6-2007

Parallel Fast Multipole Method for Molecular Dynamics

Reid G. Ormseth

Parallel Fast Multipole Method
For Molecular Dynamics

THESIS

Reid G. Ormseth, Captain, USAF

AFIT/GAP/ENP/07-J02

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

The views expressed in this document are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the United States Government.

AFIT/GAP/ENP/07-J02

Parallel Fast Multipole Method For Molecular Dynamics

THESIS

Presented to the Faculty

Department of Engineering Physics

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science in Applied Physics

Reid G. Ormseth, BS, MS

Captain, USAF

June 2007

AFIT/GAP/ENP/07-J02

Parallel Fast Multipole Method For Molecular Dynamics

Reid G. Ormseth, BS, MS
Captain, USAF

Approved:

David E. Weeks (Chairman)	Date
Ruth Pachter (Member)	Date
Gary B. Lamont (Member)	Date

AFIT/GAP/ENP/07-J02

Abstract

We report on a parallel version of the Fast Multipole Method (FMM) [18] implemented in the classical molecular dynamics code, NAMD (Not Another Molecular Dynamics program) [34]. This novel implementation of FMM aims to minimize inter-processor communication through the modification of the FMM grid to match the hybrid force and spatial decomposition scheme already present in NAMD. This new implementation has the benefit of replacing all-to-all communications broadcasts with direct communications between nearest neighbors. This results in a significant reduction in the amount of communication compared to earlier attempts to integrate FMM into common molecular dynamics programs. The early performance of FMM is similar to the existing electrostatics methods already in NAMD. It also provides an improvement in the parallel scaling of previous implementations of FMM in NAMD. In addition, tests of the stability and accuracy of the FMM algorithm in molecular dynamics as applied to several common solvated protein structures are discussed.

Acknowledgements

First and foremost, I owe a large debt of gratitude to Dr. Pachter and everybody in MLPJ who has supported me in this work. I also wish to express thanks for the Aeronautical Systems Center Major Shared Resource Center (ASC-MSRC) for the support of CPU time on their systems.

Significant help has also been provided by 'The Art of Molecular Dynamics Simulation' by Dennis Rapaport. This work is the clearest treatment of the Fast Multipole Method in three dimensions, and some of the code provided in his book has been adapted here to handle the core mathematical functions.

Finally, I would like to express great thanks for the Theoretical and Computational Biophysics Group in the Beckman Institute for Advanced Science and Technology at the University of Illinois at Urbana-Champaign who have produced NAMD2, the best Molecular Dynamics program, and made the source freely available for all who wish to attempt to improve upon their labor.

All the work presented was supported by the United States Air Force (USAF) Air Force Research Laboratory (AFRL) Materials and Manufacturing Directorate, AFRL/ML.

Reid G. Ormseth

Table of Contents

Page

List of Figures

List of Tables

List of Abbreviations

List of Symbols

Parallel Fast Multipole Method For Molecular Dynamics

I. Introduction

There is significant interest today to understand the function of materials at the nano-scale, from high performance polymers to common biological structures such as proteins. Research efforts are underway to understand the role of biological materials and mechanisms in diseases and metabolic function not only to aid drug design, but also in an attempt to leverage biological structures in the development of biomimetic materials, such as super-hydrophobic surfaces or optical coatings. Applications of biomimetic materials include aircraft structure and lubricants, photonic materials and sensor design [47].

One of the primary methods of exploring the theory of these materials is though computational modeling of molecular dynamics, known as MD. Molecular Dynamics can allow researchers to study protein function by simulating kinetics, looking at shifts in structure conformation due to environmental effects, or studying potential active sites that can be useful in drug design [48]. In molecular dynamics simulations, atoms are treated classically, and explicit bonds are predefined. By omitting the quantum mechanics and chemical kinetics, systems with as many as several hundred thousand atoms can be modeled, including entire proteins in explicit solvent. Related coarse-grained modeling uses the same programs and techniques in attempts to study larger structures such as entire cells [17].

Most natural proteins have between 50 and 2000 amino acid residues, or about one thousand to forty thousand atoms [53]. They do not maintain their structure in vacuum, but are stabilized by their environment, commonly water, membranes, or other proteins. When simulations are placed in a box of an explicit solvent, such as water, this results in system sizes between twenty thousand and a half million

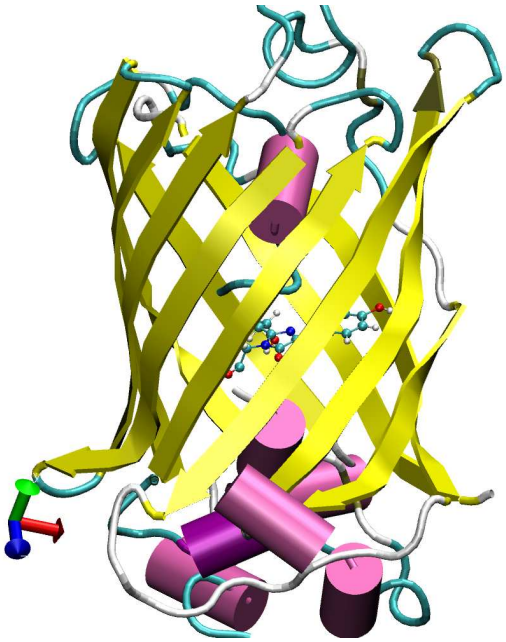


Figure 1.1: The Green Fluorescent Protein with the chromophore inside of a protective β-barrel.

atoms, with most common problems currently studied in the literature around fifty to a hundred thousand atoms. Some recent papers have modeled systems of over a million particles, and there is a significant push to attempt to model long strands of DNA [54].

The basic molecular dynamics model calculates a number of forces, frequently parameterized representations of the natural forces, including atomic bonds, Van der Wahl's and electrostatic forces (see Section 2.1). In the case of a covalent bond, because an atom is only bound to a small number of other atoms, the contributions to the forcefield are easily calculated, even for very large systems. But, for the non-bonded terms, specifically the Van der Wahl's and electrostatic forces, each atom interacts with every single other atom. This calculation very easily dominates the calculation time; in most current molecular dynamics programs, it will generally be around 80% of the run time. As a result, it is the non-bonded portion of the calculation that is usually targeted for performance improvements.

There are a number of methods designed to handle these non-bonded calculations, including cutoff methods, particle mesh methods and tree methods. One particular type of tree method, the Fast Multipole Method (FMM) was introduced in 1987 by Greengard and Rokhlin [18] as a method for calculating long range two body interactions such as electrostatic or gravitational forces (see Section 2.3). FMM was the first algorithm for this class of problem with linear scaling, which is the advantage of being able to scale to very large number of particles without significant penalty.

There are a number of molecular dynamics programs in common use today, including CHARMM [29], LAAMPS [39], GROMACS [27], Amber [11] and others. While all of these programs utilize a number of different techniques for the calculation of electrostatic forces, none of them implement an algorithm that provides linear scaling to handle extremely large systems. All these codes include a cutoff method, because it is easy to write and extremely fast, but is not accurate enough for biomolecules. Most of these codes also include particle mesh methods, because they are generally easier to implement than FMM. This ease of implementation yields better serial run times for small to medium systems.

In an effort to implement a linearly scaling algorithm capable of handling larger simulations, the Fast Multipole Method was integrated into a commonly used, parallel, molecular dynamics program. The objective is to provide linear scaling with problem size and parallel performance on par with the current electrostatics implementation.

The package NAMD version 2.5 was chosen in which to implement the Fast Multipole Method. NAMD, which stands for Not Another Molecular Dynamics code, was developed at the Theoretical and Computational Biophysics Group at the University of Illinois at Urbana Champaign (UIUC) [23]. It was chosen because it is stable, well documented, fast, and parallelizes very well to large numbers of processors. It

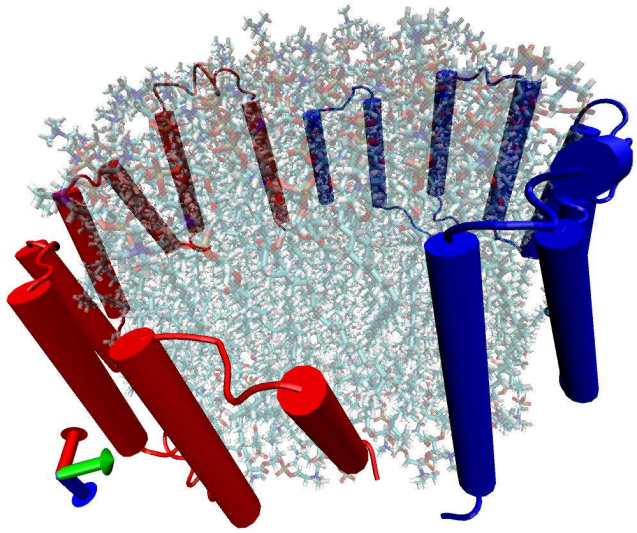


Figure 1.2: The Apolipoprotein A-I (ApoA1) Protein embedded in a lipid bilayer.

is also well integrated with the VMD (Visual Molecular Dynamics) program, utilized for analysis and visualization [20].

The two reference systems used in this study include the Green Fluorescent Protein (GFP), which has many common uses including medical imaging and drug discovery [58] (see Figure 1.1). GFP is composed of 238 amino acids and is barrel shaped, approximately 40 Å long and 30 Å in diameter, and protects a chromophore buried in the middle of the cylinder. The other reference system, Apolipoprotein A-I (ApoA1), is a large dimer membrane protein with 196 residues and 3205 atoms to each protein. The two proteins form a quaternary structure shaped roughly like a ring (see Figure 1.2), and is the protein constituent for HDL "good" cholesterol [37]. This system is the standard NAMD benchmark [36], with 6410 atoms in the protein, 21,440 atoms in a lipid bilayer in the middle of the ring and solvated in water for a total system size of just over 92 thousand atoms. Simple water boxes and water spheres were also used to provide a large number of similar systems with different sizes to model performance as particle count increases.

II. Theory and Background

Molecular Dynamics is a method of simulating substances at the atomic level through the use of classical mechanics that was initially developed in the 1960s, and has been around in its current form since the early 1980s [43, 48]. The first major development was the MM2 program developed by Allinger in the mid-seventies [4], with the field advancing dramatically since then.

Understanding material properties and function is a multi-scale problem that involves the understanding of everything from electronic wavefunctions to the bulk properties of molar quantities of a substance. Solving most problems at a quantum level, through techniques such as Time Dependent Density Functional Theory (TD-DFT), Hartree-Fock (HF) or Multiconfiguration Self Consistent Field (MCSCF) quickly becomes computationally difficult. Even the more efficient of these methods, such as TD-DFT, are impractical for systems of much larger than a couple hundred atoms, and time dependent calculations are restricted to very short time scales [30]. Molecular Dynamics provides an atomic level of detail and can yield insight without needing to solve a very large many-body problem at the computationally expensive, quantum level of theory.

2.1 Molecular Dynamics

Molecular Dynamics typically involves simulating a classical system as a series of point masses interconnected with a series of harmonic potentials representing covalent bonds. The exact form of this potential can be expressed in a number of ways, and any one form is referred to as a *forcefield*. NAMD supports the X-PLOR family of

forcefields, including CHARMM22 [29] which has the form:

$$\begin{aligned}\Phi = & \sum_{bonds} k_b(b-b_0)^2 + \sum_{angles} k_\theta(\theta-\theta_0)^2 + \sum_{dihedrals} k_\phi\left[1+\cos(n\phi-\delta)\right] \\ + & \sum_{impropers} k_\omega(\omega-\omega_0)^2 + \sum_{Urey-Bradley} k_u(u-u_0)^2 \\ + & \sum_i \sum_j \left(\epsilon \left[\left(\frac{R_{min_{ij}}}{r_{ij}} \right)^{12} - \left(\frac{R_{min_{ij}}}{r_{ij}} \right)^{6} \right] + \frac{q_i q_j}{\epsilon_0 r_{ij}} \right). \end{aligned} \tag{2.1}$$

where b is bond length, θ is bond angle, ϕ is a dihedral angle, ω is an improper angle, and Φ is the total potential. The Urey-Bradley constraint, u, is a cross term that is generally unused. The first five terms, all bonded parameters, are harmonic oscillators parameterized to represent the forces arising from covalent chemical bonds. As shown in Figure 2.1, the bond length parameter is a harmonic potential between two directly atoms that have been predefined has having a covalent bond. The bond angle is defined as the angle between two bonds that connect to a common atom. The dihedral angle is measured along four linearly bonded atoms, and is defined as the angle between the plane formed by the first three atoms in the sequence (atoms 1-2-3) and the plane formed by the last three atoms in the sequence (atoms 2-3-4.) The improper angle also involves four atoms and is defined as the angle between the plane formed by three atoms with a fourth atom. It is used to measure the height of a atom such as the nitrogen in an ammonia structure. These parameters are *bonded* terms, the sums over these parameters are only over the small number of atoms to which a given atom is covalently bonded, or connected through one or two intermediary covalent bonds. As the number of atoms in a system increases, the number of these bonds increases linearly; leading to $O(N)$ scaling of computation of the bonded parameters.

The two final terms in Equation 2.1 represent the two body *non-bonded* interactions. In general, these require N^2 calculations to perform, where N is the number of

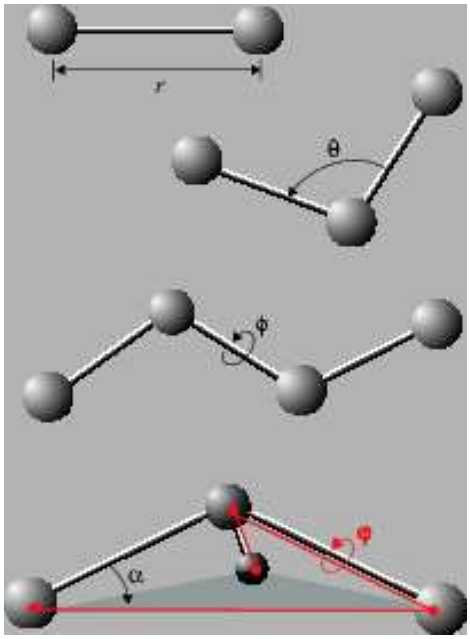


Figure 2.1: The atom bond parameters for the CHARMM forcefield include bond length (here r), bond angle θ, dihedral angle ϕ, and improper angle α. (Reproduced from [31].)

particles in the simulation. The first of these terms is the Lennard-Jones interaction,

$$\Phi_{LJ} = \sum_{i,j} \epsilon \left[\left(\frac{R_{min_{ij}}}{r_{ij}} \right)^{12} - \left(\frac{R_{min_{ij}}}{r_{ij}} \right)^{6} \right], \tag{2.2}$$

which approximates the Van der Wahl's potential between two atoms, where $R_{min_{ij}}$ is the equilibrium distance between two types of atoms, and r_{ij} is the actual distance between the center of mass of the two atoms. The second non-bonded term, representing electrostatic interactions, is a direct Coulomb sum over the point charges q_i representing each atom:

$$\Phi_{elec} = \sum_{i,j} \frac{q_i q_j}{\epsilon_0 r_{ij}}, \tag{2.3}$$

where the charge is the result of the residual charge that is a result of averaging the electronic wavefunctions at the center of mass of the atom. These are parameterized through quantum level simulation of similar chemical complexes.

Because the electronic interactions between bonded atoms are accounted for in the bonds, the electrostatic and Lennard-Jones interactions are generally excluded for directly bonded atoms and atoms bonded to a common intermediate atom (commonly notated as 1-2 and 1-3 interactions, respectively). For four atoms connected by linearly successive bonds, the Lennard-Jones interaction between the first and fourth atoms (1-4 bonds) may be excluded as well, and many forcefields allow for the attenuation of the 1-4 non-bonded interactions.

2.2 Challenges for Large-Scale and Long-Time Molecular Dynamics

2.2.1 Cutoff Calculations and Multiple Time Stepping. The $O(N^2)$ nature of the non-bonded force calculations make them prohibitively expensive for all but the smallest of systems. Initial efforts at reducing computational time involved a cutoff scheme, where the electrostatic interactions between atoms separated more than a given distance were not calculated. But, this type of abrupt cutoff leads to significant non-linearities in the potential, destabilizing dynamics. Figure 2.2 shows the Lennard-Jones potential, with the dotted line showing a gradually smoothed form, with divergence beginning at a specified switch distance, reaching zero at the cutoff distance. Given its rapidly decaying $1/r^6$ nature, this scheme is generally satisfactory for the Lennard-Jones potential, with a cutoff usually on the order of 9-15 Å. However, the $1/r$ form of the Coulomb potential does not approach zero as quickly, giving it a longer range nature. As a result, cutoff methods inappropriately truncate the electrostatic forces, and are generally ineffective for modeling biomolecules [40, 41].

Another widely used technique for overcoming the problem of the computational difficulty of calculating electrostatic interactions is Multiple Time Stepping (MTS) [52]. An all atom potential is calculated only once every several time steps, and a distinction is made between quickly varying short range and slowly varying long range interactions. A time integration technique, commonly the Velocity-Verlet/r-RESPA method, is used to add an impulse for an all-atom potential over many timesteps, and a cutoff calculation is used for short range calculations for intermediate timesteps.

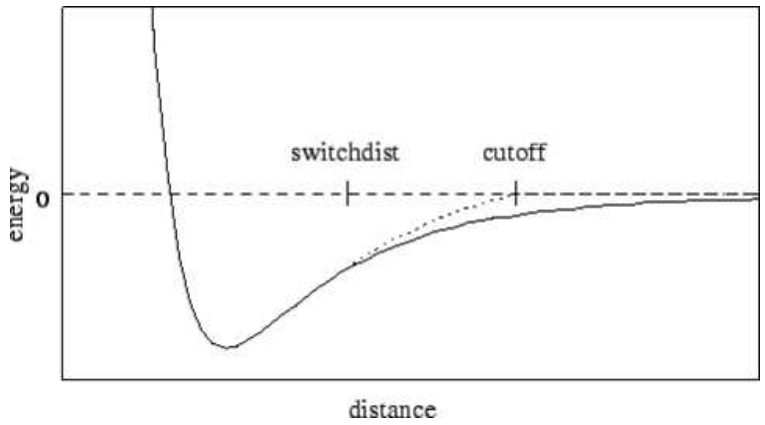


Figure 2.2: The Lennard-Jones potential. Many calculations will include an artificial cutoff, where the potential is gradually smoothed to zero between a given switch distance and the cutoff distance. (Reproduced from [7])

Care has to be taken to avoid resonance between the slowly varying timestep and vibrational modes of the system [8]. With the fastest bond-stretch being the H-H bond of around $\tau = 10$ fs, and a resonance defeating requirement to keep timesteps at less than $\tau/2$ [46], leads to a requirement for an outside timestep of around 4 fs. With a standard inside timestep of 1 or 2 fs, this only leads to, at most, a two to four times speedup in program runtime.

2.2.2 Long Range Electrostatics. Dynamics simulations, especially those of highly charged biomolecules, require electrostatic contributions of all atoms. Electrostatic potentials can be calculated using basic Particle-Particle Particle-Mesh (alternatively abbreviated PPPM or P3M) methods [19]. Here, the simulation domain is broken into a regular grid, and a single dipole or quadrapole expansion is calculated representing all particles in each box. Interactions are then calculated between each particle and the distant multipole with near-neighbor interactions calculated directly. This yields a significant improvement over full direct methods, with a scaling of $O(N \log N)$ [32].

Periodic boundary conditions (PBCs) help negate edge effects of atoms interacting with the simulation boundary by considering the simulation space to be part of

an infinite domain. Therefore, when particles pass through a boundary, the particle reappears on the opposite side. In general, periodic boundary conditions can be applied in any or all three dimensions. This requires the contribution to the electrostatic potential of an infinite number of images of the system in each periodic direction. The long range nature of the coulomb potential requires a slowly decaying sum over all these periodic images. Ewald summations greatly simplify this computational task by converting the slowly decaying sum to a quickly decaying sum in reciprocal space through a Fourier transform [16]. Particle Mesh Ewald (PME) is an extension of Particle-Mesh with Ewald summations to calculate interactions with periodic images of the system [12]. Because of the Fast Fourier Transformations (FFTs), these methods will generally scale as $O(N^{3/2})$, and can be difficult to parallelize efficiently. In certain circumstances, PME methods will scale as $O(N \log N)$, and are commonly utilized because they are relatively simple to implement, and moderately fast.

An alternative to the P3M methods is a family of algorithms known as "tree methods". Tree methods use a hierarchical structure, starting with a fine grid, condensing the electrostatic information for each small region in the grid, and then summing it with neighbor regions to create a single expression for the larger region. This process is repeated with fewer and fewer grid points at each level, until a single expression is developed for the entire system. Most tree methods, such as the Barnes-Hut mechanism, have never been implemented in molecular dynamics because of their relative complexity compared to PME.

There have been several recent developments in which are termed multi-grid methods [44, 45, 51]. These are similar to tree methods, but are not as difficult to implement. In general, these methods can scale as $O(N)$. Their relative simplicity compared to the Fast Multipole Method can result in faster serial execution time. However, like PME, when implemented on parallel computers, they still rely on a large number of all-to-all communications and require a bit more synchronization between multiple processors. As a result, they are not as adaptable to methods of overlapping communication and computation as the Fast Multipole Method (see Section 3.3.2).

While there have been a number of groups researching these methods, none of them have yet to be implemented in a common MD code.

All methods, including Particle Mesh Ewald and the Fast Multipole Method, suffer from problems relating to conservation of energy. This is a result of the approximation of the force contribution of the distant charges, where we cannot guarantee that the force of particle A on B is not exactly the same as the force of B on A, violating Newton's Third Law. Furthermore, when atoms cross grid boundaries, these methods can lead to discontinuity in the potential. Both problems require these methods to be calculated with high numerical accuracy, reducing their time-saving potential.

Improved tree methods were developed in the mid-1980s, starting with Barnes and Hut [6], and later improved to the Fast Multipole Method developed by Greengard and Rokhlin [18]. These methods were originally thought to provide $O(N \log N)$ performance, but were later shown to provide $O(N)$ scaling [14].

2.3 The Fast Multipole Method

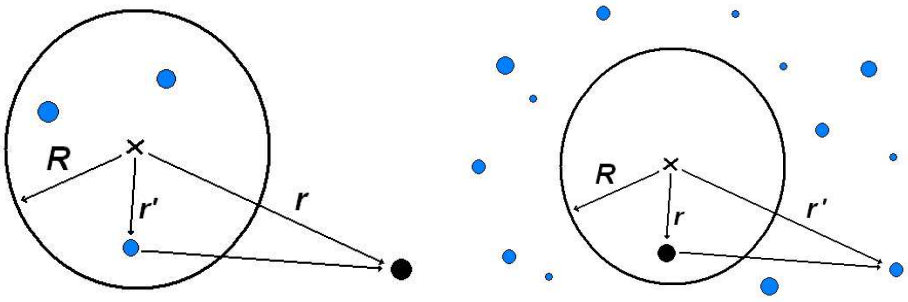
The Fast Multipole Method is an advanced tree method that calculates multipole expansions for small regions of space across the simulation domain, and then transform these multipoles across a tree hierarchy to quickly calculate the potential at any region of space.

2.3.1 Manipulation of Electrostatic Multipoles. The $1/\mathbf{r}$ function, as seen in the electrostatic potential (Equation 2.3) can be expanded about a common origin in three dimensions using spherical harmonics [21] as:

$$\frac{1}{|\mathbf{r}_> - \mathbf{r}_<|} = \sum_{l \geq 0} \sum_{|m| \leq l} \frac{r_<^l}{r_>^{l+1}} \frac{(l-m)!}{(l+m)!} P_l^m(\cos\theta_>) P_l^m(\cos\theta_<) e^{im(\phi_> - \phi_<)}, \tag{2.4}$$

where $\mathbf{r} = \{r, \theta, \phi\}$, $|\mathbf{r}_>| > |\mathbf{r}_<|$, and $P_l^m(u)$ is the associated Legendre function:

$$P_l^m(u) = \frac{(-1)^m}{2^l l!} (1-u^2)^{m/2} \frac{d^{l+m}}{dx^{l+m}} (u^2-1)^l, \tag{2.5}$$



(a) (b)

Figure 2.3: Diagram of a standard multipole (a) and a local expansion (b). The charge distribution is represented with blue points, and the position the potential will be evaluated at is the black point.

where, in the case of our expansion, u is $\cos\theta$. For standard electrostatic multipoles, with a charge at $\mathbf{r}'$ contained somewhere within a sphere of radius R and the potential evaluated at position $\mathbf{r}$, Equation 2.4 can be rewritten as

$$\frac{1}{|\mathbf{r}-\mathbf{r}'|} = \sum_{l\geq 0}\sum_{|m|\leq l} \frac{r'^l}{r^{l+1}} \frac{(l-m)!}{(l+m)!} P_l^m(\cos\theta) P_l^m(\cos\theta') e^{im(\phi-\phi')}, \tag{2.6}$$

provided $|\mathbf{r}| > R \geq |\mathbf{r}'|$. This is referred to as a *Multipole Expansion*, and is demonstrated in Figure 2.3 (a). A very similar expression known as a *Local Expansion* can be written from Equation 2.4 for a distant charge located at $\mathbf{r}'$ outside of a sphere of radius R, and acting on a point $\mathbf{r}$ inside the sphere, (see Figure 2.3 (b)):

$$\frac{1}{|\mathbf{r}-\mathbf{r}'|} = \sum_{l\geq 0}\sum_{|m|\leq l} \frac{r^l}{r'^{l+1}} \frac{(l-m)!}{(l+m)!} P_l^m(\cos\theta') P_l^m(\cos\theta) e^{-im(\phi-\phi')}, \tag{2.7}$$

Some references may loosely refer to all three of the previous expressions as multipole expansions, but here we shall use these more precise definitions.

Defining two new functions $L_{lm}(\mathbf{r})$ and $M_{lm}(\mathbf{r})$, Equation 2.6 can be written simply as

$$\frac{1}{|\mathbf{r}-\mathbf{r}'|} = \sum_{l\geq 0}\sum_{|m|\leq l} M_{lm}(\mathbf{r})L_{lm}(\mathbf{r}') \tag{2.8}$$

where

$$M_{lm}(\mathbf{r}) = \frac{(l-m)!}{r^{l+1}}P_l^m(\cos\theta)e^{im\phi}, \tag{2.9}$$

and

$$L_{lm}(\mathbf{r}) = \frac{r^l}{(l+m)!}P_l^m(\cos\theta)e^{-im\phi}. \tag{2.10}$$

The charge distribution, $\rho(\mathbf{r}')$, can be defined for a series of point charges through the use of a delta function:

$$\rho(\mathbf{r}) = \sum_{i=0}^{n} q_i\delta(\mathbf{r}-\mathbf{r_i}). \tag{2.11}$$

Using this formulation, we can write the full electrostatic potential for a multipole by substituting Equation 2.8 into Equation 2.3:

$$\Phi(\mathbf{r}) = \sum_{l\geq 0}\sum_{|m|\leq l}\int \rho(\mathbf{r}')M_{lm}(\mathbf{r})L_{lm}(\mathbf{r}')d^3r', \tag{2.12}$$

and for a local expansion the potential is:

$$\Phi(\mathbf{r}) = \sum_{l\geq 0}\sum_{|m|\leq l}\int \rho(\mathbf{r}')L_{lm}(\mathbf{r})M_{lm}(\mathbf{r}')d^3r'. \tag{2.13}$$

Note that in Equations 2.12 and 2.13, the charge distribution at all points $\mathbf{r}'$ can be algebraically separated from the location where the potential is being measured, at $\mathbf{r}$, which can be referred to as the observer. As a result, all information about a given charge distribution ρ can be calculated without knowledge of the observer,

$$\tilde{L}_{lm}(\mathbf{r}') = \sum_i q_i L_{lm}(\mathbf{r_i'}), \tag{2.14}$$

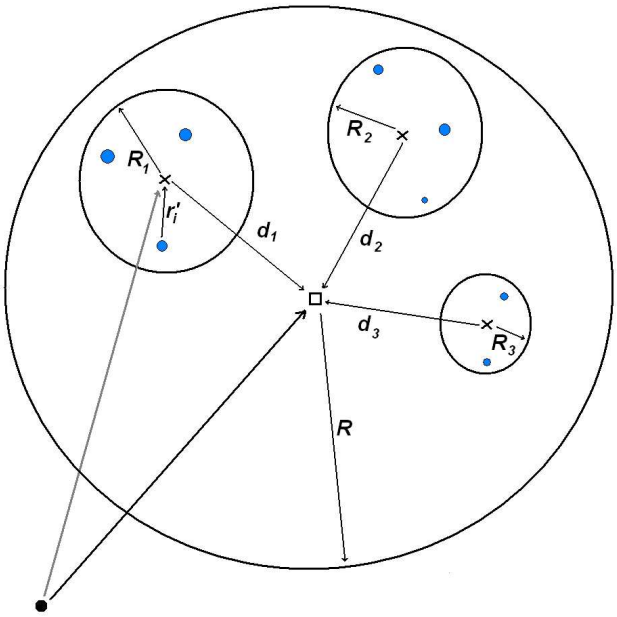


Figure 2.4: Diagram of the shifting of the center of expansion for three different multipoles by $\mathbf{d_1}$, $\mathbf{d_2}$ and $\mathbf{d_3}$. Note that the new definition of $\mathbf{R}$ is such that it encompasses all of the area circumscribed by $\mathbf{R_1}$, $\mathbf{R_2}$ and $\mathbf{R_3}$.

where $\tilde{L}_{lm}$ is referred to as an *unobserved* multipole expansion. The full potential can be calculated at a later time, provided the observer meets the criteria $\mathbf{r} > R$, with the expression

$$\Phi(\mathbf{r}) = \sum_{lm} \tilde{L}_{lm}(\mathbf{r}')M_{lm}(\mathbf{r}). \tag{2.15}$$

We can also similarly define an unobserved local expansion, $\tilde{M}_{lm}$

$$\tilde{M}_{lm} = \sum_{atoms} q_i M_{lm}(\mathbf{r_i'}). \tag{2.16}$$

These expressions are for single-center multipoles, with a common center of expansion, and were defined in terms of $\mathbf{r}$ and $\mathbf{r}'$. If we wish to shift the center of expansion to a new location, we start with the definition of a generalized two-center multipole expansion [10], defined in terms of $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c}$. This can be expressed in

L_{lm}, M_{lm} notation as:

$$\frac{1}{|\mathbf{c}-(\mathbf{a}+\mathbf{b})|} = \sum_{lm}\sum_{l'm'} L_{lm}(\mathbf{a})M_{l+l',m+m'}(\mathbf{c})L_{l'm'}(\mathbf{b}). \tag{2.17}$$

To shift the origin of a multipole by $\mathbf{d}$, rewrite Equation 2.8 as

$$\frac{1}{|\mathbf{r}-\mathbf{d}-(\mathbf{r}'-\mathbf{d})|} = \sum_{lm}\sum_{l'm'} L_{lm}(\mathbf{r}'-\mathbf{d})M_{lm}(\mathbf{r}-\mathbf{d}) \tag{2.18}$$

then substitute $\mathbf{a} \rightarrow \mathbf{r}'$, $\mathbf{b} \rightarrow -\mathbf{d}$, and $\mathbf{c} \rightarrow \mathbf{r}-\mathbf{d}$ and match terms with Equation 2.17 to get

$$L_{lm}(\mathbf{r}'-\mathbf{d}) = \sum_{l'm'} L_{lm}(\mathbf{r}')L_{l-l',m-m'}(-\mathbf{d}). \tag{2.19}$$

This is shown in Figure 2.4 for three multipole expansions. The origin of a local expansion can also be shifted by $\mathbf{d}$ with

$$M_{lm}(\mathbf{r}'-\mathbf{d}) = \sum_{l'm'} M_{l+l',m+m'}(\mathbf{r}')L_{l'm'}(\mathbf{d}). \tag{2.20}$$

Using a similar process, we know that conversions between a multipole expansion and a local expansion about a distant origin at $\mathbf{d}$, as shown in Figure 2.5, can be made by utilizing Equation 2.17 to give us the expression

$$M_{lm}(\mathbf{r}'-\mathbf{d}) = \sum_{l'm'}(-1)^{l'} L_{l'm'}(\mathbf{r}')M_{l+l',m+m'}(-\mathbf{d}). \tag{2.21}$$

Furthermore, Equations 2.11 and 2.12 show that we can express a single multipole expansion for a set of distant charges through simple summation of multiple expansions, provided all expansions are around a common origin and that the sphere defined by the new R entirely encompasses the areas circumscribed by the previous definitions of R.

If the L_{lm} in Equation 2.21 is in fact an unobserved multipole expansion, $\tilde{L}_{lm}$, then the left hand side of this equation would become a unobserved local expansion,

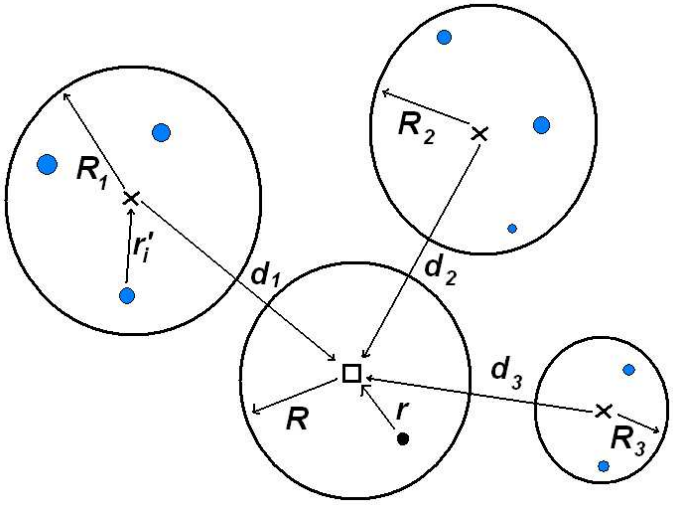


Figure 2.5: Diagram of the transformation of three multipole expansions by $\mathbf{d_1}$, $\mathbf{d_2}$ and $\mathbf{d_3}$, to local expansions about a common origin. Note that the new definition of $\mathbf{R}$ is such that it does not encompass the areas circumscribed by $\mathbf{R_1}$, $\mathbf{R_2}$ and $\mathbf{R_3}$.

$\tilde{M}_{lm}$, as in Equation 2.16. Similar to the expression in Equation 2.15, the potential from the charges can then be calculated at a point $\mathbf{r}$ by simple product with Equation 2.9,

$$\Phi(\mathbf{r}) = \sum_{lm} L_{lm}(\mathbf{r}) \sum_{atoms} \rho(\mathbf{r}') M_{lm}(\mathbf{r}' - \mathbf{d}) d^3 r' \tag{2.22}$$

$$= \sum_{lm} L_{lm}(\mathbf{r}) \tilde{M}_{lm}(\mathbf{r}' - \mathbf{d}), \tag{2.23}$$

provided that $\mathbf{r} < R < \mathbf{r}' \; \forall \; \mathbf{r}'$.

To then calculate the force acting on a charge at $\mathbf{r}$ requires differentiating Φ with respect to $\mathbf{r}$. The $\tilde{M}_{lm}$ term on the right hand side of Equation 2.23 can be arrived

at numerically, and does not involve $\mathbf{r}$. Using the analytic derivative of L_{lm} [43]:

$$\frac{\partial}{\partial x}L_{lm} = \frac{1}{2}\left[L_{l-1,m+1} - L_{l-1,m-1}\right], \tag{2.24}$$

$$\frac{\partial}{\partial y}L_{lm} = \pm\frac{1}{2}\left[L_{l-1,m+1} + L_{l-1,m-1}\right], \tag{2.25}$$

$$\frac{\partial}{\partial z}L_{lm} = L_{l-1,m}, \tag{2.26}$$

the force as a result of the charge distribution can be evaluated anywhere, provided sufficient distance between $\mathbf{r}$ and $\mathbf{r}'$.

2.3.2 The Fast Multipole Method Algorithm. The Fast Multipole Method was first developed by Greengard and Rokhlin in 1987 [18], and can provide $O(N)$ scaling in certain circumstances. It is a variation on tree algorithms that in general provided $O(N \log N)$ scaling. It allows for significant improvement of accuracy at the expense of speed through the parameterization of the number of poles in the expansion (p_m) and the degree of refinement in the tree (l_r).

The method starts with a recursive bisection of simulation space, dividing each dimension in half, descending down one level and repeating this for each of the sub-domains. In three dimensions, this tree has eight children for each parent, and is done recursively and terminated at a level L_{max} when there is approximately one atom in each box. This would imply a L_{max} of $\log_8 N$.

Schematically, this sub-division is represented as an inverted tree. The "top" of the tree is denoted as level zero, and encompasses the entire simulation space. Descending the tree, the first level has eight equally shaped boxes. Each of these has four children in two dimensions, eight in 3D, providing a total of 16 boxes at level two (or 64 in 3D) and so on. For each box at any level, interactions are separated into three regions (see Figure 2.6, drawn with two dimensions). *Short range* interactions encompass particles within the current box, and in the 8 nearest neighbor boxes. In three dimensions, this would yield 26 nearest neighbors. *Intermediate* interactions are defined as boxes that are children of the current box's parent's nearest neighbors,

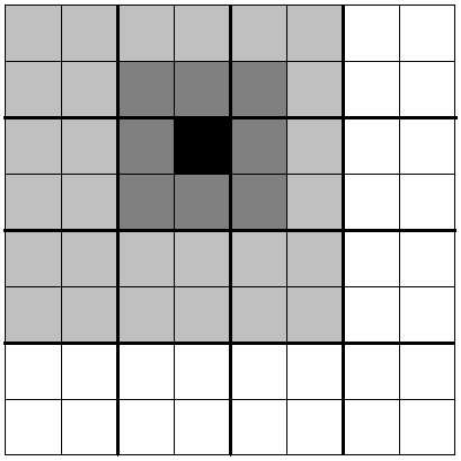


Figure 2.6: Schematic showing interaction zones in two dimensions. The black box is at level three. It has eight nearest neighbors, shaded dark gray; the 27 light gray boxes constitute intermediate interactions; and the distant interactions are white.

but are not its nearest neighbors. There are 27 intermediate boxes in 2D ($6^2 - 3^2$), shaded light gray in Figure 2.6, and 189 ($6^3 - 3^3$) intermediate boxes in three dimensions. Finally, *distant* interactions are all boxes not children of the parent's nearest neighbors.

There are a number of different geometries that are used in the Fast Multipole Method to define short range, intermediate and long range interactions. The implementation presented here uses this tree-based definition, as outlined in Figure 2.6. This is the same as used in Pfalzner [33] and Rapaport [43], as well as in the original work by Greengard and Rokhlin [18]. Other implementations, such as the work by Lupo [28] and Rankin [42] use a geometry based definition that have the advantage of providing a user definable parameter, θ, that allows the scaling of the size of these interaction zones, and the development of different neighbor lists. This geometric methodology is not included here.

The Fast Multipole Method involves the following steps:

1. Build a tree through a recursive bisection of simulation space, sorting all atoms into their respective boxes at the lowest level.
2. Calculate the unobserved multipole expansion ($\tilde{L}_{lm}(\mathbf{r}')$) using Equation 2.14 for all charges in each of the smallest boxes about the center of each box.
3. Shift the center of each $\tilde{L}_{lm}(\mathbf{r}')$ expansion to the center of the parent's box using Equation 2.19. Sum the results for each of the eight children. Repeat this process all the way to the top of the tree.
4. Starting at the second level, for each box, transform each expansion $\tilde{L}_{lm}(\mathbf{r}')$ for all intermediate and distant boxes to an unobserved local expansion, $\tilde{M}_{lm}(\mathbf{r}' - \mathbf{d})$ about the current box (at $\mathbf{d}$) using Equation 2.21. Sum all contributions to form a single unobserved local expansion.
5. Step down the tree, transforming each local expansion $\tilde{M}_{lm}(\mathbf{r}')$ to the center of each of it's children's boxes with Equation 2.20. Add in new intermediate interactions using Equation 2.21.
6. At lowest level, calculate force contribution from the local expansion on each of the individual atoms.
7. Directly calculate contributions for all short range interactions.

The center of each box is used for the origin of each expansion. Note that the multipole expansion in step 2 is merely the unobserved multipole, obtained by evaluating L_{lm} in Equation 2.10, multiplied by the charge distribution, as in Equation 2.14. The transformation in steps 4 and 5 are calculated with Equation 2.21, and are transformed to the center of the new box. The short range interactions in step 7 are calculated directly by evaluating Equation 2.3.

Constructing the tree at the beginning of each timestep requires a relatively small amount of work. The majority of the computation effort required for the Fast Multipole Method occurs in step 5, as it involves the application of Equation 2.21

Table 2.1: Timing For Serial FMM

System Size (Atoms)	$\log_8 N$	Time (sec) $l_r = 0$	Time (sec) $l_r = 1$
4898	4	26.75	2.58
5943	4	26.73	2.47
48451	5	251.88	27.85
202627	5	255.4	38.52

repeatedly for each box. Ignoring boundaries, and in three dimensions, each box, at every level, requires the transformation of 189 intermediate interactions. Note that Equation 2.21 includes a sum over m and l as well as the over m' and l'. This leads to a $O({p_m}^4)$ nature of the transformation, where p_m is the number of poles, or terms in the expansion.

Table 2.1 shows benchmarks of an early serial version of the Fast Multipole Method with 10 terms in the expansion and at various levels of refinement in the tree. This clearly shows that the cost of calculating one additional level is much greater than the cost of the direct calculations in step 7. As a result, improvements in speed are generally realized by utilizing fewer than $\log_8 N$ levels of refinement. Therefore, in this implementation, the number of levels, L_{max}, is determined from the user parameter l_r:

$$L_{max} = \log_8(N) - l_r, \tag{2.27}$$

with a default value for l_r of one. This also leads to a slight improvement in accuracy as more atoms' interactions are calculated directly, and not through the approximation inherent in a truncated multipole expansion.

The Fast Multipole Method can be fairly easily extended to handle Periodic Boundary Conditions [49, 50] by utilizing the multipole expansion calculated at the zeroth and first levels and utilizing the Ewald formulation [16]. This work is not completed as part of this project, but implementation is straightforward and requires little modification to the algorithm as presented here. Some changes are necessary to allow for finer grained interactions of particles near the boundary.

The Fast Multipole Method has the advantages of improved scaling with problem size, easily tunable accuracy through control of the number of terms in the expansion, and improved parallelizability, this comes at the expense of more complicated implementation.

The Fast Multipole Method requires the repeated application of three operators:

1. Shifting the origin of a multipole expansion.
2. Transforming from a multipole expansion to a local expansion.
3. Shifting the origin of a local expansion.

These can be broken down into the evaluation of four mathematical operations:

1. Evaluation of L_{lm}
2. Evaluation of M_{lm}
3. Product of L_{lm} and $L_{l'm'}$
4. Product of L_{lm} and $M_{l'm'}$

The implementation of these four mathematical operators was based on code provided in Rapaport [43], that was adapted into C++ from the C programming language and optimized for performance. All other code provided in this project is either written from scratch by myself, adapted from NAMD, or is directly from NAMD version 2.5 (see Appendix A.1 for overview of structure.)

There are numerous methods that can be utilized to speed up the algorithm. A Fast Fourier Transform acceleration was proposed by Greengard and Rokhlin, and first implemented by Elliott and Board [13]. White and Head-Gordon [56] have also proposed a method to reduce the $O({p_m}^4)$ scaling of the multipole to local transformations down to $O({p_m}^3)$ through a method of rotating coordinate space such that the transformation is done in the $+x$ direction before being rotated back. The initial implementation also uses the straightforward coordinate transformation schema, and future work should look at the advantage of FFT or coordinate rotation improvements.

III. Parallel Implementation

NAMD is a highly robust molecular dynamics code that has proved to be fast and highly scalable. Parallel performance can be measured by defining parallel efficiency as

$$t_{serial}/(t_p * n_p), \tag{3.1}$$

where t_p is the time for each step when run in parallel on n_p CPUs, and t_{serial} is the amount of time per timestep when run on one CPU. NAMD has shown the ability to scale with 90% efficiency when running on 256 processors, and 75% efficiency on 1500 processors [23, 38]. With the hardware trend towards even larger massively parallel systems, more recent work has shown the ability to scale with 90% efficiency to 1024 processors and 70% efficiency to 2048 on Blue Gene/L [24].

NAMD is implemented entirely in C++, and utilizes the Charm++ parallel framework [22]. This allows a message driven paradigm, which is asynchronous by nature. This provides for a maximum of overlap between communication and computation with minimal programmer effort.

3.1 NAMD's Architecture

3.1.1 NAMD's Parallel Structure. NAMD version 2 relies on a hybrid decomposition scheme that utilizes both spatial and force decomposition [35]. The simulation domain is first broken spatially, into cubic *patches*. These patches have a minimum length that is slightly greater than the cutoff distance (see Figure 2.2), but are sized so there are an integer number along each dimension of the simulation space. Atoms are then assigned to a *home patch* based upon their position. The patches are then distributed evenly across all CPUs. After a given number of timesteps, by default twenty, atoms whose position has drifted outside their respective patch are migrated to the appropriate adjacent patch. The patches provide the atom positions to the force computation objects, and handle the velocity updating and time integration for the atoms.

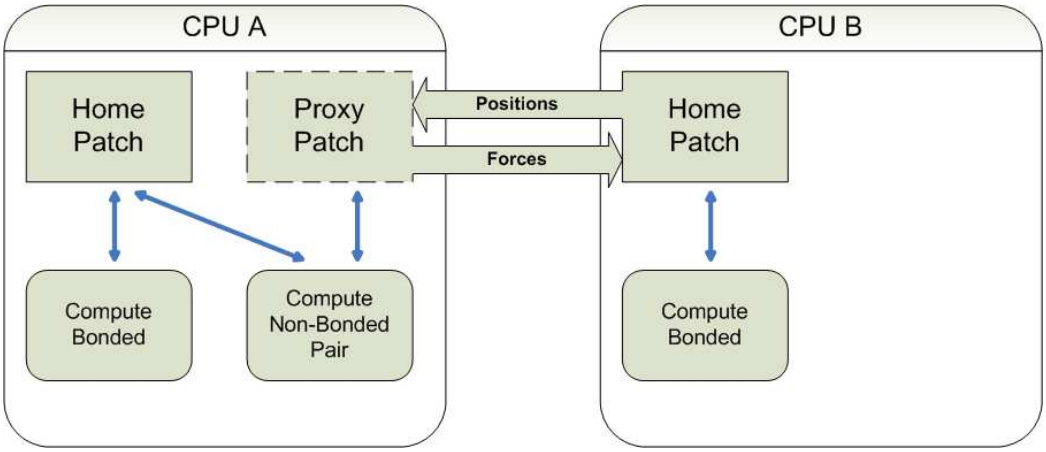


Figure 3.1: Schematic of the hybrid spatial-force decomposition in NAMD2. Home patches contain atom positions and do time integration, while compute objects receive the positions from the patches and return forces. If a compute object resides on another CPU, a proxy patch is created to centralize all communication between that processor and the home patch.

In addition, NAMD2 implements a force decomposition scheme, where each type of force is assigned to a C++ object. One instance of each *force object* is initialized on each CPU. Example force objects include `ComputeBonds`, `ComputeNonbondedSelf`, `ComputeNonbondedPair`, or `ComputePME`. Compute objects may be associated with a single patch (`ComputeBonds` object), or they may require the positions of two adjacent patches (`ComputeNonbondedPair`). If the compute object resides on a different CPU than one or both of the patches, a *proxy patch* is created. The proxy patch handles communication with the home patch and the compute objects on that CPU. It receives atom positions from the home patch, provides them to the compute objects, and then transmits the resulting force vector back (see Figure 3.1).

3.1.2 Dynamic Load Balancing in NAMD. NAMD is unique among MD programs in that it performs dynamic, measurement based, load balancing. All MD programs perform an initial load balancing when first distributing atoms to the processors. This simple, heuristic-based static load balancing has several disadvantages. First, different force types can cause different amounts of computational load. Second, processor speed, and especially inter-processor communication costs, can be very

difficult to estimate. Finally, over the length of an MD run the density may shift, with atoms moving a significant distance across the length of the simulation.

NAMD uses migratable objects, and over the course of the simulation, individual home patches or force objects may be moved based on actual measurements of their run time, and on their communications dependencies [23]. After an initial number of timesteps (by default 100), NAMD will observe the CPU load on each processor, and may move patch or force objects. This decision making process will be repeated frequently throughout the simulation, usually every 2000 timesteps after the initial re-balancing, although these are user definable parameters. The impact of dynamic load balancing can be significant. When comparing the average time per step over the first 100 steps with subsequent steps, it is common to observe a 10-30% performance improvement. This adaptive load balancing also allows for efficient use of inhomogeneous clusters, when some nodes have different clockspeeds from others. The load balancing strategy adapts to them when assigning work to various nodes, allowing faster nodes to carry a heavier load, and not having to spend idle time waiting for slower nodes to finish.

The measurement mechanisms are provided by Charm++. Charm++ also provides a number of load balancing strategies, and these may be specified by the user at runtime [57]. These strategies include Refine, which slowly moves objects from the heaviest loaded processors and RefineComm, which includes communications costs in the measurement of load. The Greedy strategy moves the biggest object from the heaviest loaded processor to the lightest, and a Random strategy will randomly reassign objects. The Metis strategy will partition according to the object communication graphs. There are also a number of Neighbor methods which will only move objects to nearby processors, and can take into account the computer's communications topology, and significant research is underway to tune these methods to various computers such as Blue Gene/L [3]. The work presented here does not take into consideration the impact of these various strategies, and all data presented is with the default load balancing scheme supplied with NAMD.

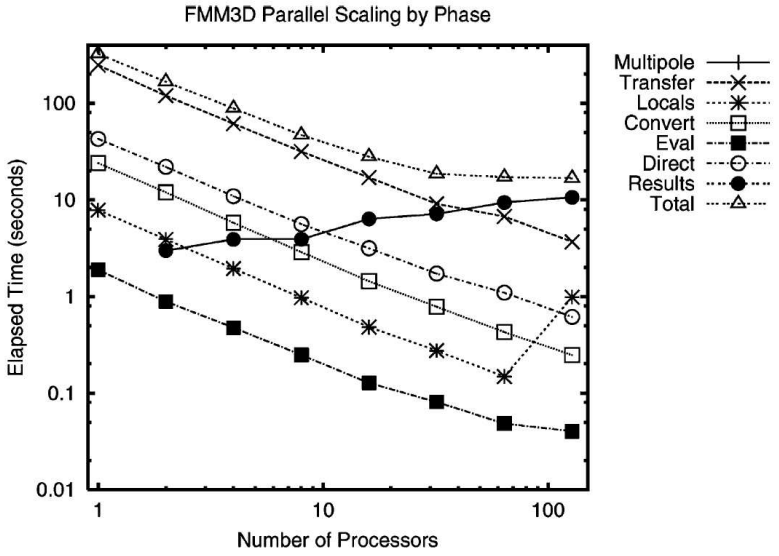


Figure 3.2: Fast Multipole Method scaling, broken down by phase. The algorithm shows very good scaling, but it is dramatically slowed by the communications-heavy results phase, where the force vector is condensed to a single CPU, and then redistributed to the appropriate patches. Figure taken from [28].

3.2 Previous Work on FMM

The initial implementations of the Fast Multipole Method were in astronomical simulations of gravitational attractions between solar systems [33,49]. However, these differ from molecular dynamics simulations because the density of galaxies are highly inhomogeneous, unlike the density of a protein in solution. Much of the earlier work on the Fast Multipole Method in molecular dynamics were proof of concept demonstrations of FMM, and were not integrated into commonly used, full featured MD programs [25,26,43]. When much of this development was done in the early 90s, MD simulations were much smaller, on the order of a couple thousand atoms. This early work suggested that FMM would be more efficient than PME at around 70 thousand particles [49].

There have been two primary efforts to integrate FMM into commonly used code, both of them targeting NAMD. Early versions of NAMD, including version 1 and versions 2.0 through 2.2, included the Distributed Multipole Tree Algorithm

(DPMTA) library [23,42]. This was a general purpose, parallel FMM implementation developed at Duke University and later integrated into NAMD. The second implementation was completed in 2001, and written and integrated directly into NAMD version 1 [28].

Both of these efforts had a common performance limitation. They utilized a parallel decomposition scheme for FMM that was separate from that used by the core NAMD program. The atom positions, distributed across all the processors, had to be consolidated to a single processor, where a single function call to FMM was executed. FMM would then redistribute the atom positions according to its own methodology, compute the forces in a distributed manner, reconsolidate the forces and return to NAMD, which would redistribute the forces back to their respective patches. This mechanism was communication heavy, and did not parallelize well to a large number of processors. Figure 3.2, taken from [28], shows this problem. The Fast Multipole Method algorithm parallelizes well, taking up significantly less wall clock time at larger number of processors, but the results portion of the implementation, the redistribution of the force vector, quickly becomes dominant, negating the parallel advantage.

3.3 Design of Parallel Algorithm

3.3.1 Domain Decomposition. In order to overcome the primary performance limitation of the two previous Fast Multipole Method implementations in NAMD, the primary goal of this project is to eliminate the communication of atom positions and the resultant force vectors by preserving the hybrid spatial-force decomposition scheme. Instead of building the FMM tree from the top down through a standard recursive bisection, this implementation starts at an intermediate level of the tree, defined as the *split level* (L_{split}), and sizes the FMM boxes at this level so that they coincide perfectly with the preexisting NAMD patches (see Figure 3.3). From this point, a recursive bisection is done to complete the bottom of the tree by subdividing the patches. The top portion of the tree is then built by working back up

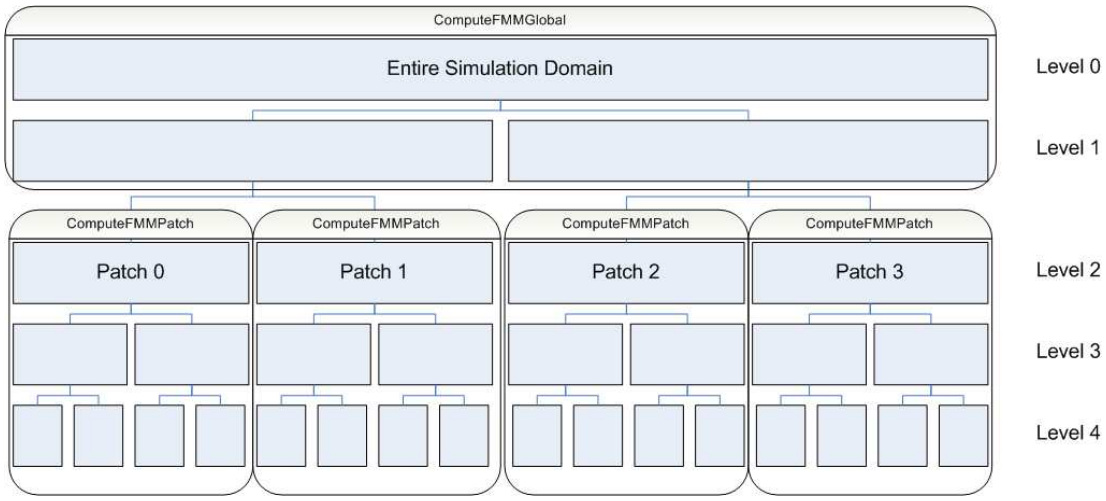


Figure 3.3: Schematic of domain decomposition and the Global and Patch schema for the Fast Multipole Method compute objects. Schema shows one dimensional representation of a tree with four levels and the split level set to 2.

from the split level. This allows execution to begin without the transmission of any atom positions, and complete without communicating force vectors. It also preserves the dynamic load balancing scheme with little additional work.

The current implementation follows the existing NAMD paradigm, and closely mimics the current `ComputePME` implementation in structure. A single compute object, `ComputeFmmObj`, is started on each processor, and execution begins with a call to `doWork()` in this class. The `ComputeFmmObj` object creates an array of `ComputeFmmPatch` objects, one for each patch that processor is responsible for, and hands off the atom positions and results force vector to each of these. The `ComputeFmmMgr` class handles all the interprocessor communications, as well as all communication between the `ComputeFmmPatch` objects. The final C++ object, `ComputeFmmGlobal`, handles the tree above the split level (see Figure 3.3); only one instance of this object is necessary for the entire simulation. A `ComputeFmm` class implements a serial version of the algorithm, and provides the bulk of the expansion manipulation functions. This class is inherited by both `ComputeFmmPatch` and `ComputeFmmGlobal`. (See Appendix A for more detail.)

3.3.2 Overlapping of Communication and Computation. For most parallel applications, the time required to communicate between processors is the primary driver of inefficiency. In instances when communications cannot be limited, the latency is best hidden by conducting other computational work while waiting for messages to arrive. As a result, information that is needed by other processors should be transmitted as soon as the information is known, and the steps necessary to compute data that will be needed on other processors should be given priority. Similarly, tasks that can be done with available data should be delayed until all data with parallel dependencies has been calculated and transmitted.

In the Fast Multipole Method, there are three points necessitating communication. One is the transmission of the expansions up the tree to the global object, and the local expansions back down (see Section 2.3.2, steps 3 and 5). Second, the expansions need to be transmitted to the nearby boxes on the interaction list (step 4). Finally, atom positions in the smallest level boxes on the edge of the patches need to be transmitted to their neighbors for the full direct portion of the calculation (step 7).

To overlap the communication and computation as much as possible, the steps as listed in Section 2.3.2 are handled in the following order. After the building of the tree, the atom positions near the edges of the patches that will be needed for the nearest neighbor full direct calculations should be determined and transmitted. Steps 2 and 3 are then completed, with the resulting expansion representing the entire patch transmitted up to the global compute object. The expansions for the boxes below the split level are transmitted to the nearest neighbor patches for use in step 4. Because the communications with the global object require two sequential transmissions, the expansions up, and then the resulting unobserved local expansions back down, these messages should be prioritized over the messages to the neighbor patches. Once all this has been transmitted, if no messages have arrived from neighbor patches, execution can skip to step 7, and the forces of neighbor atoms can be calculated where possible. The additional interactions for those on the edge of the patch, where the nearest

neighbor boxes reside on another processor can be added in as soon as those messages arrive.

The transformations of the unobserved multipole expansions from intermediate neighbors to unobserved local expansions can be calculated and summed independently from the transformation of the parent's local expansion down. As a result, all the messages that arrive can be acted on immediately, without waiting for the global object to complete its work. A counter is kept for each patch, and once all messages have arrived, both from neighbor patches as well as the local expansion from the global portion of the calculation, execution can continue the rest of the way down the tree (steps 5 and 6).

Completion of the full direct portion of the calculation is notably independent of the rest of the algorithm, and can be completed earlier, overlapping the expansion's transmission, and hiding much of the communications cost. Furthermore, chances are likely that the relevant atoms positions and charges from the neighboring patches have already been transmitted by NAMD to the local processor, to aid in the computation of the Van der Wahl's or the bonded force contributions. These atom positions are stored either in the home patch, should it be resident on the same processor, or in a proxy patch. For the sake of simplicity of programming effort, this feature is not utilized in the current implementation of FMM, as the relevant atom positions are transmitted by FMM itself. This remains as a significant future optimization, with the benefit of removing a large amount of redundant communication.

In the global compute object, the first transformation up can be completed as each message arrives. Once all expansions have been received, execution will continue to the top of the tree, and then back down to the split level (steps 3 through 5). For a three dimensional system, there will be a total of 189 boxes in the intermediate interaction zone for any given box, assuming it's not near the simulation boundary (see Figure 2.6). For boxes just below the split level, this is a significant amount of communication, as the expansions for all boxes that are two away from a patch boundary

have to be transmitted to the neighboring patches. Furthermore, the expansion for an entire patch has to be sent to 189 other patches. In addition, all expansions one below the split level have to be transmitted to the 124 neighboring patches. In order to prevent this massive amount of communication, all of this computation will be completed in the global object. For all levels below this, the expansions for the edge boxes only have to be transmitted to the 26 nearest neighbor patches. The current implementation, in order to reduce complexity, transmits one identical message to all nearest neighbors containing the unobserved multipole expansions. In an attempt to reduce communication size, customized messages should be sent to each processor containing only the relevant expansions. This remains a possible future optimization that has not been implemented.

3.3.3 Tree Representation. Every compute patch object, and the global compute object contain a copy of the entire tree, stored as a simple array. Every node in the tree contains pointers to each of the box's 8 children, 26 nearest neighbors, and parent. Memory for each box's unobserved multipole and local expansions are allocated when needed, and deleted at the end of every step. Because each patch compute object has its own copy of the tree, when two adjacent patches are on the same processor, the same information is stored in two places. Furthermore, explicit communication in the form of multiple memory copies occurs unnecessarily as the atom positions and expansions are transferred back and forth. This was done to ease implementation. In future work, it would be prudent to limit the architecture to have only one compute object per processor (as the rest of NAMD does for other force types) instead of dedicating one object per patch.

Numerous papers have discussed various hash methods for representing the tree in these multipole methods [55]. In this effort, these complicated schemes are not necessary because memory is more abundant in modern computers, and in execution time, tree traversal is trivial compared to the multipole transformations [33]. Table 3.1 shows the total execution time for each step in the initial, serial version of the

Table 3.1: Performance profile for a serial version of FMM, showing time spent in each of the major steps.

Step	Function	Time (sec)	Time (%)
1	`BuildTree() & PopulateTree()`	0.03	0.1
2	`CalculateMP()`	0.10	0.4
3	`TransformMPUpTree()`	0.28	1.2
4 & 5	`IntermediateLE()`	19.61	82.7
5	`TransformLEDown()`	0.13	0.6
6	`MPToForces()`	0.37	1.6
7	`NearInteractions()`	1.84	7.8

Fast Multipole Method, calculating the forces for a 48 thousand atom system of GFP. The number of terms (p_m) is 7, and l_r is one. This shows that very little time is spent setting up the tree; instead, the vast majority of time is spent manipulating the various expansions. Table 3.2 breaks this down further, as the `evalL()`, `evalM()`, `prodLL()`, `prodLM()`, and `IntermediateLE()` functions involve mathematical manipulations of expansions. The `NearInteractions()` function is the full direct portion in step 7. The remainder, totaling less than a half of a percent of the execution time, can be classified as building and traversing the tree.

A significant driver of the parallel performance of the algorithm as outlined is the location of the split level (L_{split}) in relation to the total number of levels (L_{max}). In general, using a cutoff distance of between 9 and 14 Å, there are approximately 1000 atoms in each patch. This means there generally are three levels of the tree below the split level, and only two if l_r is set to one. For many common problems, the patch grid is around four to eight patches per side. In the case of GFP, with a patch grid of 4 x 4 x 4, this means $L_{split} = 2$. At most, for current problem sizes, the split level is 4. Note, however, that the Fast Multipole Method without periodic boundary conditions only requires manipulations for expansions at level two and below. This means that for common problem sizes, such as GFP, there is very little work for the global compute object to handle.

Table 3.2: Performance profile for serial FMM, showing time spent in each function.

Function	Time (sec)	Time (%)
`prodLM()`	13.59	59.95
`NearInteractions()`	6.54	28.85
`IntermediateLE()`	0.51	2.25
`evalM()`	0.49	2.16
`prodLL()`	0.12	0.53
`evalL()`	0.11	0.49
`MPToForces()`	0.10	0.44
`CalculateMP()`	0.02	0.09
`PopulateTree()`	0.02	0.09
`ComputeForces()`	0.01	0.04
`BuildTree()`	0.01	0.04

For a system of 570 thousand atoms, the patch grid is around 10 x 10 x 10 yielding a split level of 4. This means that the global patch has to do expansion transformations over two levels of the tree, giving it a workload approximately equal to each of the patch compute objects. There is a limit to how well this scheme parallelizes for extremely large systems. While much of the additional work from more atoms is distributed across the compute patch objects, it simultaneously results in more, non-parallelized, work in the global object. In order to extend this methodology to systems with tens of millions of atoms, the global object would itself need to be parallelized, further distributing this load.

Most of the current implementation is transparent to the load balancing system, and allows NAMD to shift patches to different processors to re-balance the work. The exception to this is the global object. The current implementation starts the global compute object on a single processor, and does not move it. Furthermore, the load balancing system is not notified of the additional workload represented by the global object. Instead, because NAMD uses a measurement based load balancing scheme, I rely on it to recognize the additional time taken by that processor, and to migrate other computational objects off.

3.3.4 Current Limitations. The chosen methodology of enforcing that the size of the FMM boxes at the split level coincide with the pre-existing NAMD patches causes further complications when the patch grid is not perfectly cubic. The patch grid can have any number of patches along each side, while the FMM tree will always have 2^l boxes along each dimension. This is handled by fitting the patch grid inside the FMM tree, and then simply allocating no atoms to the additional FMM boxes, so that the area covered by the FMM tree is larger than the simulation size.

This approach has two repercussions. First, it makes future implementation of periodic boundary conditions more difficult. Second, while significant time can be saved by not calculating the bottom levels of the tree for these "null" patches, their contribution to their parent is not zero. Specifically, if the value of a unobserved multipole expansion is zero, and the origin of that expansion is transformed to a new origin using Equation 2.19, the transformed expansion is not zero. For example, if an expansion of zero is transformed by a distance $\mathbf{d} = (2, 1, 1)$, the result is

(re,im)	$m = 0$	$m = 1$	$m = 2$
$l = 0$	(0.00,0.00)		
$l = 1$	(0.00,0.00)	(0.00,0.50)	
$l = 2$	(0.00,0.00)	(0.00,1.00)	(0.00,0.6875)

These contributions in the imaginary part of the $m \neq 0$ elements of the expansion are canceled out, though, if eight non-overlapping children are all transformed the same distance to a common parent. As a result, for all the "null" patches, they have to have an explicitly zero expansion that is transformed to the common parent.

To implement periodic boundary conditions using the Fast Multipole Method, only a few minor adjustments need to be made: the transformation of the expansions up the tree to the zeroth level, the application of the Ewald summation technique [16] for distant images of itself, and propagation of this contribution back down the tree. Furthermore, for patches at or near the boundary, their nearest neighbor lists need to be modified to include the periodic neighbors on the other side of the simulation

domain. Because of the existence of the "null" patches outlined in section 3.3.3, this is non-trivial.

As stated earlier, the current implementation of FMM explicitly communicates the atom positions for all atoms near the edge of each patch to that patch's neighbors. This results in a significant amount of redundant communication at each timestep, as that data is already available on that processor. Section 5.1 has an analysis of the amount of communication occurring at each timestep, and how much could be saved by accessing the proxy patches.

NAMD has a number of features that are used to attempt to reduce the amount of communication between processors. As dynamics progress, some atoms drift across patch boundaries. Atoms are not reassigned to a new patch at every timestep, but once per cycle. This default value is every 20 steps. If an atom migrates too far over a patch boundary before being reassigned, this can cause some drastic errors in the electrostatic force calculation because it is assigned to the incorrect FMM box. In an effort to mitigate these problems, the NAMD configuration parameters `stepspercycle` was set to 10, and `pairlistdist` was set to the cutoff distance. Both of these parameter changes cause slight performance degradation, but the magnitude of this penalty was not measured.

The implementation of the Fast Multipole Method preserves the current NAMD mechanisms for multiple time stepping, although there has not been any robust testing to ensure the accuracy of this feature. As a result, FMM was not run using this MTS feature.

IV. FMM Algorithm Performance

The Fast Multipole Method has three potential advantages over existing methods, it can be more accurate, handle larger systems better or faster, and it has the potential to parallelize more efficiently. The performance of the Fast Multipole Method, in terms of accuracy and problem size scalability are examined in this chapter. The parallel performance of the current implementation is examined in Chapter V.

An important part of the measurement of accuracy of the Fast Multipole Method is the determination of the number of terms in the expansion necessary to model the relevant systems, which is covered in Sections 4.1 and 4.2. The accuracy of the FMM is measured against the full direct computation of the total system energy and the force vectors. This can be directly compared to existing PME methods, and is addressed in Section 4.3.1. To measure the accuracy in dynamic systems, two sample biological systems, ApoA1 and the Green Fluorescent Protein are modeled with both FMM and PME, with the resulting stability and final structures compared with each other, as well as with the accepted crystal structures in Section 4.3.2. Finally, the scaling of the Fast Multipole Method with increasing problem size is presented in Section 4.4.

The current Fast Multipole Method implementation is not as robust as the default PME method included in NAMD, as outlined in Section 3.3.4. There are a number of features, specifically multiple time stepping and dynamic load balancing optimizations, that are available to PME, but not yet throughly tested with FMM. In addition, there are a number of parameters that were modified in order to get FMM running consistently. Specifically, the pair list distance and the steps per cycle parameters were reduced. For all the performance and accuracy tests presented in the following two chapters, the parameters for each method are the parameters that will provide optimum performance for each respective algorithm. The parameters that differed between the two methodologies are listed in Table 4.1, all of which should give additional speed to PME. These benefits in serial speed can also have side effects with better parallel scaling and scaling of problem size as they reduce the workload of the more computationally expensive electrostatic calculations, and move it to the

Table 4.1: Parameter Settings for PME and FMM

Parameter	FMM	PME
`cutoff`	14 Å	14 Å
`pairlistdist`	14 Å	15 Å
`timestep`	1 fs	1 fs
`rigidBonds`	no	no
`nonbondedFreq`	1	1
`fullElectFrequency`	1	2
`stepsPerCycle`	10	20

other MD force calculations and the molecular dynamics engine, which generally scale better than the electrostatics portion.

4.1 Accuracy of the Fast Multipole Method

4.1.1 Theoretical Accuracy. The theoretical accuracy of the Fast Multipole Method was derived in the early papers on the method by Greengard, Rokhlin and Ambrosiano [5, 18]. The number of terms required for a given level of accuracy, ϵ, depends on the number of levels in the tree, and is given as

$$\langle N_{L_{max}} \rangle^{-1} \; 2^{-p_m} \leq \epsilon \tag{4.1}$$

Where $\langle N_{L_{max}} \rangle$ is the average number of atoms in each box at level L_{max}. Following from Equation 2.27, this number is one when l_r is zero, and is eight when l_r is one, meaning that the number of terms needed for an accuracy of ϵ simplifies to

$$p_m = \log_2(8 * \epsilon). \tag{4.2}$$

Test runs with several systems have shown that the current PME implementation has an accuracy of about 10^{-4} (see Table 4.2). Using this as a baseline, assuming a value of l_r of zero, and plugging into Equation 4.1, we find

$$p_m = \log_2(10^{-4}) = 13.3, \tag{4.3}$$

showing that a calculation should use around 13 poles in the expansion. But, given a l_r of one, this requirement drops to 10 terms. Note that this is a worst case error, and some empirical measurements have shown the number of terms required for that level of accuracy to be lower (see Sections 4.1.2 and 4.3).

4.1.2 Empirical Tests For Accuracy. The true test of the accuracy of the implementation is the ability to model real systems in a manner that can validated by experiment, and conforms to well known thermodynamic laws. A classic test of the latter is to measure energy drift in a constant energy, constant volume simulation, to see if it is conserved. A second major test is to measure the protein stability for a number of sample systems. Starting with a known crystal structure, the system can be minimized, heated and equilibrated. A simulation of about 1 ns is considered sufficient to show stability, and provide decent statistics of the structure conformation [48]. The secondary structure can be compared to the known or accepted conformations for that protein.

Energy conservation tests are difficult to perform without periodic boundary conditions, which are not included in the current implementation. While a constant energy simulation can easily be run, individual molecules will drift significantly from the primary system, causing an inability to equilibrate the model, as well as errors arising from atoms drifting outside of the simulation domain. Artificial boundary conditions can be imposed, but soft wall potentials add energy to the system, and there is not currently an efficient method to implement hard wall boundary conditions in NAMD.

There have been a number of previous efforts to implement the Fast Multipole Method in molecular dynamics, and there are well established error bounds for the method, although there is not consensus on the number of terms required in the FMM expansion. The work by Lupo, *et. al.* uses five terms [28]; Kurzak and Petitt use 16; and an early paper by Board, *et. al.* shows relative error with 4 and 8 terms [9]. Only one paper, by Bishop, Skeel and Schulten, has attempted to measure the number of

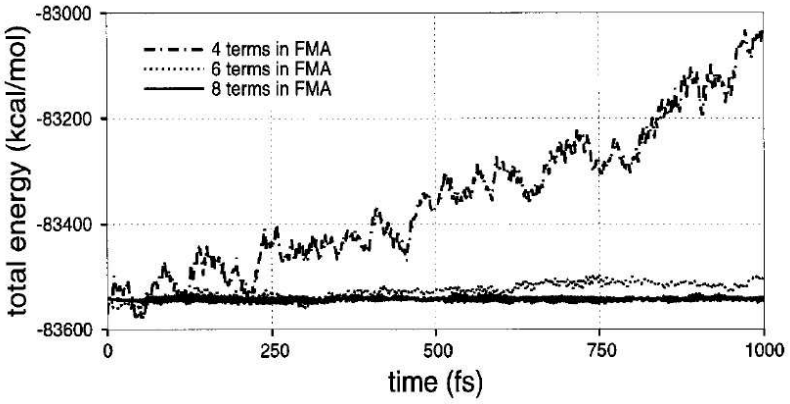


Figure 4.1: This graph shows the conservation of energy with Fast Multipole Method with various number of terms in the expansion. This figure reproduced from [8].

terms required to accurately carry out molecular dynamics simulations with the Fast Multipole Method [8]. This paper includes the results shown in Figure 4.1, which seem to indicate that six terms are sufficient for their 36k atom protein system. It is important to note that the Fast Multipole Method tested in that paper is not a complete analogue to the implementation documented here. The DPMTA algorithm uses a geometric, spherical definition of intermediate and well separated boxes that is different from the tree-based definition used here, and outlined in Figure 2.6. As a result, the measurements provided there for accuracy may not be sufficient to be applied to this work. The results included here all truncate the expansion at 7 terms, except when noted.

4.2 Validation of the Fast Multipole Method

Development started with a serial, standalone version of FMM, that read in a simple structure from a Protein Database (PDB) file. Significant portions of NAMD were used early in the development process in order to maintain consistent data structures and methodology with the target program. A simple, serial version of the Fast Multipole Method was completed, with the resulting energies and force vectors compared between the two for a large number of structures.

This program was then parallelized, using much of the NAMD architecture and structure, including the patch system for decomposition. This program read in the atom positions and calculated a single electrostatic force vector. Both the full direct and Fast Multipole Methods were parallelized and the full direct method was used for comparison purposes.

Because the electrostatic potential decays as $1/r$, most of the energy, and the resulting forces acting on any give atom, are a result of the nearest neighbor charges; with less than 5% of the total energy being contributed by atoms outside of a standard cutoff of 9 Å. As a result, a majority of the force is not actually calculated through the multipole expansions, but are instead contributed through the direct calculation outlined in Section 2.3.2 step 7. In order to differentiate between this portion of the calculation and contributions from expansion transformations, a test utilizing a single water molecule was used. In this test, the tree geometry was fixed with a large number of levels, and the location of the two hydrogens were slowly moved, with the electrostatic forces calculated for each position. Forces could also be calculated with the full direct method, as well as calculating the FMM by hand. In this manner, the parallel implementation could be methodically tested, with the each section of the code verified in isolation.

4.3 Accuracy Measurements

Performance results presented here were run on two parallel systems. The first is Eagle, at the ASC MSRC, a SGI Altix 3700 with 1.6 GHz Intel Itanium2 processors on four 512 processor nodes and 1 GB memory per CPU. All runs on this system were SMP, shared memory jobs. A cluster, Orb, was used for distributed-memory performance measurement. This system has eight dual processor 900 MHz Itanium2 nodes with 2 GB of RAM each. The systems are networked using Myrinet-2000 interconnects directly connected to a single crossbar switch. Serial runs were conducted on various Intel Pentium 4 desktop computers. All systems used Intel compilers, and NAMD was compiled using Charm++ version 5.9.

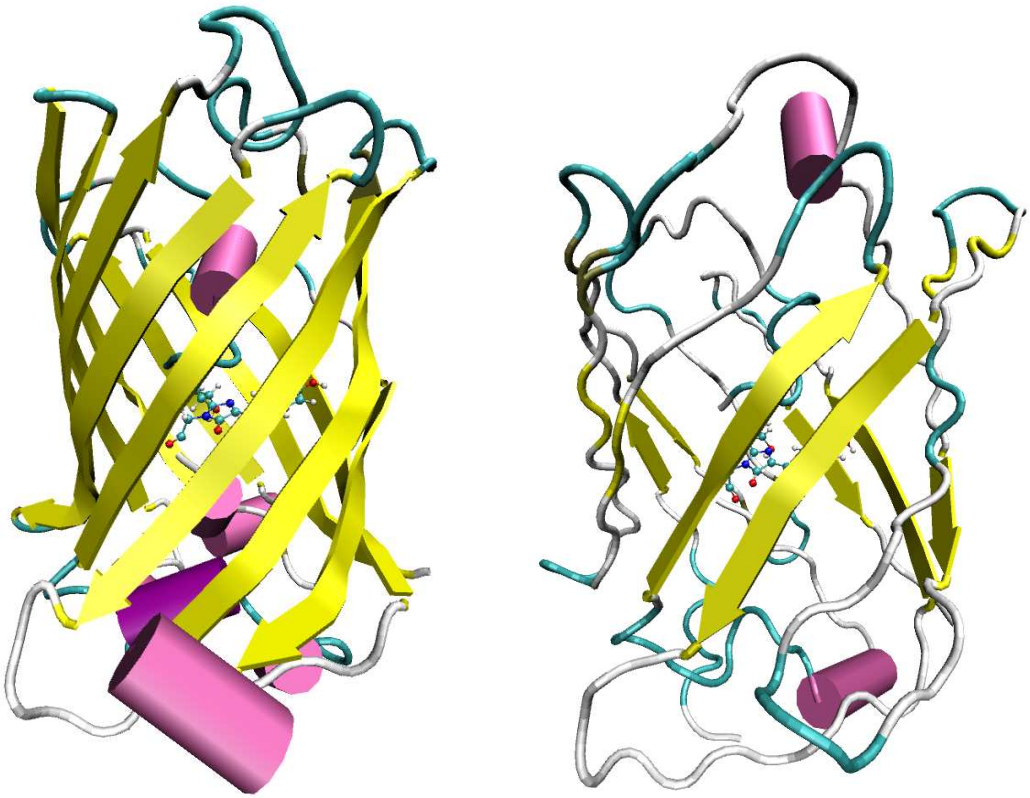
Table 4.2: Single Step Energy Accuracy

	Full Direct	FMM, 7 terms		PME	
		Energy	Error, ϵ	Energy	Error, ϵ
GFP (48k atoms)	-3184692	-3184759	.002%	-3184294	.012%
Water Box (98k)	-347765	-347782	.005%	-347558	.060%
ApoA1 (92k)	-334444	-334411	.009%	-362073	7.6%
Water Box (214k)	-780970	-781312	.044%	-798265	2.2%

4.3.1 Single Step Energy Calculation. The initial verification of the Fast Multipole Method is to calculate the electrostatic forces and the total electrostatic energy for a common structure with various methods. The results are shown in Table 4.2. The sample systems are the Green Fluorescent Protein and Apolipoprotein systems as outlined in Chapter I. In addition, multiple water boxes were created with VMD's `solvate` command, and left at the default density.

The total system electrostatic energy when calculated with the Fast Multipole Method is the same for both the serial standalone version and for the final implementation integrated into NAMD. The full direct numbers are also calculated both in my serial program and the NAMD version, and match exactly. These results are also compared with the standard PME method, both the FMM and PME calculations are compared to full direct to determine relative error. Two conclusions can be taken from this data. First, the Fast Multipole Method, when calculated with seven terms in the expansion, has an advantage in error by about an of magnitude over PME. Secondly, both methods seem to show a slow increase in error as the system size increases, although this rate of increase is significantly faster for PME, which sees the error increase by two orders of magnitude when the system size increases by one order of magnitude. FMM, on the other hand sees the error increase only a single order of magnitude over the same range.

In addition, tests were done with the GFP sample system as the number of terms in the FMM expansion was increased. Interestingly, the level of error shown in Table 4.2 was achieved with as few as three terms in the expansion. As the number



(a) (b)

Figure 4.2: The Green Fluorescent Protein crystal structure (a) and the conformation of GFP after it has been minimized, heated and allowed to equilibrate for 100 ps using FMM (b).

of terms was increased further, the energy remained consistent, within the number of significant digits presented in the table.

4.3.2 Protein Structure Conformation. Runs were done with both FMM and PME on the two sample systems, the Green Fluorescent Protein and ApoA1. All of the runs were done using the parameters listed in Table 4.1, with the PME models using periodic boundary conditions on roughly cubic systems, and FMM using spherical harmonic boundary conditions. All systems began from the crystal structures, were minimized, slowly heated and allowed to equilibrate. The PME systems were run for 2 ns, with the data taken over the second nanosecond. Due to downtime at the ASC

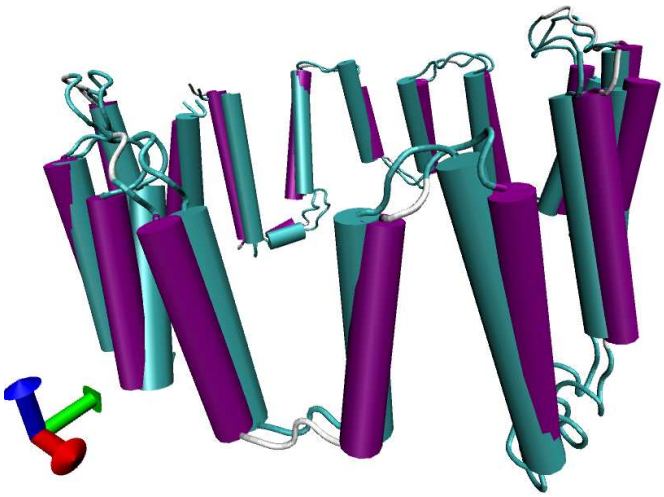
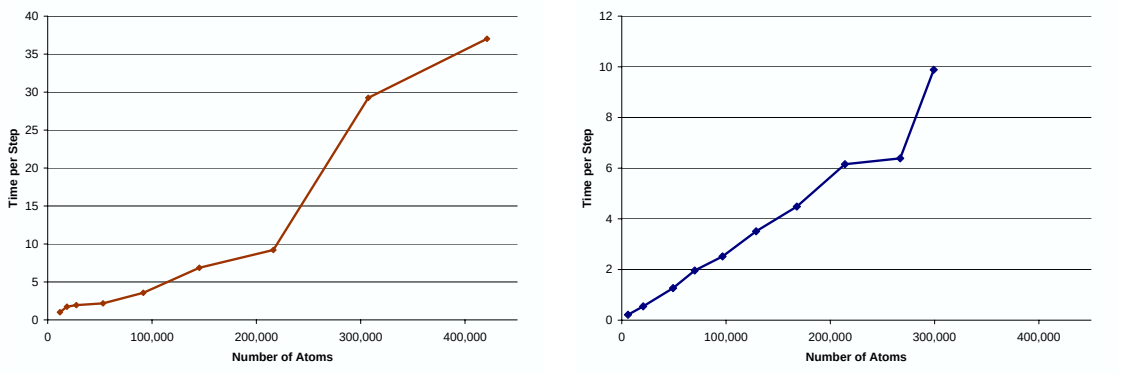


Figure 4.3: The Apolipoprotein A-I protein crystal structure (in violet) and its conformation after it has been minimized, heated and allowed to equilibrate for 100 ps using FMM (in green).

MSRC, less than 100 ps of simulation time has been completed for the FMM runs. Full one nanosecond runs are underway, and the data presented here will be updated when the runs complete.

For both systems, the root mean square deviation (RMSD) was computed for the backbone atoms on the protein only, using the average position for reference. For GFP, when calculated with the Fast Multipole Method, the RMSD was 0.479 Å, and 2.24 Å when run with PME. For ApoA1, the RMSD was 0.705 Å for FMM and 1.68 Å for PME. The fact that the RMSD is significantly lower for the FMM simulations should not be taken as evidence that FMM is more accurate. It is probably entirely a result of the very short simulation time that was completed at this time, and these numbers will probably increase.

The final structure for GFP is shown in Figure 4.2 (b), with the crystal structure in (a). As can be seen here, the overall structure is similar, but some of the secondary structure has been lost. The β- barrel is noticeably larger for the simulated system, but this is due to the heating of the system– the crystal structure was taken at lower temperature. The 100 ps length of the run is too short to conclude if the



(a) (b)

Figure 4.4: These graphs show the time, in seconds, required as the number of atoms increase, for both the Fast Multipole Method (a) and Particle Mesh Ewald (b). Note the stair stepping effect in FMM, as the first three data points have three levels in the tree, the next four have four levels, and the last two points have five.

structure shown here is completely stable. The total system energy had just started to equilibrate, and while the RMSD over time was increasing, it was beginning to level off. The α-helices of the ApoA1 protein are remarkably more stable, as shown in Figure 4.3. In this figure, the final structure, in green, is superimposed over the initial crystal structure, in violet. It is not surprising that this protein held its secondary structure better, as α-helices are known to be far more stable than other structures [53].

4.4 Scaling of Problem Size

Early characterizations of the Fast Multipole Method noted that the linear scaling was offset by a very large constant calculation costs due to the manipulation of the expansions. The PME methods, easier to implement and to calculate, were hampered by their $O(N \log N)$ scaling. Early work showed that the particle mesh methods were faster for the system sizes of the time, and the literature varies on predictions on when the Fast Multipole Method will outperform PME. Schmidt and Lee, working on astrophysical simulations with multipole and particle mesh methods,

Table 4.3: FMM Scaling with problem size.

Num Atoms	Time per Step (s)	Scale Factor
11717	1.009	8.617E-05
18473	1.739	9.414E-05
27494	1.946	7.080E-05
53054	2.181	4.112E-05
91619	3.573	3.900E-05
145325	6.864	4.723E-05
216266	9.209	4.258E-05
307244	29.24	9.518E-05
421025	37.02	8.794E-05

showed a crossover point of about seventy thousand particles, after which FMM is faster [49]. Other work by Pollock and Glosli, also not specific to molecular dynamics, suggests that "FMM only becomes faster at some unphysical size $N > 10^{60}$."

Scaling of the Fast Multipole Method is shown in Figure 4.4 (a), these numbers were calculated using 8 processors on a SGI Altix system. This data was collected by running 200 MD steps in NAMD for a variety of water boxes. The per step timing as reported by NAMD for steps 150 to 200 are given. This methodology is used for all NAMD timing numbers in all tables and figures in Chapters IV and V. These results tend to support the conclusions of Pollock and Glosli. For all systems sizes and computer types, the PME method included in NAMD was faster than FMM with performance ranging from a factor of two to almost a full order of magnitude.

According to Phillips, *et. al.* [34], NAMD uses the smooth PME [15] methodology. In this, although PME is proportional to $N \log N$, parameters are chosen to put the work load into the reciprocal sum which is proportional to N only. Figure 4.4 (b) bears out that this approach is quite effective. The scaling of PME is close to linear, with the one data point at 267k atoms varying from linear only due to normal variations in overall computer load and load imbalance. Because the PME runs were conducted with MTS, the rest of the MD calculation, which scales as $O(N)$, takes up

Table 4.4: PME Scaling with problem size.

Num Atoms	Time per Step (s)	Scale Factor
5945	0.214	3.602E-05
20537	0.546	2.663E-05
49160	1.265	2.574E-05
69956	1.959	2.801E-05
96605	2.515	2.603E-05
128834	3.508	2.723E-05
167879	4.484	2.671E-05
214016	6.155	2.876E-05
266972	6.386	2.392E-05
299152	9.883	3.303E-05
570350	8.671	1.520E-05
783200	12.38	1.581E-05
1042208	16.10	1.545E-05

a more computational time relative to the electrostatic portion. This also helps to reduce the effect of poorer scaling on the part of PME.

The results for FMM in Figure 4.4 (a), show close to linear scaling, although there is a significant stair-stepping effect as a result of the number of levels in the tree, that was also seen in Table 2.1. The first three data points, with less than 30k atoms, are calculated with three levels in the tree. The next four, between 30k and 250k atoms have four levels in the tree, with the final two data point using five levels. Within these subsections, scaling is highly linear, with sudden jumps at each point where an additional level of multipole expansion are required. A scaling factor can be easily defined as the time per step divided by the number of atoms, and the results for PME and FMM are shown in Tables 4.4 and 4.3.

The high computational cost of Fast Multipole Method provides a significant barrier to performance, and the modern implementations of PME have been able to successfully flatten the $O(N \log N)$ scaling to provide good performance for current system sizes. The ability to provide better performance with FMM rests on implementation optimizations and on potentially better parallel performance.

V. Parallel Performance

In recent decades, high performance computing efforts have concentrated on very large parallel supercomputers to achieve the computational efficiency needed to provide the level of detail and accuracy needed for scientific research. This emphasis has become even more dramatic recently, as performance improvements have shifted from emphasizing single CPU speed to parallel hardware. This process is most visible with the introduction of multi-core CPUs at the consumer level. At the high end, IBM's Blue Gene project aims to achieve significant performance improvements for supercomputers by focusing on parallel communication, increasing the number of CPUs in a system by more than an order of magnitude. Blue Gene/L currently uses relatively slow 700MHz CPUs, sacrificing single processor speed to meet power, heat and physical density limitations to improve interprocessor communications speed and scale systems to more than a hundred thousand processors, resulting in a Blue Gene/L currently being ranked as the fastest computer in the world, with a total of 16 systems in the top 100 according to Top500.org [1, 2]. The result of these research and development efforts means that parallel scaling should be one of the key drivers of performance for scientific modeling applications.

This chapter focuses on the parallel performance of the FMM implementation, comparing it to the current PME method in NAMD, analyzing the communication costs and the benefits of load balancing.

5.1 Estimation of Communication Costs

The current Particle Mesh Ewald electrostatic computation is characterized by a number of all-to-all broadcast communications at each timestep. For an model GFP system running on 16 processors, the patch grid will be about 4 x 4 x 4. During computation, PME transmits a `PmeGridMsg`, a `PmeTransMsg` and a `PmeUnTransMsg`; each of these being an all-to-all broadcast from each processor to every other one. The `PmeGridMsg` has four main elements: `int zlist[]`, `char fgrid[]`, `float qgrid[]` and `PmeReduction evir[]`. With a PME grid of 64 x 64 x 64, the size of the first

three arrays are approximately 64 x 20 x 20. Multiplying these by their respective word sizes, the `zlist` array is 820 kilobits (kb) in size, with `fgrid` 205 kb and `qgrid` 1.6 megabit (Mb). The `PmeReduction evir[]` element is merely seven double precision floating point numbers, and therefore only 448 bits long. There are several other message elements requiring negligible storage, but this equates to a total message size of about 2.7 megabits. Using similar methodology, the `PmeTransMsg` and `PmeUnTransMsg` are each about 1.6 Mb. Transmitting one of each message type every timestep, this results in 5.9 Mb of message transmission per CPU. Multiplying by the number of processors, this is 43 Mb in 16 broadcast messages. If each broadcast message is treated as 15 individual messages, this is a total of 480 Mb of data in 240 messages to be transmitted each timestep.

Using the cluster Orb as an example system, each node contains two processors and has to receive a total of 165 Mb of data ($5.9Mb * 14 * 2$) each timestep using interconnects capable of transmitting 2.0 gigabits per second. Factoring in communications overhead, this puts a theoretical limit of at most four to eight steps per second. This roughly agrees with the measured performance of one half second per step.

The fast multipole method has four main communication points, the transmission of atom positions to nearby atoms, the sending of the expansion up the tree from each patch to global, the reply from global back down the tree to each patch, and the sending of expansions to intermediate neighbors between patches. For the atoms message, each patch of about 1000 atoms transmits approximately 875 atom positions to each of its 26 nearest neighbor patches. With each atom position containing three 64 bit numbers and factoring in overhead, messages are approximately 300 kb in size. Considering simulation boundaries and the fact many of these neighbor patches reside on the same processor, measurements show that each patch communicates to on average 1.5 other processors. Multiplying this times 64 total patches results in a total of about 29 Mb for the position messages.

Assuming a total of 7 terms in the expansions, there are 39 elements, or a total of about 5 kb for every expansion transmitted. The patch to global and global to patch messages include 9 expansions for a size of around 45 kb. Therefore, the patch to global messages transmits a total of 2.9 Mb in 64 messages and the global to patch messages transmits 2.9 Mb in only 16 messages. For the patch to patch transmissions, each has 72 expansions for a total of 360 kb per message. With each patch transmitting to 1.5 other CPUs, this is a total of 34.6 Mb in 96 separate messages. All told, there are about 70 megabits of data transmitted every step, in a total of 272 messages. Just as importantly, if the atom messages can be removed through proper use of proxy patches, this communications load can be reduced 42%. Because the fast multipole method consists entirely of one-to-one transmissions, this is significantly less communications overhead than the 480 Mb for the PME calculations.

Just as important as the total volume of the communications is the timing of the communications. If the communications and computation can be sufficiently overlapped, the communications come at very little cost. Similarly, if the communications and computation are strictly serialized, the time for transmission of messages can become critical. Charm++ has sophisticated monitoring and analysis tools that can show real performance profiling for parallel programs, including the ability to highlight time spent waiting for communications to arrive. This profile timing data was taken on a number of computer systems when using PME and FMM for the electrostatics calculations in NAMD. Unfortunately, we could not get the Charm++ data analysis and visualization tool, Projections, working to examine the performance. The Charm++ developers were contacted, but unable to resolve the problem in time for inclusion.

5.2 Parallel Scaling

The parallel performance for NAMD is shown in Figure 5.1 for a shared memory SGI Altix. Performance data for more than 32 CPUs was not available because of MPI errors on all jobs requesting 32 or more processors. For the Fast Multipole

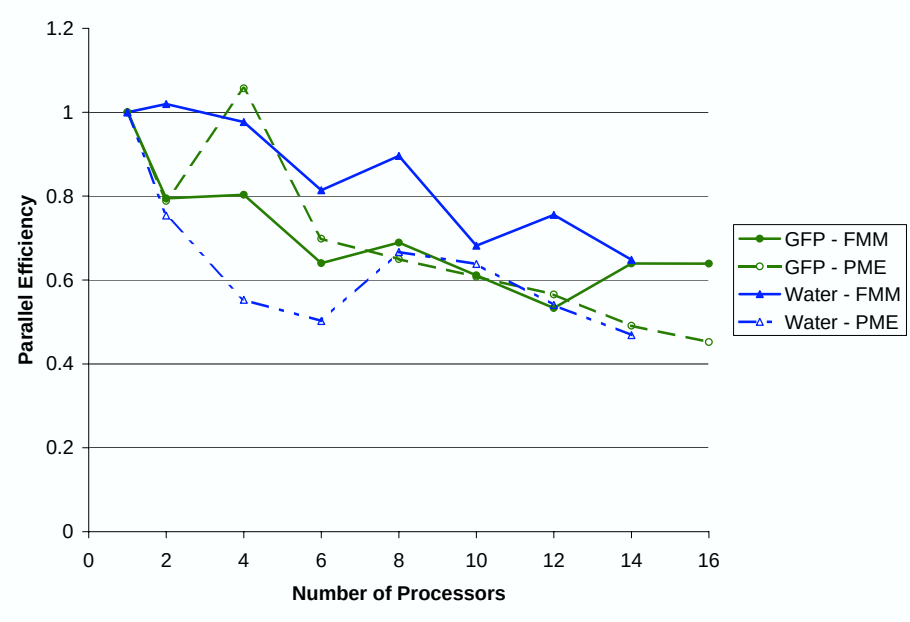


Figure 5.1: This graph shows the parallel efficiency of NAMD using both electrostatic techniques for GFP, and a 97k atom water box. Data was collected on a shared memory, 16 processor SGI Altix.

Method runs, the GFP was enclosed in a water sphere with 57k atoms, the PME data was taken with the protein in a water box and 48 thousand atoms. The water box for PME has 97k atoms and the water sphere used for FMM has 92k atoms. For both systems and both methods, the parallel efficiency was around 60 to 70%, with the Fast Multipole Method handling the larger, 100k atom water simulation better than PME. The actual computation time for FMM took about six times longer than PME, so even at 16 CPUs, the slight parallel advantage of FMM is not enough to overcome the serial speed difference. The previous implementation by Lupo and McKenney showed a parallel efficiency of slightly less than 60% at 16 processors and around 40% on 32 processors [28]. It should be noted, that performance data is with the multiple time stepping feature turned on. By comparison, the current implementation shows a slight advantage, without the additional speedup from MTS. Multiple time stepping should provide a boost in parallel efficiency, because it emphasizes the other portions of the MD engine, which parallelize significantly better than the electrostatics calculation.

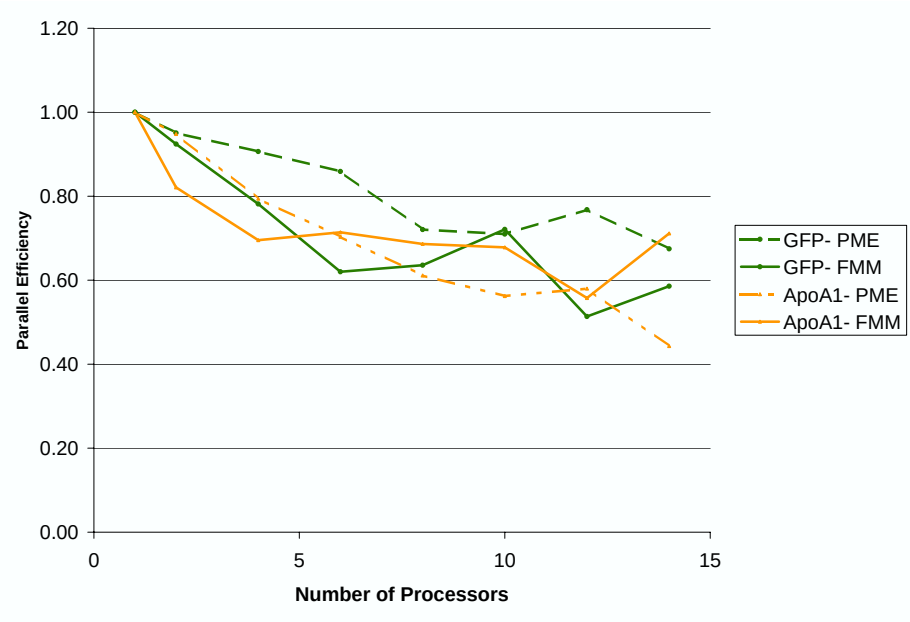


Figure 5.2: This graph shows the parallel efficiency of NAMD using both electrostatic techniques for GFP and ApoA1 run on a 16 processor cluster.

When multiple time stepping is implemented with the current FMM implementation, it should provide a clear advantage over the previous version.

Figure 5.2 shows similar scaling results for the two methods when run on a distributed memory cluster. Here, the overall performance is not significantly worse than for the shared memory system, a result of both the high speed Myrinet interconnects and the 800 MHz processors that are significantly slower than those on the Altix in Figure 5.1. For these tests, the ApoA1 system was used, with 136k atoms in the spherical system used for FMM, and 92k atoms in the box used for PME. Here, the PME method seems more tolerant of the communication latency with the smaller GFP system, but the performance is somewhat similar in the three other cases.

Figure 5.3 shows the constant workload speedup for both PME and FMM. A series of water boxes were run on a SGI Altix, and one processor was used for every ten thousand atoms in the simulation. Here, PME's stability in providing good scaling in both problem size and parallel efficiency show a relatively constant time of about one half second per timestep. When there is more than thirty thousand atoms in

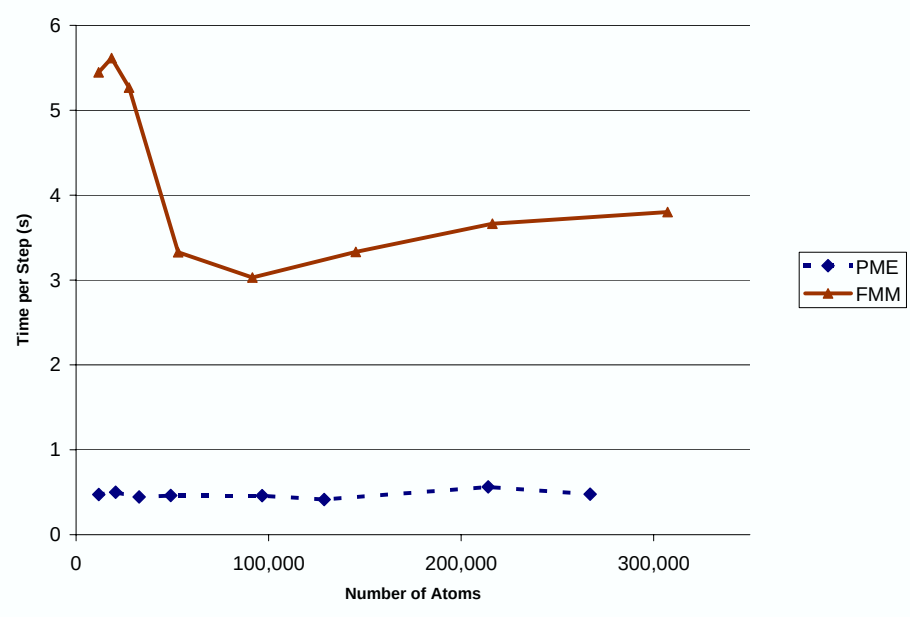


Figure 5.3: This is the constant workload timings for a series of water boxes run on a SGI Altix, the time per step in seconds is shown for PME and FMM. One processor was used for every ten thousand atoms in the simulation.

the simulation, and thus four levels in the tree, the fast multipole method shows decent performance, with only slight degradation as the problem size and number of processors increase. For extremely small simulations, less than 30 thousand atoms, there are only three levels in the tree, so there is very little parallel work distributed among the processors.

5.3 Load Balancing

One of NAMD's most unique features is the ability to handle measurement-based dynamic load balancing, as outlined in Section 3.1.2. The Charm++ framework keeps accurate timing data for all objects during the course of a run. NAMD will periodically consult this data, and move patches and compute objects to different processors. This scheme is very effective, and it is not unusual to see performance gains of 15% or more after one or two rebalancing iterations.

Table 5.1: Load balancing and CPU load after 50 and 150 timesteps

System	Electrostatic Type	100 Steps		150 Steps	
		Average	Max	Average	Max
ApoA1	PME	5.1	7.5	5.4	5.5
ApoA1	FMM	48	97	48	92
GFP	PME	6.7	9.3	6.9	7.1
GFP	FMM	46	93	47	87

Table 5.1 shows the average processor load and maximum processor load from the first two load balancing steps as reported by NAMD. As you can see, in PME, the load is nicely equilibrated at the second stage, with the processor having the heaviest load brought down close to the average. This is not the case with FMM; the maximum load is reduced slightly, but remains at a level that is twice that of the average, showing that there remains a significant load imbalance. This heavy load is the result of the global object. The table shows that the load is brought down slightly, but testing in FMM shows that most patches are not migrated off of the global processor, an effect that would be expected. This shows that either the load balancer is not as effective as would be hoped, or that that work from the global object remains significant, and outstrips the computational work done by several of the patches. As mentioned before, this is not believed to be the case, but because of the unavailability of performance profiling data, the cause of this cannot be determined with certainty.

VI. Conclusions

The work presented here shows the Fast Multipole Method is capable a providing a fast, accurate calculation of the electrostatic potential in molecular dynamics. Tests outlined in Chapter IV show that it models protein structures with accuracy. The current method provides a level of accuracy that is a full order of magnitude better than the current PME implementation, and it has the capability to scale as well as, and in some cases better than, PME. Much of the speed differential may be reduced by using fewer terms in the expansion, but it is unknown if this would still correctly model the protein systems. The current implementation of the Fast Multipole Method is rough, and a number of areas have been identified where clear performance gains can be realized. Benchmarks of NAMD, from version 2.0 to the current version 2.6 showed significant improvement, cutting the time per step in half, and improving the scaling dramatically [36, 38]. The scaling with problem size is remarkably good with PME, equivalent to FMM in these tests. In addition, if common features, particularly multiple time stepping are integrated and tested, this should have a significant effect on the performance. With the current version of FMM running at about half to one fifth of the speed of PME, there is clear potential for the Fast Multipole Method to perform as fast as PME.

This work removed the significant communications load that was a result of the communication of atom positions and results between FMM and the core molecular dynamics engine. This had the effect of better scaling than the implementation by Lupo and McKenney, because we were able to completely eliminate the primary impediment to parallel performance in that implementation. Unfortunately, MPI errors on the large multiprocessor system Eagle prevented the measurement of parallel scaling past 32 CPUs.

6.1 Further Work

The implementation of periodic boundary conditions in FMM in NAMD should allow a more direct measurement of performance and accuracy differentials with PME.

This would allow for a more through exploration of the number of expansion terms necessary to model relevant structures. It would also produce a more general use methodology, as most research is carried out with periodic boundary conditions.

Further exploration needs to be conducted of the parallel performance of this Fast Multipole Method implementation. Fixing problems with the Charm++ performance tracing toolkit would allow for a better examination of communications, providing insight into drawbacks and highlighting potential optimizations.

As noted earlier, significant optimizations remain in the current code. The primary work would be to properly utilize the proxy patch structure in NAMD to eliminate the redundant communication of atom positions in FMM. Cutting out 40% of the communications in the algorithm has the potential to provide significant speedup, although there is no indication that the current implementation is communications limited. In many areas of the code, large memory copies are performed to ease the initial development effort and to reduce the potential for errors and bugs from memory management errors. Further improvements are available by replacing these with pass by reference mechanisms.

Significant other improvements should be available by utilizing a single FMM compute object on every processor, as NAMD does for other force types, instead of the current method of initializing separate compute objects for each patch. This should significantly reduce the memory footprint of FMM, and reduce the communications efforts between patches on a single processor.

The methodology used here of enforcing the Fast Multipole Method boxes to coincide perfectly with the NAMD patch schema successfully eliminated the dominant communications of atom positions and resulting forces. But, it also introduced a number of problems as a result of atoms moving over patch boundaries and non cubic patch grids impeding the implementation of periodic boundary conditions. It is recommended further efforts use a similar schema, starting the building of the FMM tree at the patch level, but allowing for FMM boxes to be the size of, or smaller

than, the individual patches. FMM boxes straddling patch boundaries can be handled through proxy patches. This would allow for a proper recursive bisection of simulation space and a more straightforward implementation of periodic boundary conditions as a result of not having to deal with null boxes and non-cubic trees.

While other implementations of the Fast Multipole Method have been able to show significantly better scaling [26], these implementations did not have to deal with the constraints of interacting with a standard MD engine. The work presented here shows that the Fast Multipole Method can in fact integrate with common molecular dynamics packages, providing good accuracy and performance on par with current PME methods.

Appendix A. Fast Multipole Method Code

A.1 FMM Algorithm

The listing below shows the code for the serial Fast Multipole Method algorithm. The subroutines called here are inherited by the parallel version of the algorithm, and used largely unchanged.

Listing A.1: The C++ code for the primary sequence of the serial Fast Multipole Method algorithm. (Appendix2/ComputeFmm.C)

```
ForceList* ComputeFmm::ComputeForces(CompAtomList *input)  {
    system = input;
    int numAtoms, nLevels;      // parameters
    int level, myi;             // local variables

    numAtoms = system->size();
    nLevels = log(numAtoms)/log(8);
    SizeTree();
    PopulateTree();

//  First, do multipole expansion at lowest level.
    CalculateMP();

//  Then fransform this to center of parent's boxes all the way
//  up the tree.
    for(level=nLevels-1; level>=2; --level)  {
        TransformMPUpTree(level);
    }

//  Convert multipole expansion to local expansion and walk down
    level_end = levelStart[2] + pwrEight[2];
    for(myi=levelStart[2]; myi<level_end; ++myi)  {
        tree[myi].me = 0.0;
    }
    IntermediateLE(2);
    for(level=3; level<=nLevels; ++level)  {
        TransformLEDown(level);
        IntermediateLE(level);
    }

    MPToForces();
    NearInteractions();
    return forces;
}
```

A.2 Parallel Algorithm Data Structures

The following shows the data structures for the fast multipole method implemented as a part of this work.

Listing A.2: The C++ headers for the core fast multipole method and serial implementation. (Appendix2/ComputeFmm.h)

```
struct FmmNode  {
   Domain box;
   CompAtomList *atoms;
   MPole *me;
   MPole *le;
   unsigned int myIndex : 31;
   unsigned int populated : 1;

   FmmNode* parent;
   FmmNode* child[8];
   FmmNode* nneighbor[26];

   FmmNode();
   ~FmmNode();
};


class ComputeFmm  {
  public:
    ComputeFmm();
    ~ComputeFmm();
    void Initialize(int);
    ForceList* ComputeForces(CompAtomList *);

  protected:
    CompAtomList *system;
    FmmNode *tree;
    int splitLevel;
    int numAtoms;
    int nLevels;
    int nBoxes;
    int nTerms;
    int nElements;
    Force* f;
    BigReal energy;

    // Core FMM Algorithm
    void CalculateMP(void);
    void TransformMPUpTree(int);
    void IntermediateLE(int);
    void TransformLEDown(int);
    void MPToForces();
    void NearInteractions(void);
```

```
45
      // The core mathematical multipole functions
      void prodLL(MPole*, MPole*, MPole*);
      void prodLM(MPole*, MPole*, MPole*);
      void evalM(MPole*, Vector);
50    void evalL(MPole*, Vector);

      // Tree building and utility functions
      inline long int FirstChild(int, int);
      inline long int Child(int, int);
55    void BuildTree(void);
      void SizeTree(void);
      void PopulateTree(void);
      void EmptyTree(void);
      void ClearTree(void);
60 };
```

Listing A.3: The Fast Multipole Method Compute Object. (Appendix2/ComputeFmmObj.h)

```
class ComputeFmmObj : public ComputeHomePatches {
    friend class ComputeFmmMgr;
    public:
5       ComputeFmmObj(ComputeID c);
        ~ComputeFmmObj();
        inline void setMgr(ComputeFmmMgr *);
        void doWork(void);
        void PatchDone(PatchID pid, BigReal *evir);
10      BigReal TotalEnergy(void);
        inline int PatchTreeMap(int);
        inline int MyComputeMap(int);
        inline int Step(void);

15  private:
        ComputeFmmMgr *myComputeFmmMgr;
        ComputeFmmPatch *computes;
        ComputeFmmGlobal *global;
        SimParameters *simParams;
20      SubmitReduction *reduction;
        Domain systemSize;

        IntegerList patchTreeMap; // give me a PID i'll return the
                                  //   box in the tree it is in
25      IntegerList myComputeMap; // give me a PID i'll return the
                                  //   element in compute array.

        int myNumPatches;
        int totNumPatches;
30      int nElements;
        unsigned int nTerms : 10;
        unsigned int nLevels : 10;
        unsigned int splitLevel : 10;
        unsigned int initialized : 1;
```

```
        unsigned int unused : 1;
        BigReal totEnergy;
        BigReal fmmEnergy;
        BigReal evir[7];
        int globalNode;
        int numProcs;
        int myNode;
        int numComputes;
        int numDone;
        int lr;                  //level of refinement
        int step;
};
```

Listing A.4: The fast multipole method Charm++ manager. It handles all communication between `ComputeFmmPatch`es as well as all interprocessor communication. It has seperate classes for each type of message, and also implements a message queue in case some messages arrive early. (Appendix2/ComputeFmmMgr.h)

```
class NAMsg : public CMessage_NAMsg  {
    public:
        NAMsg(void ) { ; }
        NAMsg(PatchID, PatchIDList, IntegerList, CompAtomList, int...
            );
        ~NAMsg(void) { ; }
        static void* pack(NAMsg* msg);
        static NAMsg* unpack(void *ptr);

        int step;
        int sourceNode;
        PatchID sourcePatch;
        PatchIDList destPatches;
        IntegerList boxList;
        CompAtomList atoms;

    private:
};

class MPMsg : public CMessage_MPMsg  {
    public:
        MPMsg(void ) { ; }
        MPMsg(PatchID, PatchIDList, IntegerList, BigRealList, int)...
            ;
        ~MPMsg(void) { ; }
        static void* pack(MPMsg* msg);
        static MPMsg* unpack(void *ptr);

        int step;
        int sourceNode;
        PatchID sourcePatch;
        PatchIDList destPatches;
        IntegerList boxes;
```

```
        BigRealList MP;

    private:
};


class ComputeFmmMgr : public Group  {
    public:
        ComputeFmmMgr();
        ~ComputeFmmMgr();
        void initialize(CkQdMsg*);          // Charm++ entry method
        void setCompute(ComputeFmmObj *);
        void SendNeighAtoms(PatchID, PatchIDList *, IntegerList...
            *, CompAtomList *, int);
        void RecvNeighAtoms(NAMsg *);       // Charm++ entry method
        void SendL(PatchID, PatchIDList *, IntegerList *, ...
           MPoleList *, MPole *, int);
        void RecvLGlobal(MPMsg *);          // Charm++ entry method
        void RecvLNeigh(MPMsg *);           // Charm++ entry method
        void SendMDown(MPoleList *, int);
        void RecvMDown(MPMsg *);            // Charm++ entry method
        void DrainMsgQueues(int);
        void DrainGlobalMsgQueue(int);
        void PatchDone(PatchID pid, BigReal energy, BigReal ...
           fmmEnergy);
        inline int NumLevels(void);
        inline int SplitLevel(void);

        SimParameters *simParams;
        Domain systemSize;
        int totNumPatches;


    private:
        CProxy_ComputeFmmMgr computeFmmPxy;
        PatchMgr* myPatchMgr;
        ComputeFmmObj *myComputeObj;
        MsgQueue mQueue;
        int myNumPatches;
        int splitLevel;
        int nLevels;
        int nTerms;
        int nElements;
        int globalNode;
        int numProcs;
        int myNode;
        int numComputes;
        PatchMap* patchMap;   // give me a PID i'll give you the
                              //    processor it's on.
        int lr;               // level of refinement
};
```

```
struct QueuedNAMsg  {
        NAMsg* qMsg;
        QueuedNAMsg* next;
};

struct QueuedMPMsg  {
        MPMsg* qMsg;
        QueuedMPMsg* next;
};

class MsgQueue  {
    public:
        MsgQueue(void);
        ~MsgQueue(void);
        void AddNAMsg(NAMsg *);
        void AddLGlobMsg(MPMsg *);
        void AddLNeiMsg(MPMsg *);
        void AddMDnMsg(MPMsg *);
        NAMsg* PopNAMsgQ(int);
        MPMsg* PopLGlobQ(int);
        MPMsg* PopLNeiQ(int);
        MPMsg* PopMDnQ(int);

    private:
        int numNAMsg;
        int numLGlobMsg;
        int numLNeiMsg;
        int numMDnMsg;
        QueuedNAMsg *naMsgQueueHead;
        QueuedNAMsg *naMsgQueueTail;
        QueuedMPMsg *mpLGlobQueueHead;
        QueuedMPMsg *mpLGlobQueueTail;
        QueuedMPMsg *mpLNeiQueueHead;
        QueuedMPMsg *mpLNeiQueueTail;
        QueuedMPMsg *mpMDnQueueHead;
        QueuedMPMsg *mpMDnQueueTail;
};
```

Listing A.5: The fast multipole method compute object for each patch. (Appendix2/ComputeFmmPatch.h)

```
class ComputeFmmPatch : public ComputeFmm  {
    public:
        ComputeFmmPatch();
        ~ComputeFmmPatch();
        inline void SetComputeObj(ComputeFmmObj *);
        void GiveAtoms(PatchElem *,int);   // get patch from
                                           //   ComputeObj
        void doWork(void);
        void GetNeighAtoms(PatchID, IntegerList *, CompAtomList *,...
           int);
```

```
        void GetNeighL(PatchID, IntegerList *, MPoleList *,int);
        void GetMDown(MPoleList *,int);
        inline void CloseBox(void);

    private:
        ComputeFmmMgr* myComputeFmmMgr;
        ComputeFmmObj* myComputeFmmObj;
        IndexLocation indexLocation;
        IntegerList boxesOnWall;
        PatchIDList myNeighbors;
        PatchElem *myPatch;
        PatchID myPid;
        Results *results;
        BigReal evir[7];
        int step;
        unsigned int selfDone : 1;     // 1 if done self
                                       //    interactions
        unsigned int gotM : 1;
        unsigned int mpDone : 1;       // 1 if done with Multipole
                                       //    contributions to patch
        unsigned int numNeighDir : 5; // # patches received atom
                                       //    position for direct
                                       //    calc max 26
        unsigned int numNeighL : 5;    // # patches i've received
                                       //    Multipole expansions
                                       //    from max 26
        unsigned int numPatches : 19;

        void NearInteractions2(void);
        void MPToForces2(void);
        inline void AddAtoms(int, CompAtomList*, IntegerList*);
        void GoBackDown(void);
        void AllDone(void);
};

class IndexLocation  {
    public:
        IndexLocation(void);
        ~IndexLocation(void);
        void Initialize(int);
        void Add(int);
        int ArrIndex(int);

    private:
        int length;
        int counter;
        int* atomIDs;
        int* location;
};
```

Listing A.6: The fast multipole method compute object that handles the tree above the split level. (Appendix2/ComputeFmmGlobal.h)

```
class ComputeFmmGlobal : public ComputeFmm  {
    public:
        ComputeFmmGlobal(ComputeFmmObj *);
        ~ComputeFmmGlobal();
        void doWork(int);
        inline int Step(void);
        void GetL(PatchID, MPole *, int);

    private:
        void UpTree();
        void AllDone();
        ComputeFmmMgr* myComputeFmmMgr;
        ComputeFmmObj* myComputeFmmObj;
        unsigned int numPatches;
        unsigned int numLRecvd : 31;   // # of patches i've ...
           received
                                       //    Multipoles from
        unsigned int treeReady : 1;
        int step;
};
```

Bibliography

1. "Top 500 List, November 2006". http://www.top500.org/list/2006/11/100, 2006.
2. "IBM Research Blue Gene". http://research.ibm.com/bluegene, 2007.
3. Agarwal, Tarun, Amit Sharma, and Laxmikant V. Kalé. "Topology-aware task mapping for reducing communication contention on large parallel machines". *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, 279–284. Institute of Electrical and Electronics Engineers (IEEE), 2006.
4. Allinger, N. L. "Conformational Analysis". *J. Amer. Chem. Soc.*, 99:8127–8134, 1977.
5. Ambrosiano, J., L. Greengard, and Vladimir Rokhlin. "The fast multipole method for gridless particle simulation". *Comp. Phys. Commun.*, 48:117–125, 1988.
6. Barnes, John and P. Hut. "A Hierarchical $O(N \log N)$ Force-Calculation Algorithm". *Nature*, 324:446–449, 1986.
7. Bhandarkar, M., R. Brunner, C. Chipot, A. Dalke, S. Dixit, P. Grayson, J. Gullingsrud, A. Gursoy, D. Hardy, J. Hnin, W. Humphrey, D. Hurwitz, N. Krawetz, S. Kumar, M. Nelson, J. Phillips, A. Shinozaki, G. Zheng, and F. Zhu. "NAMD User's Guide". http://www.ks.uiuc.edu/Research/namd/current/ug/, 2005.
8. Bishop, Thomas C., Robert D. Skeel, and Klaus Schulten. "Difficulties with Multiple Time Stepping and Fast Multipole Algorithm in Molecular Dynamics". *J. Comp. Chem.*, 18:1785–1791, 1997.
9. Board, John, Jeffrey Causey, Jr. James Leathrum, Andreas Windemuth, and Klaus Schulten. "Accelerated molecular dynamics simulation with the parallel fast multipole algorithm". *Chem. Phys. Lett.*, 198:89–94, 1992.
10. Carlson, B. and G. Rushbrooke. "On the Expansion of a Coulomb Potential in Spherical Harmonics". *Proc. Camb. Phil. Soc.*, 46:626, 1950.
11. Case, D.A., III T.E. Cheatham, T. Darden, H. Gohlke, R. Luo, K.M. Merz, Jr., A. Onufriev, C. Simmerling, B. Wang, and R. Woods. "The Amber biomolecular simulation programs". *J. Comp. Chem.*, 26:1668–1688, 2005.
12. Darden, T. A., D. M. York, and L. G. Pedersen. "Particle Mesh Ewald, An $N \log(N)$ method for Ewald sums in large systems". *J. Chem. Phys.*, 98:10089, 1993.
13. Elliott, W. D. and J. A. Board. "Fast Fourier Transform Accelerated Fast Multipole Method". *SIAM J. Sci. Comput.*, 17:398–415, 1996.
14. Esselink, Klaas. "The order of Appel's algorithm". *Information Processing Let.*, 41:141–147, 1992.

15. Essmann, Ulrich, Lalith Perera, Max Berkowitz, Tom Darden, Hsing Lee, and Lee Pederson. "A smooth particle mesh Ewald method". *J. Chem. Phys.*, 103:8577, 1995.

16. Ewald, Paul P. "Die Berechnung optischer und elektrostatischer Gitterpotentiale". *Ann. Physik*, 64:253, 1921.

17. Freddolino, Peter L., Anton S. Arkhipov, Steven B. Larson, Alexander McPherson, and Klaus Schulten. "Molecular dynamics simulations of the complete satellite tobacco mosaic virus". *Structure*, 14:437–449, 2006.

18. Greengard, L. and Vladimir Rokhlin. "A Fast Algorithm for Particle Simulations". *J. Comp. Phys.*, 73:325–348, 1987.

19. Hockney, R. and J. Eastwood. *Computer Simulation Using Particles*. Adam Hilger, New York, NY, 1988.

20. Humphrey, W., A. Dalke, and K. Schulten. "VMD - Visual Molecular Dynamics". *J. Molec. Graphics*, 14:33–38, 1996.

21. Jackson, John David. *Classical Electrodynamics*. Wiley and Associates, New York, NY, third edition, 1998.

22. Kalé, Laxmikant and S. Krishnan. *Parallel Programming using C++*. MIT Press, Cambridge, MA, 1996.

23. Kalé, Laxmikant, Robert Skeel, Milind Bhandarkar, Robert Brunner, Attila Gursoy, Neal Krawetz, James Phillips, Aritomo Shinozaki, Krishnan Varadarajan, and Klaus Schulten. "NAMD2: Greater scalability for parallel molecular dynamics". *J. Comp. Phys.*, 151:283–312, 1999.

24. Kumar, Sameer, Chao Huang, Gheorghe Almasi, and Laxmikant V. Kale. "Achieving Strong Scaling with NAMD on Blue Gene/L". *Proceedings of IEEE International Parallel & Distributed Processing Symposium 2006*, 2006.

25. Kurzak, Jakub and B. Montgomery Pettitt. "Communications overlapping in fast multipole particle dynamics methods". *J. Comp. Phys.*, 203:731–743, 2005.

26. Kurzak, Jakub and B. Montgomery Pettitt. "Massively parallel implementation of a fast multipole method for distributed memory machines". *J. Parall. and Dist. Comp.*, 65:870–881, 2005.

27. Lindahl, E., B. Hess, and D. van der Spoel. "GROMACS 3.0: A package for molecular simulation and trajectory analysis". *J. Mol. Mod.*, 7:306–317, 2001.

28. Lupo, James, Zhiqiang Wang, Alan McKenney, Ruth Pachter, and William Mattson. "A large scale molecular dynamics simulation code using the fast multipole algorithm (FMD): performance and application". *J. Molec. Graphics and Modeling*, 21:89–99, 2002.

29. MacKerell, Alexander D., D. Bashford, M. Bellott, R. L. Dunbrack, Jr., J. D. Evanseck, M. J. Field, S. Fischer, J. Gao, H. Guo, S. Ha, D. Joseph-McCarthy,

L. Kuchnir, K. Kuczera, F. T. K. Lau, C. Mattos, S. Michnick, T. Ngo, D. T. Nguyen, B. Prodhom, W. E. Reiher III, B. Roux, M. Schlenkrich, J. C. Smith, R. Stote, J. Straub, M. Watanabe, J. Wirkiewicz-Kuczera, D. Yin, and M. Karplus. "All-Atom Empirical Potential for Molecular Modeling and Dynamics Studies of Proteins". *J. Phys. Chem. B*, 102(18):3586–3616, April 1998.

30. Nordelli, M. B., J. L. Fattebert, and J. Bernholc. *Phys. Rev. B*, 64:245–263, 2001.

31. Oakley, Aaron, Timothy Isgro, and Yi Wang. "NAMD Topology File Tutorial". http://www.ks.uiuc.edu/Training/Tutorials, 2005.

32. Petersen, Henrik G. "Accuracy and Efficiency of the Particle Mech Ewald Method". *J. Chem. Phys.*, 103:3668–3679, 1995.

33. Pfalzner, Susanne and Paul Gibbon. *Many-Body Tree Methods in Physics*. Cambridge University Press, Cambridge, UK, first edition, 1996.

34. Phillips, James C., Rosemary Braun, Wei Wang, James Gumbart, Emad Tajkhorshid, Elizabeth Villa, Christophe Chipot, Robert D. Skeel, Laxmikant Kalé, and Klaus Schulten. "Scalable molecular dynamics with NAMD". *J. Comp. Chem.*, 26:1781–1802, 2005.

35. Phillips, James C., Robert Brunner, Aritomo Shinozaki, Milind Bhandarkar, Neal Krawetz, Laxmikant Kalé, Robert D. Skeel, and Klaus Schulten. *Avoiding algorithmic obfuscation in a message-driven parallel MD code*. Springer-Verlag, 1999.

36. Phillips, James C. and Klaus Schulten. "Theoretical and Computational Biophysics Group, NAMD Performance". http://www.ks.uiuc.edu/Research/namd/performance.html, 2005.

37. Phillips, James C., Willy Wriggers, Zhigang Li, Ana Jonas, and Klaus Schulten. "Predicting the structure of apolipoprotein A-I in reconstituted high density lipoprotein disks". *Biophysical Journal*, 73:2337–2346, 1997.

38. Phillips, James C., Gengbin Zeng, Sameer Kumar, and Laxmikant Kalé. "NAMD: Biomolecular Simulation on Thousands of Processors". *Proceedings of the IEEE/ACM SC2002 Conference.*, 277, 2002.

39. Plimpton, Steve J. "Fast Parallel Algorithms for Short-Range Molecular Dynamics". *J. Comp. Phys.*, 117:1–19, 1995.

40. Procacci, P., T. Darden, and M. Marchi. "A very fast molecular dynamics method to simulate biomolecular systems with realistic electrostatic interactions". *J. Chem. Phys.*, 100:10464–10468, 1996.

41. Procacci, P, M. Marchi, and G. J. Martyna. "Electrostatic calculations and multiple time scales in molecular dynamics simulation of flexible molecular systems". *J. Chem. Phys.*, 108:8799–8803, 1998.

42. Rankin, W. and J. Board. "A portable distributed implementation of the parallel multipole tree algorithm". *IEEE Symposium on High Performance Distributed*

Computing, volume 002. Institute of Electrical and Electronics Engineers (IEEE), Duke University Technical Report, 1995.

43. Rapaport, Dennis. *The Art of Molecular Dynamics Simulation*. Cambridge University Press, Cambridge, UK, second edition, 2004.

44. Sagui, Celeste and Thomas Darden. "Multigrid methods for classical molecular dynamics simulations of biomolecules". *J. Chem. Phys.*, 114:6578–6591, 2001.

45. Sandak, Bilha. "Multiscale Fast Summation of Long-Range Charge and Dipolar Interactions". *J. Comp. Chem.*, 22:717–731, 2001.

46. Sandu, Adrian and Tamar Schlick. "Masking Resonance Artifacts in Force-Splitting Methods for Biomolecular Simulations by Extrapolative Langevin Dynamics". *J. Comp. Phys.*, 151:74–133, 1999.

47. Sarikaya, Mehmet, Candan Tamerler, Alex K Jen, Klaus Schulten, and Francois Baneyx. "Molecular biomimetics: nanotechnology through biology". *Nature Materials*, 2:577–585, 2003.

48. Schlick, Tamar. *Molecular Modeling and Simulation: An Interdisciplinary Guide*. Springer-Verlag, New York, NY, first edition, 2002.

49. Schmidt, K. E. and M. A. Lee. "Implementing the Fast Multipole Method in Three Dimensions". *J. Stat. Phys.*, 63:1223–1235, 1991.

50. Schmidt, K. E. and M. A. Lee. "Multipole Ewald Sums for the Fast Multipole Method". *J. Stat. Phys.*, 89:1223–1235, 1997.

51. Skeel, Robert, Ismail Tezcan, and David Hardy. "Multiple Grid Methods for Classical Molecular Dynamics". *J. Comp. Chem.*, 23:673–684, 2002.

52. Street, W. B., D. J. Tildesley, and G. Saville. *Mol. Phys.*, 35:639, 1978.

53. Stryer, Lubert, Jeremy M. Berg, and John L. Tymoczko. *Biochemistry*. W. H. Freeman and Company, New Your, NY, fifth edition, 2002.

54. Walsh, Phillip, Andrey Omeltchenko, Rajiv K. Kalia, Aiichiro Nakano, and Priya Vashishtaa. "Nanoindentation of silicon nitride: A multimillion-atom molecular dynamics study". *Appl. Phys. Let.*, 14:118–120, 2006.

55. Warren, Michael S. and John K. Salmon. "A Parallel Hashed Oct-Tree N-Body Algorighm". *Proceedings of Supercomputing 1993*, 12–21, 1993.

56. White, Christopher A. and Martin Head-Gordon. "Rotating around the quartic angular momentum barrier in fast multipole method calculations". *J. Chem. Phys.*, 105:5061–5067, 1996.

57. Zeng, Gengbin and Laxmikant Kalé. "Charm++ Programming Manual". http://charm.cs.uiuc.edu/manuals/html/charm++/3_11Load_Balancing.html, 2007.

58. Zimmer, Marc. “Green Fluorescent Protein: Applications, Structure, and Related Photophysical Behavior”. *Chem. Rev.*, 102:759–781, 2002.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
13-06-2007	Master's Thesis	Jan 2005 - Jun 2007

4. TITLE AND SUBTITLE

Parallel Fast Multipole Method For Molecular Dynamics

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Ormseth, Reid, G., Captain, USAF

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB, OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GAP/ENP/07-J02

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFRL/MLPJ
Attn: Dr. Ruth Pachter
3005 Hobson Way
WPAFB, OH 45433 DSN:785-9689

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

We report on a parallel version of the Fast Multipole Method (FMM) implemented in the classical molecular dynamics code, NAMD (Not Another Molecular Dynamics program). This novel implementation of FMM aims to minimize interprocessor communication through the modification of the FMM grid to match the hybrid force and spatial decomposition scheme already present in NAMD. This new implementation has the benefit of replacing all-to-all communications broadcasts with direct communications between nearest neighbors. This results in a significant reduction in the amount of communication compared to earlier attempts to integrate FMM into common molecular dynamics programs. The early performance of FMM is similar to the existing electrostatics methods already in NAMD. In addition, tests of the stability and accuracy of the FMM algorithm in molecular dynamics as applied to several common solvated protein structures are discussed.

15. SUBJECT TERMS

Molecular Dynamics, Fast Multipole Method, Electrostatics, Parallel Processing, High Performance Computing.

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			David E. Weeks, PhD
U	U	U	UU	81	19b. TELEPHONE NUMBER *(Include area code)* (937) 255-3636 x4561, e-mail: david.weeks@afit.edu