discussed in this paper. Instead, we work in two dimensions, on the complex plane. We give a point (x, y) on the Real plane the value $z = x + yi$ on the Complex plane. This way, we can represent each point as a single number, and use complex versions of functions while calculating the multipole expansions.

Multipole expansions

A multipole expansion is a function which is a sum of a series of terms, which converges to some other function (in this case, the potential energy function). This convergence is fast, which makes it a good approximation to use in the Fast Multipole Algorithm. These multipole expansions are centered on one point, and are valid for points within a certain distance of this center, but may be shifted and combined to gain more general multipole expansions. [6]

The mesh

The mesh is similar to the one used in the Particle Mesh Ewald method. We say that the mesh has n levels, and at each level we split each cell into quarters, starting with level 0, which is the simulation cell. Each cell is split in four when moving down a level. We call the four sub cells of a cell c the *children* of c. Conversely, we call the cell that c is a sub cell of the *parent* of c.

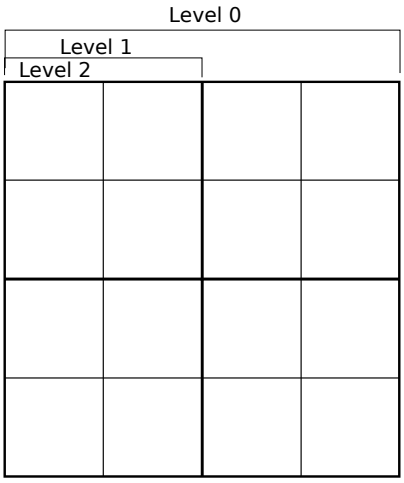


Figure 2.8: A simulation cell, with meshes up to level 2 displayed, giving $2^n = 2^2 = 4$ boxes per side

Well separated cells

For a cell c, we call a cell d *Well separated* from c if
1) Parent(d) is adjacent to Parent(c) (adjacent horizontally, vertically or diagonally)
2) d is not adjacet to c

These *well separated* cells are the ones which we will use the expansions to calculate potential for. For cells that are not well separated, we will calculate interactions directly. Figure 2.9 gives an example of well separated cells.

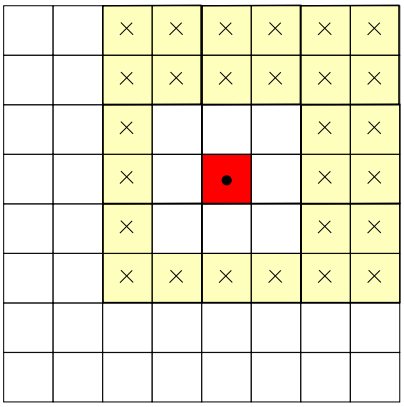


Figure 2.9: A cell, marked with a circle, and the squares that are well separated from it, marked with crosses.

2.3.2 Mathematical description

Potential and the multipole expansion approximation

We describe the potential at a point $x_0 \in \mathbb{C}$ from the charge q at $x \in \mathbb{C}$ by

$$\phi_{x_0}(x) = -q\log|x - x_0| \tag{2.7}$$

This function describes the logarithmic potential in 2-Dimensions. [6] [13].

A multipole expansion for this function is derived in [6]. Suppose that m charges of strength $\{q_i, i = 1, ..., m\}$ are located at points $\{z_i, i = 1, ..., m\}$, with $|z_i| < r$. Then for any $z \in \mathbb{C}$ with $|z| > r$, the potential $\phi(z)$ is given by

$$\phi(z) = Q\log(z) + \sum_{k=1}^{\infty} \frac{a_k}{z^k} \tag{2.8}$$

Where

$$Q = \sum_{i=1}^{m} q_i \quad \text{and} \quad a_k = \sum_{i=1}^{m} \frac{-q_i z_i^k}{k} \tag{2.9}$$

While this is exact for an infinite sum of terms, we can truncate the series at term p, and if p is large enough, this approximation is close to the actual potential $\phi(z)$. A description of how close can be found in [6]

Shifting multipole expansions

A multipole expansion's center may be shifted, which is necessary for the Fast Multipole Algorithm. Suppose that

$$\phi(z) = a_0\log(z - z_0) + \sum_{k=1}^{\infty} \frac{a_k}{(z - z_0)^k} \tag{2.10}$$

Describes a multipole expansion which is centered on z_0. We can then shift this multipole expansion to be centered at the origin,

$$\phi(z) = a_0\log(z) + \sum_{l=1}^{\infty} \frac{b_l}{z^l} \tag{2.11}$$

Where

$$b_l = \left(\sum_{k=1}^{l} a_k z_0^{l-k} \binom{l-1}{k-1} - \frac{a_0 z_0^l}{l} \right) \tag{2.12}$$

Note that this procedure of shifting to the origin is equivalent to shifting from any point a to point b, if we treat point b as the origin, and $a - b$ as our previous multipole expansion center.

Local expansions

We can find a local expansion about the origin due to a set of charges within radius R of z_0, with $|z_0| > (c + 1)R, \quad c > 1$. This local expansion is based on the multipole expansion at the same point.

$$\phi(z) = \sum_{l=0}^{\infty} b_l * z^l \tag{2.13}$$

Where

$$b_0 = \sum_{k=1}^{\infty} \frac{a^k}{z_0^k}(-1)^k + a_0 \log(-z_0) \tag{2.14}$$

2.3.3 The algorithm

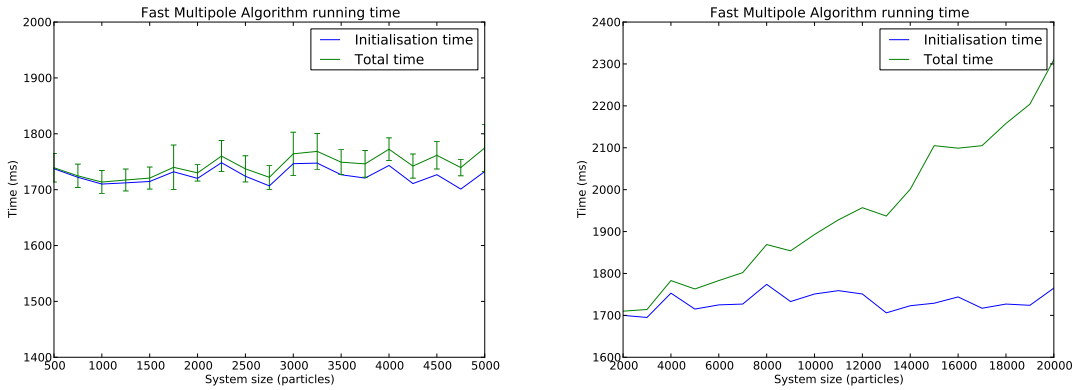
The basic flow of the algorithm is given below

2.3.4 The implementation

A more object oriented approach was taken in the implementation of the Fast Multipole Algorithm than in the Particle Mesh Ewald method. The primary classes the mathematics are `LocalExpansion.java` and `MultipoleExpansion.java`, which contain methods for creating, shifting, evaluating and storing the expansions described in Section 2.3.2.

An object for a mesh is implemented in `Mesh.java`, which contains a 2D array of cells, defined in `Cell.java`. The `Mesh` class contains a function called `makeCoarserMesh()` which creates a coarser

Data: level-count: the number of mesh levels we create, P: the set of particles, r: a point at which we wish to calculate the potential
for *i=0 to level-count* **do**
  Initialise mesh[i] by adding all particles $p \in P$ to the appropriate cells
end
for *Each cell c in mesh[level-count]* **do**
  Form a multipole expansion at at c, using Equation 2.8
end
for *i=(levelcount-1) down to 0* **do**
  for *Each cell c in mesh[i]* **do**
    Shift each multipole expansion for the child cells of c (in mesh[i+1]) to c (Eq. 2.11);
    Combine and save these multipole expansions by addition;
  end
end
for *Each cell c in mesh[0]* **do**
  Form a local expansion at c based on c's multipole expansion;
  Shift c's local expansion to the children of c;
end
for *i=1 to level-count* **do**
  for *Each cell c in mesh[i]* **do**
    Shift c's local expansion to the children of c;
  end
end
Set the cumulative potential to 0;
for *Each cell c in mesh[level-count]* **do**
  if *c is Well Separated from d's cell* **then**
    Calculate the potential at r from c's local expansion;
    Add this potential to the cumulative potential;
  else
    Calculate the potential at r from c's particles directly;
    Add this potential to the cumulative potential;
  end
end
Return the cumulative potential;



(a) Range 0 - 2000

(b) Range 0 - 20,000

Figure 2.10: The Fast Multipole Algorithm's performance over the group of systems in the range 0 to 5,000 particles, and 0 to 20,000.

Method	Proportion of running time	Method description
`fma.LocalExpansion()`	31.6%	Local expansion initialisation
`math.Complex.power()`	27.8%	Calculates c^x
`math.Complex.ln()`	24.9%	Calculates $\log(c)$
`math.Binomial.binomial()`	9.6%	Calculates binomial function
`fma.MultipoleExpansion.add()`	1%	Adds multipole expansions

Table 2.2: The proportion of the running time spent in each method for the Fast Multipole Algorithmwhile benchmarking.

mesh, shifting and merging multipole expansion in the process. This coarser mesh is saved into an array of each mesh level, from the maximum depth, to zero.

2.3.5 Running time analysis

Complexity analysis

The Fast Multipole Algorithm was run over the class of problems with sizes in the range [0,5000] with an increment of 250, and [0, 20000] with an increment of 1000. The results are shown in Figure 2.10.

The graphs here show the initialisation time and total time for the Fast Multipole Algorithm on a set of problems. The initialisation time is the time required to compute the multipole and local expansions, and the total time is the initialisation time, with the time taken to evaluate these expansions included.

Figure 2.10(a) shows the same range as Figure 2.2, the basic algorithm's complexity. In it, we see that the initialisation time is taking most of the computation time. Each data point is an average of 20 trials, with the error bar indicating the standard deviation.

Figure 2.10(b) shows the Fast Multipole Algorithm's run time for much larger problems, with up to 20,000 particles. We can see the initialisation time stay constant, while the total time grows linearly. So, we confirm that our implementation of the Fast Multipole Algorithmis in the complexity class $O(n)$

Method analysis

Using `VisualVM` [3], the table below of running times spent in each method was produced.

We can see that most of the running time is spent forming local expansions, and doing complex arithmetic. The run time is split over several different methods, which indicates there are no severe efficiency concerns. Most of the run time is spent performing complex arithmetic, which is what should be expected.

[3] VisualVM, `http://visualvm.java.net/`

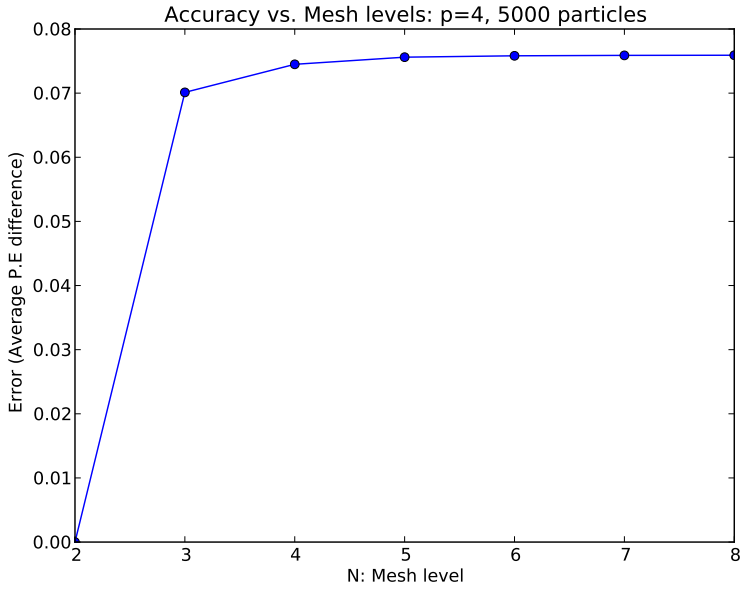


Figure 2.11: The relationship between N, mesh levels, and the error, when compared with the basic algorithm.

2.3.6 Accuracy analysis

The Fast Multipole Algorithm has two main parameters that control accuracy - the number of mesh levels, and the number of terms in the multipole expansion. We call the number of mesh levels N and the number of terms in the multipole expansion p. For reasonable accuracy, we should expect $N \approx \log_4(n)$, where n is the number of particles, and $p \approx \log_2(\epsilon)$, where ϵ is the desired precision. [6]

Maximum mesh level

We plot the accuracy of the method as a function of these two parameters. In figures 2.11 and 2.12, we demonstrate the effect the mesh level has on the error.

Figure 2.11 shows the error *increasing* with mesh level. This seems counter intuitive, however, according to The Fast Multipole Algorithm from Section 2.3.3, we calculate interactions directly for cells that are not well separated (See Section 2.3.1). With low mesh levels, such as $N = 2$, this would include the entire simulation unit, which leads to the error of 0 we see at $N = 2$. Increasing from here, we see the error level out to approximately 0.075. This graph shows that increasing maximum mesh level does not necessarily lead to improved accuracy.

Figure 2.12 shows the effect of an increasing system size for a fixed N. We can see that the error grows linearly as we increase the number of particles if we keep N and p constant. From this we can confirm that a good rule is $N \approx \log_4(n)$, with n being the total number of particles in our system. This is recommended, as it keeps the number of particles per cell approximately constant.

Number of terms

Next we examine the effect the number of terms we truncate our multipole expansion at, p, has on the accuracy of our simulation.

From Figure 2.13 we see that the accuracy of the Fast Multipole Algorithm increases exponentially as we increase the number of terms in our expansion. However, after $N \approx 20$, the improvement slows down. This is because we reach the machine precision in our accuracy at these levels.

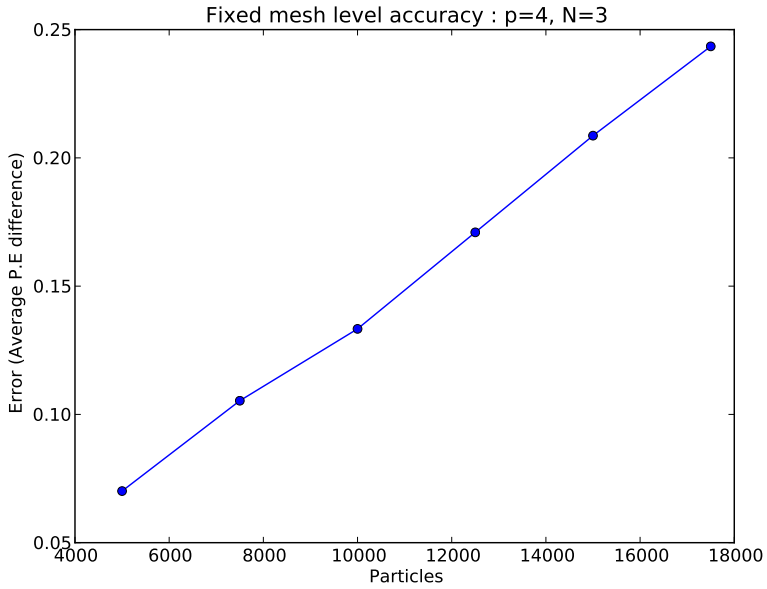


Figure 2.12: The error for the Fast Multipole Algorithmwith a fixed mesh level $N = 3$, with an increasing system size

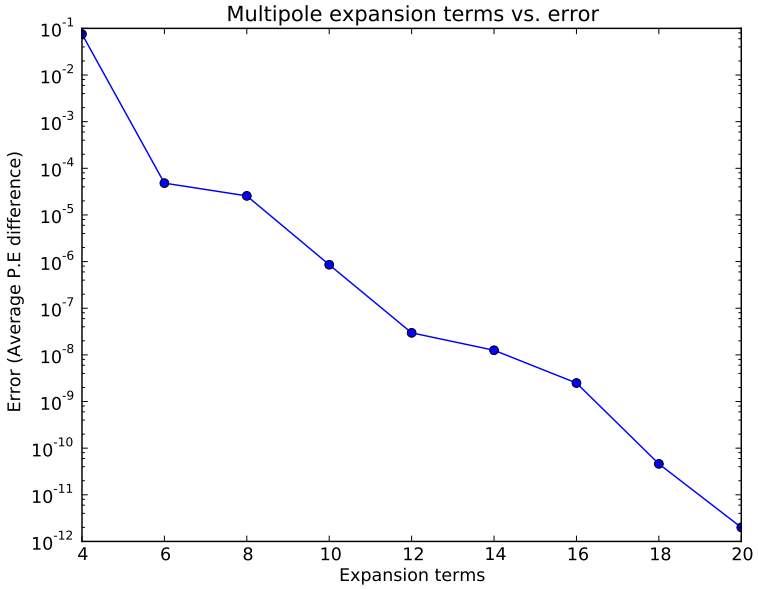


Figure 2.13: The error, on a logarithmic scale, vs. the number of terms in the multipole expansion, with a system size of 5000 particles

Chapter 3

Comparison of the algorithms

3.1 Comparison of the running times

The implementations were run on the same sets of data for different running times. The results of this experiment are shown in Figure 3.1.

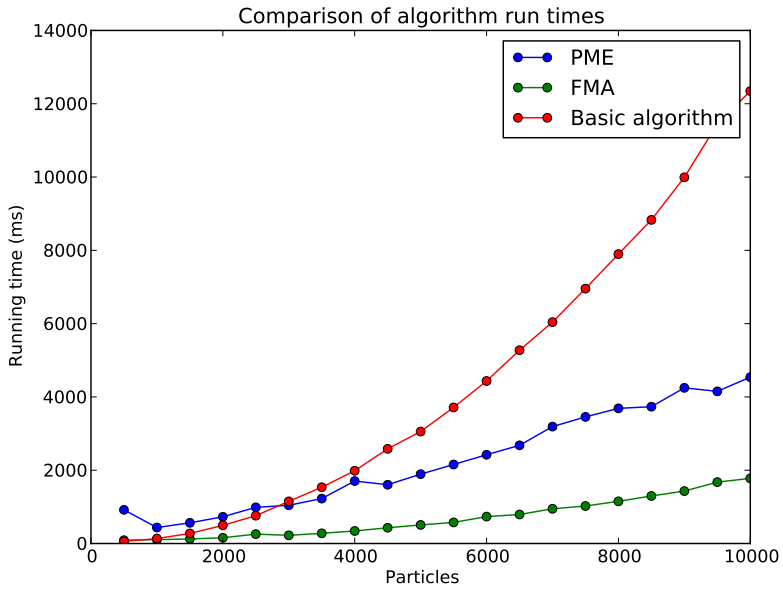


Figure 3.1: A comparison of the running times for each algorithm, set to give accuracies on the order of 10^{-12}, with increments of 500 particles, going up to 10,000 particles.

The simulations were run on the systems ranging from 100 to 10,000 particles, and were set to give accuracies on the order of 10^{-12} (in the case of the Particle Mesh Ewald method, parameters were set to give a theoretical accuracy of 10^{-12} to ensure fairness in the experiment).

We can see from the graph that after a certain point, the implementations are ranked in the order of the Basic algorithm, the Particle Mesh Ewald method, and the Fast Multipole Algorithm. Below $N = 3000$ however, we see that the Basic algorithm is faster than the Particle Mesh Ewald method. After $N = 3000$, the Particle Mesh Ewald method becomes faster.

To determine which algorithm is fastest for $N < 1000$, a higher resolution experiment was performed on these values. Figure 3.2 shows that the crossover point for the Fast Multipole Algorithm and the Particle Mesh Ewald method occurs at approximately $N = 750$. This is a small system size when it comes to practical simulations.

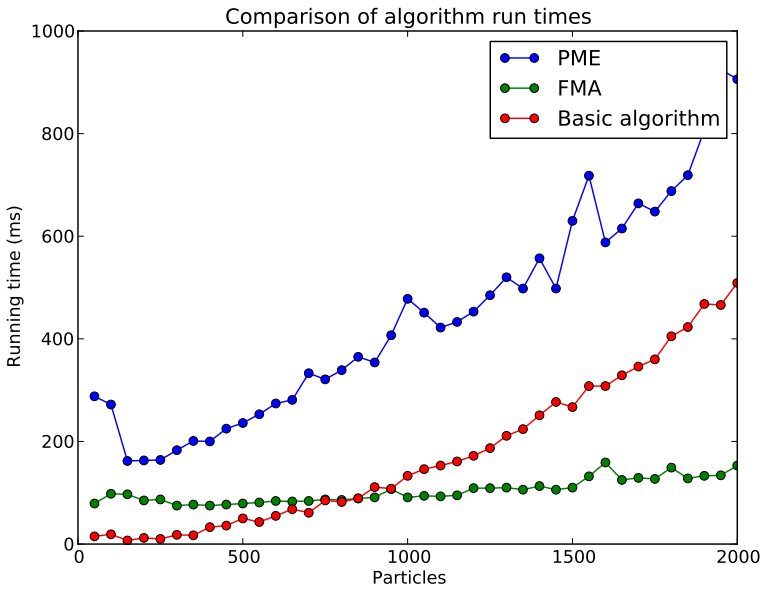


Figure 3.2: A comparison of the running times for each algorithm, set to give accuracies on the order of 10^{-12}, with increments of 50 particles, going up to 2000 particles.

3.2 Discussion

3.2.1 The Particle Mesh Ewald method

Advantages

The Particle Mesh Ewald method has several features that differentiate it from the Fast Multipole Algorithm. The first of these is Periodic Boundary Conditions (discussed in Section 2.2.1). Periodic Boundary Conditions allow simulation of large spaces realistically, with no reduction in performance. For practical applications, such as chemistry, boundary condition effects are dramatically reduced.

In addition to this, the Particle Mesh Ewald method as it is described in [2] is described in three dimensions. The Fast Multipole Algorithm requires the addition of Spherical Harmonics for the algorithm to be extended to 3-Dimensions [3], which increases the complexity of the algorithm, and the decreases the algorithm's efficiency.

Another advantage of the Particle Mesh Ewald method is it's suitability for parallel execution. Most of the calculations performed in the Particle Mesh Ewald method are independent of each other, and can be easily parallelised with a small amount of overhead (for example, the NAMD [14] molecular dynamics simulation software). Parallel methods for calculating the the Fourier Transformations are well explored [9].

Disadvantages

For some purposes, the Periodic Boundary Conditions introduce error rather than introduce it. This is the case for small systems, that should not tile infinitely in all dimensions. This was the case for the implementation presented in this paper, and padding was required to reduce the effects of the Periodic Boundary Conditions.

The mathematics of the Particle Mesh Ewald method is complex and intricate. This makes debugging, writing and maintaining software implementing the Particle Mesh Ewald method more difficult than for the Fast Multipole Algorithm.

While the Particle Mesh Ewald method has great potential to be optimised, it is easy to write suboptimal implementations [1]. In place FFT, and careful use of floating point operations should be used to guaran-

tee optimal performance [9]. This was not implemented in this project, as readibility was preferred over marginal speed improvements.

3.2.2 The Fast Multipole Algorithm

Advantages

The advantage of the Fast Multipole Algorithm is it's $O(n)$ computational complexity, with a simple algorithm. We can see this in the graphs from Section 3.1, where the Fast Multipole Algorithm performs best of the three algorithms examined.

Disadvantages

The Fast Multipole Algorithm is not as simple to implement in 3-Dimensions, as it requires speherical harmonics [3]. This brings extra computational time, and leads to a more complicated implementation.

The Fast Multipole Algorithm is not as well suited to parallelisation as the Particle Mesh Ewald method is. The nature of the algorithm requires a tree structure, with each level dependent on the lower and higher expansions recursively.

3.2.3 The Basic Algorithm

3.2.4 Advantages

The Basic Algorithm is the fastest for systems with sizes in the range 0 to 750 (see Figure 3.2). For small systems, this is the best choice of algorithm, as it has no initialisation time, and it's error is restricted only by the machine precision.

The Basic Algorithm is also well suited to parallelisation, and a parallel extension of the algorithm would be trivial, as particle-particle potential calculations are independent of each other.

3.2.5 Disadvantages

The obvious disadvantage to the Basic Algorithm is it's complexity of $O(n^2)$, which is inhibitive for most useful simulations. For molecular simulation, it is virtually impossible to use this approach.

3.3 Conclusion

For the system sizes tested, the Basic Algorithm was the fastest up to a system size with approximately 750 particles, and the Fast Multipole Algorithm was consistently the fastest after this, by a factor of roughly 2 to 3. The Fast Multipole Algorithm is recommended if algorithmic simplicity is sought, if the simulation environment is 2-Dimensional, or if a multi cpu architecture is not used.

However, the Particle Mesh Ewald method enjoys many advantages that cater it to high perfomance scientific computing, and particularly computational chemistry. While the Particle Mesh Ewald method is slower in this implementation, other implementations with a focus on optimisation could make the Particle Mesh Ewald method competitive with the Fast Multipole Algorithm.

Appendix A

Implementation

A.1 Benchmark computer details

CPU	Intel(R) Core(TM) i7 CPU 950 @ 3.07GHz
L1 Cache	256KB, 4-way set associative, write-through
L2 Cache	1024KB, 8-way set associative, write-through
L3 Cache	8192KB, 16-way set associative, write-back
Main memory	6GB DDR3 1333Mhz
Mflops/s	64.424 Mflops/s

Table A.1: The relevant specifications for the computer used to benchmark the algorithms presented in the paper.

Appendix B

Algorithms and Mathematics

B.1 The Particle Mesh Ewald method

B.1.1 Derivation of the reciprocal energy formula

The reciprocal space contribution to the electrostatic energy can be written as,

$$E_{\text{rec}} = \frac{1}{2 * \pi * V} \sum_{m \neq 0} \frac{\exp(-\pi^2 m^2 / \beta^2)}{m^2} B(m_1, m_2) S(m) S(-m) \tag{B.1}$$

Where

- V is the volume (or in the two dimensional case, area) of the simulation cell,
- β is the ewald coefficient,
- S is the structure factor.
- B is the matrix of B spline inverse fourier transform moduli, $B(m_1, m_2) = |b_1(m_1)|^2 * |b_2(m_2)|^2$. More detail on this can be found in [4].

It is shown in [4] that $S(m) \approx F(Q(m))$, so we can rewrite this as a convolution
With this Q array we can calculate the long range contribution to the electrostatic potential in reciprocal space.

$$E_{\text{rec}} = \frac{1}{2} \sum_{m_1=0}^{K} \sum_{m_2=0}^{K} Q(m_1, m_2) * (\theta_{\text{rec}} \star Q)(m_1, m_2) \tag{B.2}$$

With $\theta_{\text{rec}} = F(B * C)$, and so $(\theta_{\text{rec}} \star Q)(m_1, m_2) = F(B * C * F^{-1}(Q))$ [4] [9]
Where C is the matrix for the original exponential term from Equation B.1, that is,

$$C(m_1, m_2) = \frac{1}{\pi V} \frac{\exp(-\pi^2 m^2 / \beta^2)}{m^2} \text{ for } m \neq 0, C(0,0) = 0$$

B.1.2 Evaluation of the B spline

The recursive method for the calculation of the B spline is included here for completeness. See [4] [9] for more details.

The B spline is defined by the following recurisve formula,

$$M_2(x) = 1 - |x - 1| \text{ for } 0 \leq x \leq 2, \qquad M_2(x) = 0 \text{ for } u < 0 \text{ or } u > 2.$$

$$M_n(x) = \frac{x}{n-1} M_{n-1}(x) + \frac{n-x}{n-1} M_{n-1}(x-1)$$

The B spline's derivatives can also be calculated analytically. This is used in the force calculation.

$$\frac{d}{dx} M_n(x) = M_{n-1}(x) - M_{n-1}(u-1)$$

B.1.3 Verlet list algorithm

The following algorithm utilises a list known as a *Verlet list* to keep track of which particles are within a cutoff distance of each other, with only $O(n)$ complexity. It is used in the direct energy calculation of the Particle Mesh Ewald method

After this, the array closeParticles[x][y] contains all particles within r_{cut} distance from a particle contained

for *i=0 to N* **do**
  cell$_x := \lceil u_{xi} \rceil$; (With u_{xi} the scaled fractional x coordinate, as defined in Equation 2.1))
  cell$_y := \lceil u_{yi} \rceil$; (With u_{yi} the scaled fractional y coordinate))
  direct range := $\lceil r_{cut}/$mesh cell width$\rceil$
  for $\delta_x = -$*direct range to* $+$*direct range* **do**
    for $\delta_y = -$*direct range to* $+$*direct range* **do**
      if *cell*$_x + \delta_x$ *and cell*$_y + \delta_y$ *are in the mesh* **then**
        Add i to the array closeParticles[cell$_x + \delta_x$][cell$_y + \delta_y$]
      end
    end
  end
end

in mesh cell x, y.

B.1.4 Ewald coefficient estimation algorithm

The following algorithm uses a binary search to determine the best value of β to use as the ewald coefficient by a binary search. Our goal is to satisfy the inequality in B.3 as closely as possible by modifying β.

$$\frac{\text{erfc}(\beta r_{\text{cut}})}{r_{\text{cut}}} \leq \epsilon_{\text{tol}} \tag{B.3}$$

Data: ϵ_{tol}: a tolerance value for our erfc term, r_{cut}: the chosen cutoff distance that is the border between direct and mesh based computation, c: some constant indicating how precise we wish the ewald coefficient to be.

$\beta_{\text{low}} := 0$
$\beta_{\text{high}} := 1$
while $\frac{erfc(\beta_{high} r_{cut})}{r_{cut}} < \epsilon_{tol}$ **do**
  $\beta_{\text{high}} := \beta_{\text{high}} * 2$
end

(At this point, $\beta_{\text{low}} \leq \beta \leq \beta_{\text{high}}$.)

for *i:=0 to c* **do**
  $\beta := (\beta_{\text{high}} + \beta_{\text{low}})/2$
  if $\frac{erfc(\beta r_{cut})}{r_{cut}} > \epsilon_{tol}$ **then**
    $\beta_{\text{high}} := \beta$ **else**
      $\beta_{\text{low}} := \beta$
    end
  end
end
return β

Bibliography

[1] John A. Board Jr. Abdulnour Y. Toukmaji. Ewald summation techniques in perspective: a survey. *Computer Physics Communications*, 95:73–92, 1996.

[2] Tom Darden, Darrin York, and Lee Pedersen. Particle mesh ewald: An n log(n) method for ewald sums in large systems. *The Journal of Chemical Physics*, 98(12):10089–10092, 1993.

[3] T.F. Eibert. Antennas and propagation, ieee transactions on. 53(2):814 – 817, feb. 2005.

[4] Ulrich Essmann, Lalith Perera, Max L. Berkowitz, Tom Darden, Hsing Lee, and Lee G. Pedersen. A smooth particle mesh ewald method. *The Journal of Chemical Physics*, 103(19):8577–8593, 1995.

[5] P. Ewald. *Ann. Phys.*, 24:325–348, 1921.

[6] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *Journal of Computational Physics*, 1987.

[7] Alexander T. Ihler. An overview of fast multipole methods, 2004.

[8] Max Planck institute of Astrophysics. The millennium simulation project. `http://www.mpa-garching.mpg.de/galform/virgo/millennium/`. Accessed: 10/10/2012.

[9] Sam Lee. An fpga implementation of the smooth particle mesh ewald reciprocal sum compute engine (rsce). 2005.

[10] Sir Isaac Newton. Newton's principia: the mathematical principles of natural philosophy. 1846.

[11] Henrik G. Petersen. Accuracy and efficiency of the particle mesh ewald method. *The Journal of Chemical Physics*, 103(9):3668–3679, 1995.

[12] Henri Poincarè. Les mthodes nouvelles de la mcanique cleste. 3, 1899.

[13] L. Samaj. *J. Phys. A: Math. Gen.*, 36:5913, 2003.

[14] Theoretical and Computational Biophysics group. Namd scalable molecular dynamics. `http://www.ks.uiuc.edu/Research/namd/`. Accessed: 10/10/2012.