An $O(N)$ Algorithm
for Three-dimensional $N$-body Simulations

by

Feng Zhao

B.S. Chiao Tung University, Shanghai, China (1984)

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Master of Science
in Electrical Engineering and Computer Science
at the
Massachusetts Institute of Technology
October, 1987

©Massachusetts Institute of Technology, 1987

Signature of Author

Department of Electrical Engineering and Computer Science
October 2, 1987

Certified by

Harold Abelson
Associate Professor of Computer Science and Engineering
Thesis Supervisor

and

Gerald Jay Sussman
Professor of Electrical Engineering
Thesis Supervisor

Accepted by

Arthur C. Smith
Chairman, Department Committee
An $O(N)$ Algorithm for Three-dimensional $N$-body Simulations

by

Feng Zhao

Submitted to the Department of Electrical Engineering and Computer Science on October 2, 1987 in partial fulfillment of the requirements for the Degree of Master of Science in Electrical Engineering and Computer Science

Abstract

We develop an algorithm that computes the gravitational potentials and forces on $N$ point-masses interacting in three-dimensional space. The algorithm, based on analytical techniques developed by Rokhlin and Greengard, runs in order $N$ time. In contrast to other fast $N$-body methods such as tree codes, which only approximate the interaction potentials and forces, this method is exact—it computes the potentials and forces to within any prespecified tolerance up to machine precision. We present an implementation of the algorithm for a sequential machine. We numerically verify the algorithm, and compare its speed with that of an $O(N^2)$ direct force computation. We also describe a parallel version of the algorithm that runs on the Connection Machine in order $O(\log N)$ time. We compare experimental results with those of the sequential implementation and discuss how to minimize communication overhead on the parallel machine.

Keywords: $N$-body algorithm, particle simulation, Multipole expansion, 3-D tree, spatial refinement, parallel computing

Thesis Supervisor: Harold Abelson
Title: Associate Professor of Computer Science and Engineering

Thesis Supervisor: Gerald Jay Sussman
Title: Professor of Electrical Engineering
To my Father and Mother
Acknowledgments

I would like to thank the many people who have contributed to the preparation of this thesis.

Hal Abelson and Gerry Sussman, my thesis advisors, have provided encouragement and support throughout the course of this research. Gerry deserves thanks for getting me started on the problem and for pointing out what is obvious and what is not. Hal has offered excellent critique of my writing and has improved the clarity of many ideas in the thesis. I have benefited a great deal from inspiring discussions with Hal and Gerry which led to the development of this thesis.

Many people in the MAC research group at MIT have provided helpful comments and assistance. Professor Jacob Katzenelson from Technion—Israel Institute of Technology—read and commented on early versions of the thesis. Franklyn Turbak has been a major source of help, and has spent hours and hours helping me prepare the final draft of this thesis. Mark Sheldon has provided help with typesetting problems. Andy Beriin and Ken Yip have provided constructive criticism.

I thank Thinking Machines Corporation for providing research facilities for my experimental work. Special thanks are to Danny Hillis, Lennart Johnson, Alan Ruttenberg, and Luis Ortiz.

I thank Vicki Ledet and Craig Withers for their constant belief in me, and Jane Poet for helping me spiritually maintain an active life.

This thesis would never have been completed without the moral support and love from my family.

This thesis describes research done at the Artificial Intelligence Laboratory and the Laboratory for Computer Science at the Massachusetts Institute of Technology, supported by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-86-K-0180.
# Contents

1 Introduction 1

2 The $N$-body Algorithm in 2-D 4
   2.1 The Multipole Expansion Method 4
   2.2 Description of the Algorithm 6
   2.3 Remarks 9

3 A 3-D $N$-body algorithm 11
   3.1 The Multipole Expansion in 3-D 12
   3.2 Theorems 13
   3.3 A Sequential Algorithm 27

4 Numerical Verifications 32
   4.1 A Sequential Implementation 32
      4.1.1 3-D Tree Data Structure 32
      4.1.2 Implementation Details 33
   4.2 Experimental Verifications 35
      4.2.1 Accuracy of the Algorithm 35
      4.2.2 Running Time of the Algorithm 40
   4.3 Discussion 43

5 A Parallel Algorithm 45
   5.1 Introduction 45
   5.2 Why a Parallel Computer? 46
   5.3 3-D Tree on the Parallel Computer 47
   5.4 A Parallel Algorithm 48
   5.5 Experimental Results 51
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Decomposition in a two-dimensional space</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>Near-field, far-field, and interactive-field</td>
<td>8</td>
</tr>
<tr>
<td>3.1</td>
<td>Gravitational field</td>
<td>13</td>
</tr>
<tr>
<td>3.2</td>
<td>Lemma 3.2.1</td>
<td>21</td>
</tr>
<tr>
<td>3.3</td>
<td>Lemma 3.2.2</td>
<td>23</td>
</tr>
<tr>
<td>3.4</td>
<td>Lemma 3.2.3</td>
<td>26</td>
</tr>
<tr>
<td>3.5</td>
<td>Decomposition in a three-dimensional space</td>
<td>28</td>
</tr>
<tr>
<td>3.6</td>
<td>A sequential algorithm</td>
<td>30</td>
</tr>
<tr>
<td>3.7</td>
<td>A sequential algorithm (con’t)</td>
<td>31</td>
</tr>
<tr>
<td>4.1</td>
<td>Initial configuration of the Pythagorean configuration of three bodies</td>
<td>36</td>
</tr>
<tr>
<td>4.2</td>
<td>Plot of the accuracy test for the Pythagorean configuration of three bodies</td>
<td>38</td>
</tr>
<tr>
<td>4.3</td>
<td>Initial configuration of two Plummer systems</td>
<td>41</td>
</tr>
<tr>
<td>4.4</td>
<td>Running time growth rate of the Multipole-expansion algorithm</td>
<td>42</td>
</tr>
<tr>
<td>5.1</td>
<td>A parallel algorithm</td>
<td>49</td>
</tr>
<tr>
<td>5.2</td>
<td>A parallel algorithm (con’t)</td>
<td>50</td>
</tr>
<tr>
<td>5.3</td>
<td>Running time growth rate of the parallel algorithm</td>
<td>52</td>
</tr>
<tr>
<td>5.4</td>
<td>Concurrent write</td>
<td>55</td>
</tr>
<tr>
<td>5.5</td>
<td>Combining messages</td>
<td>56</td>
</tr>
<tr>
<td>5.6</td>
<td>Delegating messages</td>
<td>56</td>
</tr>
<tr>
<td>5.7</td>
<td>Superimposed mapping</td>
<td>58</td>
</tr>
<tr>
<td>5.8</td>
<td>Non-superimposed mapping</td>
<td>58</td>
</tr>
<tr>
<td>5.9</td>
<td>Classification of squares</td>
<td>60</td>
</tr>
<tr>
<td>5.10</td>
<td>Symmetric communication pattern</td>
<td>60</td>
</tr>
</tbody>
</table>
List of Tables

4.1 Accuracy test for the Pythagorean configuration of three bodies 37
4.2 Accuracy test for the uniform model 40
4.3 Accuracy test for the Plummer model 41
4.4 Speed test for the Multipole-expansion algorithm 42

5.1 Experimental results on the Connection Machine 52
Chapter 1

Introduction

Numerical simulations of $N$-body interactions are extremely important in astrophysics, plasma physics, fluid dynamics, and molecular dynamics. To accomplish such a simulation by directly evaluating the interaction potentials requires computation time on the order of $O(N^2)$, which is prohibitively expensive for large $N$. Consequently, there have been many efforts to develop algorithms whose complexity grows more slowly [Lec72]. These algorithms all make assumptions about the distribution of particles to approximate the interaction potentials, and thus are applicable only to certain classes of $N$-body interactions.

The best-known "fast" algorithm is the tree-code [App85], which uses a tree-like data structure to form hierarchical clusters of particles. The algorithm approximates the force exerted by a cluster on a distant particle as the force exerted by an equivalent mass located the cluster's center of mass. The overall computational complexity of this algorithm is believed to be $O(N \log N)$. However, the force approximations are effective when the particle distribution is relatively nonuniform, and there are no known a priori estimates of the accuracy of these approximations, although methods have been developed for use in practice [Por85] [BH86].

Rokhlin and Greengard [GR86] discovered an $N$-body algorithm with complexity $O(N)$. The algorithm separates the computation of the far-field potentials from that of the near-field potentials, and expands the far-field potentials into Taylor series. Significantly, given a required precision $\epsilon$, one can determine in advance how many terms must be retained in the Taylor series expansions, and thus one can control the accuracy of the approximation.
CHAPTER 1. INTRODUCTION

Rokhlin and Greengard’s approach relies heavily upon analytical techniques to develop the series expansions, to derive error estimates, and to shift series expansions between coordinate frames. For the $N$-body problem in two dimensions, this analysis is greatly simplified by modeling two-dimensional space as the complex plane and using the theory of functions of a complex variable.

This thesis applies Rokhlin and Greengard’s idea to develop an $O(N)$ algorithm for the three-dimensional $N$-body problem. Although there is no direct analogue of complex analysis in three dimensions, we are able to exploit the harmonic nature of the gravitational potential in order to produce suitable series expansions. We implement the algorithm and experimentally verify its accuracy and its linear growth. We also implement a parallel version of the algorithm and present experimental results.

Chapter 2 of the thesis reviews Rokhlin and Greengard’s algorithm and their use of complex variables for the two-dimensional problem and points out difficulties in applying the method in three dimensions.

Chapter 3 develops a three-dimensional analogue of Rokhlin and Greengard’s method. The basic idea is to expand the potentials in terms of spherical harmonics. The analytical basis of the algorithm includes two theorems that derive the required expansions and give bounds on the error terms. There are also three lemmas that describe how to shift these expansions from one origin to another. Using these analytical results, we present a sequential algorithm in three-dimensions. Independently of this work, a similar extension of Rokhlin and Greengard’s method has been carried out by Greengard [Gre87]. His technique differs from ours in that he uses expansions in polar coordinates and we work in Cartesian coordinates.

Chapter 4 concerns numerical verifications of the three-dimensional algorithm. We first describe a sequential implementation of the algorithm. We discuss our choice of a tree-shaped data structure and describe heuristic methods for increasing the efficiency of the implementation. We then present experimental results to demonstrate that the algorithm does have linear growth and does compute the forces and potentials to within any pre-specified tolerance up to machine precision. We compare the speed of the algorithm with that of an $O(N^2)$ direct force computation. For a required average accuracy of $10^{-4}$ for potential fields, our algorithm will be faster than the direct force computation when there are more than 1,000 particles.

Chapter 5 presents an extremely fast implementation of the $N$-body
code—a parallel implementation of the algorithm, which runs in $O(\log N)$ time, implemented on the Connection Machine. We compare the experimental results with that of the sequential implementation and find that the parallel algorithm has a speed-up of 10 for $N = 1,000$ and a speed-up of 100 for $N = 10,000$. For a large parallel machine, interprocessor communication is often the bottleneck of the computation. By exploiting regularity and locality in communication patterns, by combining and delegating messages, we demonstrate how to minimize communication overhead due to message congestion.

Throughout this thesis, if not specified otherwise, $N$ will always refer to the total number of particles in a simulation, and $p$ will always refer to the highest degree of harmonics retained in an expansion.
Chapter 2

The $N$-body Algorithm in Two Dimensions

This chapter follows the presentation of the two-dimensional algorithm given by Rokhlin and Greengard in [GR86]. We review the major theorems and give a heuristic description of the algorithm. The key idea is to use complex analysis to produce a series expansion of the potentials.

2.1 The Multipole Expansion Method

In the two-dimensional $N$-body problem, we consider point charges and Coulombic forces. For a charge $q$ located at the point $y = (y_1, y_2) \in \mathbb{R}^2$, the potential at a point $x = (x_1, x_2)$ is $q \log \| x - y \|$. If we identify $\mathbb{R}^2$ with the complex plane $\mathbb{C}^1$ via $(x_1, x_2) \leftrightarrow x_1 + ix_2$, we can consider the potential to be the real part of the analytic function $\phi_y : \mathbb{C}^1 \rightarrow \mathbb{C}^1$

$$\phi_y(x) = q \log(x - y).$$

We can expand $\phi_y(x)$ in a Taylor series that converges for any $x$ with $|x| > |y|:

$$\phi_y(x) = q \log(x - y) = q \left[ \log(x) + \log \left( 1 - \frac{y}{x} \right) \right] = q \left[ \log(x) - \sum_{k=1}^{\infty} \frac{1}{k} \left( \frac{y}{x} \right)^k \right]$$

If we retain only $p$ terms of the expansion, the error is bounded:
\[
\left| \phi(y) - q \log(x) + q \sum_{k=1}^{p} \frac{1}{k} \left( \frac{y}{x} \right)^k \right| \leq \frac{q}{1 - \frac{y}{x}} \left| \frac{y}{x} \right|^{p+1}.
\]

This observation leads to the following multipole expansion: Given an ensemble of \( m \) charges \( \{q^{(i)}, i = 1, \ldots, m\} \) located at points \( \{y^{(i)}, i = 1, \ldots, m\} \), with \( |y^{(i)}| \leq r_0 \), the potential \( \phi^{(i)}(x) \) at any point \( x \in C^1 \) due to the charge \( q^{(i)} \) at \( y^{(i)} \), with \( |x| > r_0 \), is

\[
\phi^{(i)}(x) = q^{(i)} \left[ \log(x) - \sum_{k=1}^{\infty} \frac{1}{k} \left( \frac{y^{(i)}}{x} \right)^k \right], \quad i = 1, \ldots, m. \tag{2.1}
\]

The total potential at the point \( x \in C^1 \) due to all the charges, therefore, is

\[
\phi(x) = \sum_{i=1}^{m} \phi^{(i)}(x) = Q \log(x) + \sum_{k=1}^{\infty} \frac{a_k}{x^k}, \tag{2.2}
\]

where

\[
Q = \sum_{i=1}^{m} q^{(i)} \quad \text{and} \quad a_k = \sum_{i=1}^{m} \frac{-q^{(i)} \left( y^{(i)} \right)^k}{k}.
\]

Moreover, for any \( p \geq 1 \),

\[
\left| \phi(x) - Q \log(x) - \sum_{k=1}^{p} \frac{a_k}{x^k} \right| \leq \frac{A}{1 - \frac{r_0}{x}} \left| \frac{r_0}{x} \right|^{p+1},
\]

where

\[
A = \sum_{i=1}^{m} |q^{(i)}|.
\]

In particular, if \( |r_0/x| < 1/2 \), we have

\[
\left| \phi(x) - Q \log(x) - \sum_{k=1}^{p} \frac{a_k}{x^k} \right| \leq 2A \left| \frac{1}{x} \right|^{p+1}.
\]
CHAPTER 2. THE N-BODY ALGORITHM IN 2-D

To obtain an approximation accurate to within a given precision \( \epsilon \), it suffices to retain only the first \( p \) terms in the multipole expansion (2.2) and to take \( p \) to be of the order \( \lfloor -\log_2 \epsilon \rfloor \).

The ability to sum individual expansions of (2.1) to obtain the multipole expansion of (2.2) depends on the fact that these expansions \( \{ \phi_i(x), i = 1, \ldots, m \} \) have a common reference point and a common region of convergence. If they don’t, we need to shift the series \( \{ \phi_i(x), i = 1, \ldots, m \} \) to a common reference point. Also, to ensure that all the shifted series have a common region of convergence, we may need to "flip" a series so that its region of convergence lies inside a disk, rather than outside a disk. Rokhlin and Greengard [GR86] derive shifting and flipping lemmas that describe how to perform these operations in time independent of the total number of particles. These lemmas allow us to manipulate the multipole expansions in a manner required by the following fast algorithm.

2.2 Description of the Algorithm

Rokhlin and Greengard’s \( O(N) \) algorithm computes separately the potentials of close-by particles ("near-field potentials") and the potentials of far-away particles ("far-field potentials"). The near-field potentials are computed using direct evaluation. If the number of particles near any given particle is bounded by a small constant, that is, if the distribution of particles is relatively uniform, then the work required to compute the near-field potential at this particle is of order \( O(1) \). The far-field potential is obtained by evaluating the \( p \)-term multipole expansion described above at the particle position, which takes constant number of operations for a prespecified \( p \). The computation is organized so that the total amount of work for computing the potentials at all the \( N \) particles is of order \( O(N) \).

More precisely, the two-dimensional space under consideration is regarded as a square (Figure 2.1). It is divided into four subsquares, and each of which is then recursively subdivided into four sub-subsquares, and so on. This decomposition is represented using a tree-like data structure in which each square is represented by a node. A square at level \( l \) of the tree has four child nodes that represent the subsquares at level \( l + 1 \). This recursive decomposition is continued until there are only several particles in a square at finest
grain. Under the assumption that the particles are uniformly distributed\(^1\), the level of the tree \(n\), i.e., the level of the decomposition, is usually chosen as about \(\lceil \log_4 N \rceil\). Therefore each square at the finest level has in average one or two particle in it.

We define for a square its near-field, far-field, and interactive-field. The near-field of a square consists of those neighboring squares that are at the same level of decomposition as the square. In Figure 2.2, the squares labeled as \(B\) are in the near-field of square \(A\). The far-field of a square is defined to be the exterior area of its near-field. The interactive-field of a square is the part of its far-field that is in the near-field of the square’s parent. In Figure 2.2, the squares labeled as \(A'\) are in the interactive-field for the square \(A\).

The goal of the computation is to compute the far-field potential expansions at all particle positions with \(O(N)\) work. This is achieved by propagating values, first upwards through the tree, then downwards, as follows:

Initially, for each of the squares at the finest level, we compute the \(p\)-term multipole expansion, valid outside the square, for the potential due to the charges inside the square. The expansion is performed relative to the center of the square.

\(^1\)Rokhlin [CGR87] has an \(O(N)\) adaptive form of the algorithm which does not depend for its performance on the particle distributions.
These expansions are propagated upwards through the tree to produce, for every square at every level, a multipole expansion for the potential due to all charges located inside the square. The valid region of convergence for the multipole expansion is the far-field of the square. To form the multipole expansion for a square at level $l$, we take each of its level-$l+1$ subsquares, shift their multipole expansions to the center of the level-$l$ square and add them together. This is done recursively at each level of the tree while propagating upwards through the tree.

The downward-propagation phase of the computation produces, for every square, a local potential expansion due to its far-field interactions. The local expansion for a square is a multipole expansion that is valid inside the square. The computation proceeds recursively. Suppose we already have a local potential expansion $\phi'_{A_p}(x)$ for square $A$'s parent $A_p$. The local expansion $\phi'_A(x)$ for the square $A$ is, by the definition of an interactive-field, the sum of $\phi'_{A_p}(x)$ and those multipole expansions due to particles in $A$'s interactive-field. However, $\phi'_{A_p}(x)$ is relative to the center of $A_p$. In order to combine it with other multipole expansions for the square $A$, we shift $\phi'_{A_p}(x)$ to the center of $A$. We have, from the upward-propagation phase of the computation, the multipole expansion $\phi_{A'}(x)$ for each square $A'$ in $A$'s
interactive-field. In order to combine $\phi_{A'}(x)$, we shift them to the center of $A$, and flip regions of convergence so that they have a common reference point and are valid inside the square $A$. The sum of the shifted $\phi'_{A'}(x)$ and $\phi_{A'}(x)$ is the local expansion $\phi_A(x)$ for the square $A$. This computation is performed recursively all the way down to the leaves of the tree.

Now we have for every square at the finest decomposition level the local expansion due to its far-field valid in the square. To get the far-field potential at a given particle, we have only to evaluate the local expansion at the particle’s position. The near-field potential is obtained by evaluating directly interactions with particles in the near-field. The sum of the near-field potential and the far-field potential is the desired potential at the particle.

Rokhlin and Greengard [GR86] show that the total work required is $N(ap^2 + bp + ck_n + d)$, where $k_n$ is the upper-bound for the number of particles in a square at the finest level. The key observation in this estimate is that each shifting or flipping of a $p$-term multipole expansion takes $O(p^2)$ work, and that the number of squares in a given square’s interactive-field is bounded by 27. The total number of squares at all levels is

$$4^0 + 4^1 + \ldots + 4^n = \frac{4^{n+1} - 1}{3} = \frac{4N - 1}{3}.$$ 

Therefore the upward-propagation phase and the downward-propagation phase of the algorithm accounts for the $p^2$ term in the estimate. The initial expansion step and the final evaluation step in the algorithm is responsible for the $p$ term, and the direct computation on near-field potentials is for the $k_n$ term.

The overall computational complexity of the algorithm, from the above estimate, is of order $O(N)$.

Rokhlin and Greengard [GR86] also derive that the error estimate in the computation is $C(1/2)^{p+1}$. Given a precision requirement $\epsilon$, the number of terms $p$ retained in the expansion is set as $\lceil -\log_2 \epsilon \rceil$.

2.3 Remarks

The key to the two-dimensional $O(N)$ algorithm is that, for the potential $\phi(x)$ due to charges at $\{y^{(i)}, i = 1, \ldots, m\}$, we evaluate the potential at each of the points $\{x^{(j)}, j = 1, \ldots, n\}$ by applying the multipole expansion of $\phi(x)$, rather
than by directly evaluating and summing the individual potentials \( \phi_{x^{(i)}}(x) \) at each \( x^{(j)} \). Our ability to apply this method relies on the fact that the Taylor series expansions of the \( \phi_{x^{(i)}}(x) \{i = 1, \ldots, m\} \) are absolutely convergent power series whose coefficients are independent of \( x \). We produce the expansion of \( \phi(x) \) by summing corresponding coefficients of the expansions \( \phi_{x^{(i)}}(x) \). In order to do so, these expansions must have a common reference point and a common region of validity, or they can be shifted so that they do so.

The applicability of complex-analytic techniques in the two-dimensional case results from the fact that the two-dimensional potential, viewed as a real-valued function on \( \mathbb{R}^2 \), is harmonic, and thus is the real part of a complex-analytic function on \( \mathbb{C} \). In three dimensions, the potential function is likewise harmonic. Unfortunately, there are no corresponding complex-analytic techniques in three dimensions to simplify the analysis.

In the next chapter, we will produce expansions of the three-dimensional potential that are suitable to support the multipole expansion method.
Chapter 3

A three-dimensional $N$-body algorithm

We consider now the three-dimensional $N$-body problem, where the gravitational potential at a point $x \in \mathbb{R}^3$ due to a point mass $\mu$ located at $y \in \mathbb{R}^3$ is

$$\phi_y(x) = -\frac{\mu}{\|x - y\|}.$$  \hspace{1cm} (3.1)

Coulombic potential due to a point charge $q$ has the same formula, except that $\mu$ is replaced by $-q$ [Gol59] [Arn78].

In order to apply the multipole expansion method, we must expand this potential as an absolutely convergent series whose coefficients are independent of $x$, and we must produce an a priori bound on the error when we retain only a finite number of terms in the series. We must also be able to shift the series expansion to a new origin.

The potential function in (3.1) is harmonic in everywhere other than $y$ [Kos64]. This suggests the possibility of expanding $\phi_y(x)$ in terms of spherical harmonics [Hob55].

In this chapter we prove two theorems that derive the required expansions and give bounds on the error terms. There are also three lemmas that describe how to shift these expansions from one origin to another. Using these analytical results, we present an analogue of Rokhlin and Greengard’s algorithm in three dimensions.

At the same time the author derived the results [Zha87], Greengard also
extended the two-dimensional algorithm to three dimensions [Gre87]. His
method is similar to the one given here, but the underlying theorems and ex-
pansions are different. Greengard works in terms of polar coordinates, while
the formulas and derivations here are done in terms of Cartesian coordinates.

In Chapter 4, we present experimental results that demonstrate the er-
ror bound and order-of-growth estimates given here. However, there are no
comparable published experimental results for the polar-coordinate form of
the algorithm. It is unknown how the two different forms of the algorithm
compare in practical implementations.

3.1 The Multipole Expansion in Three Di-

mensions

Given an ensemble of particle masses \{\mu^{(n)}, n = 1, ..., m\} located at points
\{y^{(n)} \in \mathbb{R}^3, n = 1, ..., m\}, the gravitational potential at any given point
x \in \mathbb{R}^3 is

$$\Phi(x) = \sum_{n=1}^{m} \frac{-\mu^{(n)}}{\|x - y^{(n)}\|} = \sum_{n=1}^{m} \frac{-\mu^{(n)}}{r^{(n)}}.$$  (3.2)

Here \(r^{(n)} = \|x - y^{(n)}\|\). We will show how to expand \(1/r^{(n)}\) into an
absolutely convergent series

$$\frac{1}{r^{(n)}} = \sum_{ijk} a_{ijk}(y^{(n)}) \Theta_{ijk}(x), \quad n = 1, ..., m,$$  (3.3)

where \(a_{ijk}(y^{(n)})\) is independent of \(x\) and depends only on \(y^{(n)}\). This will
enable us to get the multipole expansion for the potential \(\Phi(x)\) due to all the
masses \(\{\mu^{(n)}\}\) by summing together the \(a_{ijk}(y^{(n)})\)'s weighted by the masses
\(\mu^{(1)}, \mu^{(2)}, ..., \mu^{(m)}\) respectively:

$$\Phi(x) = \sum_{ijk} \left( \sum_{n=1}^{m} \left(-\mu^{(n)}\right) a_{ijk}(y^{(n)}) \right) \Theta_{ijk}(x).$$  (3.4)

The force field is obtained by taking the gradient of the potential field

$$\overline{F} = -\nabla \Phi(x)$$
\[ = \sum_{ijk} \left( \sum_{n=1}^{m} \left(-\mu^{(n)}\right) a_{ijk}(y^{(n)}) \right) (-\nabla \Theta_{ijk}(x)) \]
\[ = \sum_{ijk} \left( \sum_{n=1}^{m} \left(-\mu^{(n)}\right) a_{ijk}(y^{(n)}) \right) \Theta'_{ijk}(x). \tag{3.5} \]

The series expansion of (3.5) has the same coefficients as that of (3.4), only this time, \( \Theta_{ijk}(x) \) is replaced by \( \Theta'_{ijk}(x) \). This enables us to use the same expansion formulas for potentials derived in the next section to compute multipole expansions for forces.

### 3.2 Theorems

This section derives necessary formulas for producing and shifting multipole expansions in three dimensions, together with bounds on the error when retaining only a finite number of terms in the expansions.

![Figure 3.1: Gravitational field](image)

Figure 3.1 illustrates the quantities needed in our derivation. The potential at point \( x \in R^3 \) is due to mass \( \mu \) at point \( y \in R^3 \). \( y^{(0)} \) is the reference point, and \( \gamma \) is the angle between vectors \( y^{(0)}y \) and \( y^{(0)}x \). Also there are
quantities \( r = \| x - y \|, r_0 = \| y - y^{(0)} \|, \) and \( R = \| x - y^{(0)} \|. \) We need the following relationships later in deriving the multipole expansion formula.

From the basic triangle relationship, we have

\[
    r = \sqrt{R^2 + r_0^2 - 2Rr_0 \cos \gamma}. \tag{3.6}
\]

We derive the following theorem which produces an expansion for \( 1/r \) in a region of analyticity outside a sphere. This is essential in obtaining the multipole expansion (3.4).

**Theorem 3.2.1** Using the quantities as shown in Figure 3.1, if \( R > r_0, \) that is, if \( x = (x_1, x_2, x_3) \) measured relative to \( y^{(0)} \) is outside the sphere of radius \( r_0 \) centered at \( y^{(0)} \), then

\[
    \frac{1}{r} = \sum_{i,j,k=0}^{\infty} a_{ijk} \frac{\partial^{i+j+k}}{\partial x_1^i \partial x_2^j \partial x_3^k} \left( \frac{1}{r} \right) \tag{3.7}
\]

where

\[
    a_{ijk} = (-1)^{i+j+k} \frac{r_0^{i+j+k}}{i!j!k!} \nu_1^{i} \nu_2^{j} \nu_3^{k}, \tag{3.8}
\]

and \( \nu_1, \nu_2, \nu_3 \) are the direction cosines of \( y^{(0)} y \).

If we retain only terms of \( i + j + k \leq p \) then the error satisfies

\[
    |\epsilon| \leq C \left| \frac{r_0}{R} \right|^{p+1} \tag{3.9}
\]

where \( C \) is independent of \( p \).

**Proof.**

In Figure 3.1, the Cartesian coordinate distance between the points \( x \) and \( y \) is

\[
    r(x, y) = \| x - y \| = \sqrt{\sum_{\beta=1}^{3} \left[ (x_\beta - y^{(0)}_\beta) - (y_\beta - y^{(0)}_\beta) \right]^2}. \tag{3.10}
\]
CHAPTER 3. A 3-D N-BODY ALGORITHM

We expand $1/r(x, y)$ as a Taylor series in powers of $(y_\beta - y^{(0)}_\beta)$ and get

$$\frac{1}{r(x, y)} = \frac{1}{r(x, y)}_{y = y^{(0)}} + \sum_{\alpha = 1}^{\infty} \frac{1}{\alpha!} \left\{ \left[ \sum_{\beta = 1}^{3} (y_\beta - y^{(0)}_\beta) \frac{\partial}{\partial y_\beta} \right]^{\alpha} \frac{1}{r(x, y)} \right\}_{y = y^{(0)}}. \quad (3.11)$$

The subscript $y = y^{(0)}$ following the brace indicates that after the differentiation is done $y_1, y_2, y_3$ of point $y$ should be replaced by $y_{1}^{(0)}, y_{2}^{(0)}, y_{3}^{(0)}$ of point $y^{(0)}$. When $x$ lies outside the sphere of radius $r_0$ centered at $y^{(0)}$, the above Taylor series converges absolutely and uniformly. We make the substitution $y = y^{(0)}$ before the differentiation is performed. Using the identities

$$\frac{\partial}{\partial y_\beta} \left( \frac{1}{r(x, y)} \right) = -\frac{\partial}{\partial x_\beta} \left( \frac{1}{r(x, y)} \right) \quad \beta = 1, 2, 3$$

and

$$\frac{1}{r(x, y)}_{y = y^{(0)}} = \frac{1}{R}$$

the Taylor series (3.11) becomes

$$\frac{1}{r(x, y)} = \frac{1}{R} + \sum_{\alpha = 1}^{\infty} \frac{(-1)^\alpha}{\alpha!} r_0^\alpha \left( \sum_{\beta = 1}^{3} \frac{y_\beta - y^{(0)}_\beta}{r_0} \frac{\partial}{\partial x_\beta} \right)^\alpha \left( \frac{1}{R} \right). \quad (3.12)$$

Let $\nu_\beta = (y_\beta - y^{(0)}_\beta) / r_0$. Since $\nu_1, \nu_2, \nu_3$ are the direction cosines of $y^{(0)}y$, it follows that

$$\left( \sum_{\beta = 1}^{3} \nu_\beta \frac{\partial}{\partial x_\beta} \right)^\alpha = \sum_{i+j+k = \alpha} \frac{\alpha!}{i!j!k!} \nu_1^i \nu_2^j \nu_3^k \frac{\partial^\alpha}{\partial x_1^i \partial x_2^j \partial x_3^k}. \quad (3.13)$$

Substitute this into (3.12), we get

$$\frac{1}{r(x, y)} = \sum_{i,j,k = 0}^{\infty} (-1)^{i+j+k} \frac{r_0^{i+j+k}}{i!j!k!} \nu_1^i \nu_2^j \nu_3^k \frac{\partial^{i+j+k}}{\partial x_1^i \partial x_2^j \partial x_3^k} \left( \frac{1}{R} \right). \quad (3.14)$$

This completes the proof for (3.7).

In order to derive the error bound of (3.9), we will use the fact that $1/r$ can be expanded in powers of $R$ using the Legendre Polynomials [And85].
The polynomials $P_\alpha(x)$ are called the Legendre Polynomials

$$P_0(x) = 1, \quad P_1(x) = x, \quad P_2(x) = \frac{1}{2}(3x^2 - 1), \ldots$$

The series of (3.15) is absolutely convergent. Notice that the Legendre Polynomials $P_\alpha(\cos \gamma)$ vary with the directions of the vectors $y^{(0)}x$ and $y^{(0)}y$, and thus is a function of the point $x$ at which the potential is to be evaluated, as well as a function of the mass location $y$.

Now we can complete the proof of (3.9). Using the notation

$$\frac{\partial^\alpha}{\partial r_0^\alpha} = \left( \sum_{\beta=1}^{\nu} \nu_{\beta} \frac{\partial}{\partial x_{\beta}} \right)^\alpha$$

the series of (3.12) becomes

$$\frac{1}{r(x,y)} = \frac{1}{R} + \sum_{\alpha=1}^{\infty} (-1)^\alpha \frac{\partial^\alpha}{\alpha! \partial r_0^\alpha} \left( \frac{1}{R} \right). \quad (3.16)$$

Notice that there is a term-by-term correspondence between this series and the series of (3.15). By equating terms, we have

$$P_\alpha(\cos \gamma) = (-1)^\alpha \frac{R^{\alpha+1}}{\alpha!} \frac{\partial^\alpha}{\partial r_0^\alpha} \left( \frac{1}{R} \right). \quad (3.17)$$

Surprisingly, $\frac{R^{\alpha+1}}{\alpha!} \frac{\partial^\alpha}{\partial r_0^\alpha} \left( \frac{1}{R} \right)$ only depends on $\gamma$, not $R$.

Using the correspondence between (3.16) and (3.15), we have

$$|\epsilon| = \left| \frac{1}{r} - \sum_{i,j,k} a_{ijk} \frac{\partial^{i+j+k}}{\partial x_1^i \partial x_2^j \partial x_3^k} \left( \frac{1}{R} \right) \right|$$
\[ = \left| \frac{1}{r} - \sum_{\alpha \leq p} \frac{(-1)^\alpha}{\alpha!} r_0^\alpha \frac{\partial^\alpha}{\partial r_0^\alpha} \left( \frac{1}{R} \right) \right| \\
= \left| \sum_{\alpha = p+1}^\infty \frac{(-1)^\alpha}{\alpha!} r_0^\alpha \frac{\partial^\alpha}{\partial r_0^\alpha} \left( \frac{1}{R} \right) \right| \\
= \left| \sum_{\alpha = p+1}^\infty \frac{r_0^\alpha}{R^{\alpha+1}} P_\alpha(\cos \theta) \right| \\
\]

Since the Legendre polynomials have the property of \(|P_\alpha(\cos \gamma)| \leq 1\), the error therefore is
\[
|e| \leq \sum_{\alpha = p+1}^\infty \left| \frac{r_0^\alpha}{R^{\alpha+1}} \right| = \frac{1}{R - r_0} \left| \frac{r_0}{R} \right|^{p+1} \\
= C \left| \frac{r_0}{R} \right|^{p+1}.
\]

This completes the proof.

Using the results derived in Theorem 3.2.1, we can produce the multipole expansion of (3.4) in Section 3.1. As in Section 3.1, suppose that all \(m\) particle masses are located within a spherical region with radius \(r_0\) centered at \(y^{(0)}\), that is, \(\|y^{(n)} - y^{(0)}\| < r_0\), \(n = 1, ..., m\), we have from Theorem 3.2.1 the series expansion of \(1/r^{(n)}\) converging for any \(x\) with \(\|x - y^{(0)}\| > r_0\)
\[
\frac{1}{r^{(n)}} = \sum_{i,j,k=0}^\infty a_{ijk} \frac{\partial^{i+j+k}}{\partial x_1^i \partial x_2^j \partial x_3^k} \left( \frac{1}{\|x - y^{(0)}\|} \right),
\]
and the error estimate
\[
\left| \frac{1}{r^{(n)}} - \sum_{i,j,k} a_{ijk} \frac{\partial^{i+j+k}}{\partial x_1^i \partial x_2^j \partial x_3^k} \left( \frac{1}{\|x - y^{(0)}\|} \right) \right| \leq C \left| \frac{r_0}{\|x - y^{(0)}\|} \right|^{p+1},
\]
\(n = 1, ..., m\).

The multipole expansion of (3.4), therefore, is
\[
\Phi(x) = \sum_{ijk} \alpha_{ijk} \frac{\partial^{i+j+k}}{\partial x_1^i \partial x_2^j \partial x_3^k} \left( \frac{1}{\|x - y^{(0)}\|} \right),
\] (3.18)
where
\[ \alpha_{ijk} = \sum_{n=1}^{m} (-\mu^{(n)} a_{ijk}). \]

Furthermore,
\[ \left| \Phi(x) - \sum_{i,j,k}^{i+j+k \leq p} \alpha_{ijk} \frac{\partial^{i+j+k}}{\partial x_1^i \partial x_2^j \partial x_3^k} \left( \frac{1}{\|x - y^{(0)}\|} \right) \right| \leq C' \left| \frac{r_0}{\|x - y^{(0)}\|} \right|^{p+1}, \]
where
\[ C' = \sum_{n=1}^{m} C \mu^{(n)}. \]

The following theorem produces an expansion for $1/r$ in a spherical region of analyticity, which is the basis for flipping the region of analyticity in shiftings, as required by the fast algorithm.

**Theorem 3.2.2** Using the quantities in Figure 3.1 again, if $R < r_0$, that is, if $x = (x_1, x_2, x_3)$ measured relative to $y^{(0)}$ lies inside the sphere of radius $r_0$ centered at $y^{(0)}$, then
\[ \frac{1}{r} = \sum_{i,j,k=0}^{\infty} b_{ijk} x_1^i x_2^j x_3^k \]
where
\[ b_{ijk} = \frac{(-1)^{i+j+k}}{r_0^{i+j+k+1}} \sum_{\gamma=0}^{\frac{i}{2}} \sum_{\beta=0}^{\frac{j}{2}} \sum_{\alpha=0}^{\frac{k}{2}} AB \]
\[ A = \begin{pmatrix} i + j + k - \alpha - \beta - \gamma \\ i + j + k - \alpha - \beta - \gamma \end{pmatrix} \begin{pmatrix} i + j + k - \alpha - \beta - \gamma \\ i - \alpha \\ (i - \alpha)(j - \beta)(k - \gamma) \end{pmatrix} \begin{pmatrix} (2\nu_1)^{i-2\alpha}(2\nu_2)^{j-2\beta}(2\nu_3)^{k-2\gamma} \end{pmatrix} \]
\[ B = \begin{pmatrix} i - \alpha \\ \beta \\ (i - \alpha)(j - \beta)(k - \gamma) \end{pmatrix} \begin{pmatrix} (2\nu_1)^{i-2\alpha}(2\nu_2)^{j-2\beta}(2\nu_3)^{k-2\gamma} \end{pmatrix} \]
here $\nu_1, \nu_2, \nu_3$ are the direction cosines of $y^{(0)} y$.

The error bound for truncating terms of $i + j + k > p$ satisfies
\[ |\epsilon| \leq C \left| \frac{R}{r_0} \right|^{p+1} \]
CHAPTER 3. A 3-D N-BODY ALGORITHM

Proof.
Using the relationship in (3.6), we have

$$\frac{1}{r} = \frac{1}{\sqrt{R^2 + r_0^2 - 2Rr_0 \cos \gamma}}$$

$$= \frac{1}{r_0 \sqrt{(\frac{R}{r_0})^2 + 1 - 2\frac{R}{r_0} \cos \gamma}}. \quad (3.23)$$

Denote by $\tau_1, \tau_2, \tau_3$ the direction cosines of $\vec{y}^{(0)}x$, and $\nu_1, \nu_2, \nu_3$ the direction cosines of $\vec{y}^{(0)}y$. Then

$$\cos \gamma = \tau_1 \nu_1 + \tau_2 \nu_2 + \tau_3 \nu_3$$

where $\gamma$ is the angle between $\vec{y}^{(0)}x$ and $\vec{y}^{(0)}y$. This leads to the equality

$$\left(\frac{R}{r_0}\right)^2 - 2\frac{R}{r_0} \cos \gamma = \frac{x_1^2 + x_2^2 + x_3^2}{r_0^2} - 2\frac{x_1 \nu_1 + x_2 \nu_2 + x_3 \nu_3}{r_0}.$$

If we introduce the variables

$$t_1 = \frac{x_1}{r_0}, t_2 = \frac{x_2}{r_0}, t_3 = \frac{x_3}{r_0}, \text{ and }$$

$$\alpha = t_1(t_1 - 2\nu_1), \beta = t_2(t_2 - 2\nu_2), \gamma = t_3(t_3 - 2\nu_3), \quad (3.24)$$

equation (3.23) becomes

$$\frac{1}{r} = \frac{1}{r_0} \frac{1}{\sqrt{1 + (\alpha + \beta + \gamma)}}.$$

We can expand this as a Taylor series

$$\frac{1}{r} = \frac{1}{r_0} \sum_{n=0}^{\infty} \left(-\frac{1}{2}\right)(\alpha + \beta + \gamma)^n, \quad (3.25)$$

that is valid for $|\alpha + \beta + \gamma| < 1$. But

$$(\alpha + \beta + \gamma)^n = \sum_{i+j+k=n} \frac{n!}{i!j!k!} \alpha^i \beta^j \gamma^k,$$
thus
\[
\frac{1}{r} = \frac{1}{r_0} \sum_{i,j,k=0}^{\infty} \binom{-\frac{1}{2}}{i+j+k} \frac{(i+j+k)!}{i!j!k!} \alpha^i \beta^j \gamma^k
\]
\[
= \frac{1}{r_0} \sum_{i,j,k=0}^{\infty} b'_{ijk} \alpha^i \beta^j \gamma^k
\]  
(3.26)

where
\[
b'_{ijk} = \binom{-\frac{1}{2}}{i+j+k} \frac{(i+j+k)!}{i!j!k!}.
\]

Substitute \(\alpha, \beta, \gamma\) with (3.24), the Taylor series of (3.26) becomes
\[
\frac{1}{r} = \frac{1}{r_0} \sum_{i,j,k=0}^{\infty} b''_{ijk} \left( \frac{x_1}{r_0} \right)^i \left( \frac{x_2}{r_0} \right)^j \left( \frac{x_3}{r_0} \right)^k
\]  
(3.27)

where
\[
b''_{ijk} = (-1)^{i+j+k} \sum_{\tau=0}^{\frac{1}{3}} \sum_{\beta=0}^{\frac{1}{3}} \sum_{\alpha=0}^{\frac{1}{3}} b'_{(i-\alpha),(j-\beta),(k-\gamma)}
\]
\[
\binom{i-\alpha}{\alpha} \binom{j-\beta}{\beta} \binom{k-\gamma}{\gamma} (2\nu_1)^{i-2\alpha} (2\nu_2)^{j-2\beta} (2\nu_3)^{k-2\gamma}.
\]

Write (3.27) explicitly in powers of \(x_1, x_2, \) and \(x_3\)
\[
\frac{1}{r} = \sum_{i,j,k=0}^{\infty} b_{ijk} x_1^i x_2^j x_3^k
\]

where
\[
b_{ijk} = \frac{b''_{ijk}}{r_0^{i+j+k+1}}.
\]

As in Theorem 3.2.1, the error bound for truncating terms of \(i+j+k > p\) in (3.20) satisfies
\[
|\epsilon| \leq C \left| \frac{R}{r_0} \right|^{p+1}.
\]

In the following lemma, we produce a formula which shifts the center of a multipole expansion valid in a region of analyticity outside a sphere.
Lemma 3.2.1 Given a multipole expansion

$$
\Phi(x) = \sum_{\alpha, \beta, \gamma=0}^{\infty} a_{\alpha, \beta, \gamma} \frac{\partial^{\alpha+\beta+\gamma}}{\partial x_1^\alpha \partial x_2^\beta \partial x_3^\gamma} \left( \frac{1}{R} \right), \tag{3.28}
$$

with respect to $y^{(0)}$, valid in the region outside the sphere of radius $r_0$ centered at $y^{(0)}$, suppose we shift the reference point $y^{(0)}$ to $y''^{(0)}$ with $\| y^{(0)} - y''^{(0)} \| = \Delta r$, as shown in Figure 3.2. Note that $R' = \| x - y''^{(0)} \|$. The new multipole expansion of (3.28), with respect to $y''^{(0)}$, in the region outside the sphere of radius $r_0 + \Delta r$ centered at $y''^{(0)}$ is

$$
\Phi(X) = \sum_{i, j, k=0}^{\infty} c_{ijk} \frac{\partial^{i+j+k}}{\partial X_1^i \partial X_2^j \partial X_3^k} \left( \frac{1}{R'} \right), \tag{3.29}
$$

where

$$
c_{ijk} = \sum_{\alpha=0}^{i} \sum_{\beta=0}^{j} \sum_{\gamma=0}^{k} a_{\alpha, \beta, \gamma} a'_i \cdot a'_j \cdot a'_k. \tag{3.30}
$$

Here $X = (X_1, X_2, X_3)$ are the Cartesian coordinates of the point $x$ relative to $y''^{(0)}$, and $a'_{ijk}$ are the coefficients in the expansion for $1/R$ with respect to $y''^{(0)}$. 
The error bound for retaining terms of \( i + j + k \leq p \) in (3.29) satisfies

\[
|e| \leq C \left| \frac{r_0 + \Delta r}{R'} \right|^{p+1}.
\] (3.31)

**Proof.**

Using Theorem 3.2.1, we expand \( 1/R \), with respect to \( y''(0) \), in the region outside the sphere of radius \( \Delta r \) centered at \( y''(0) \) as

\[
\frac{1}{R} = \sum_{i,j,k=0}^{\infty} a'_{ijk} \frac{\partial^{i+j+k}}{\partial X_1^i \partial X_2^j \partial X_3^k} \left( \frac{1}{R'} \right),
\] (3.32)

where

\[
a'_{ijk} = (-1)^{i+j+k} \frac{\Delta r^{i+j+k}}{i!j!k!} \tau_1^i \tau_2^j \tau_3^k.
\]

\( \tau_1, \tau_2, \tau_3 \) are direction cosines of \( y''(0) \). Notice that after the shifting there is

\[
X_i - z_i = y''(0) - y''(0) \quad i = 1, 2, 3
\]

Therefore

\[
\frac{\partial}{\partial x_i} = \frac{\partial}{\partial X_i}.
\]

Substitute (3.32) into (3.28), we have

\[
\Phi(x) = \sum_{\alpha, \beta, \gamma = 0}^{\infty} a_{\alpha, \beta, \gamma} \frac{\partial^{\alpha+\beta+\gamma}}{\partial x_1^\alpha \partial x_2^\beta \partial x_3^\gamma} \left[ \sum_{i,j,k=0}^{\infty} a'_{ijk} \frac{\partial^{i+j+k}}{\partial X_1^i \partial X_2^j \partial X_3^k} \left( \frac{1}{R'} \right) \right],
\]

that is

\[
\Phi(X) = \sum_{\alpha, \beta, \gamma = 0}^{\infty} \sum_{i,j,k=0}^{\infty} a_{\alpha, \beta, \gamma} a'_{ijk} \frac{\partial^{\alpha+\beta+\gamma+i+j+k}}{\partial X_1^\alpha \partial X_2^\beta \partial X_3^\gamma} \left( \frac{1}{R'} \right),
\]

which is valid outside the sphere of radius \( r_0 + \Delta r \) centered at \( y''(0) \). Make substitutions of \( \alpha + i = i', \beta + j = j', \gamma + k = k' \), we get

\[
\Phi(X) = \sum_{i', j', k'=0}^{\infty} c_{i', j', k'} \frac{\partial^{i'+j'+k'}}{\partial X_1^{i'} \partial X_2^{j'} \partial X_3^{k'}} \left( \frac{1}{R'} \right).
\]
where
\[ c_{j'k'}^{i} = \sum_{\alpha=0}^{i} \sum_{\beta=0}^{j} \sum_{\gamma=0}^{k} a_{\alpha \beta \gamma} a_{i' - \alpha, j' - \beta, k' - \gamma}. \]

So far we have proved the expansion (3.29).

Since the multipole expansion of (3.29) is unique in terms of the derivatives of \(1/R'\), we could estimate the error bound for truncation in (3.29) as if we expanded it directly with respect to \(y^{(0)}\). From (3.19) we know that is

\[ |\epsilon| \leq C' \left| \frac{r_0 + \Delta r}{R'} \right|^{p+1}. \]

We now show how to convert a multipole expansion into a local expansion valid within a spherical region of analyticity, by shifting the reference point. The region of analyticity is flipped.

![Figure 3.3: Lemma 3.2.2](image)

**Lemma 3.2.2** Given a multipole expansion

\[ \Phi(x) = \sum_{\alpha, \beta, \gamma=0}^{\infty} a_{\alpha \beta \gamma} \frac{\partial^{\alpha + \beta + \gamma}}{\partial x_1^\alpha \partial x_2^\beta \partial x_3^\gamma} \left( \frac{1}{R} \right), \quad (3.33) \]
with respect to \( y^{(0)} \), valid in the region outside the sphere of radius \( r_0 \) centered at \( y^{(0)} \), suppose this time the reference point \( y^{(0)} \) is shifted \( \Delta r \) to \( y^{(0)} \) with \( \Delta r > r_0 \), as shown in Figure 3.3. The new multipole expansion of (3.33), with respect to \( y^{(0)} \), this time inside a sphere of radius \( (\Delta r - r_0) \) centered at \( y^{(0)} \) is

\[
\Phi(X) = \sum_{ijk}^{\infty} c_{ijk} X_i^i X_j^j X_k^k
\]  

(3.34)

where

\[
c_{ijk} = \frac{1}{i!j!k!} \sum_{\alpha, \beta, \gamma=0}^{\infty} a_{\alpha \beta \gamma} b_{i+\alpha, j+\beta, k+\gamma} (i+\alpha)!(j+\beta)!(k+\gamma)!
\]  

(3.35)

Again \( X = (X_1, X_2, X_3) \) are the coordinates of the point \( x \) relative to \( y^{(0)} \).

\( b_{ijk} \) are the coefficients in the expansions for \( 1/R \) with respect to \( y^{(0)} \).

The error bound for retaining terms of \( i + j + k \leq p \) in (3.34) is

\[
|e| \leq C \left| \frac{R'}{\Delta r - r_0} \right|^{p+1}.
\]  

(3.36)

Here \( R' = \| x - y^{(0)} \| \).

Proof.

From Theorem 3.2.2 we can expand \( 1/R \), with respect to \( y^{(0)} \), inside a spherical region with radius \( (\Delta r - r_0) \) and center \( y^{(0)} \) as

\[
\frac{1}{R} = \sum_{i,j,k=0}^{\infty} b_{ijk} X_i^i X_j^j X_k^k
\]  

(3.37)

where

\[
b_{ijk} = \frac{(-1)^{i+j+k}}{r_0^{i+j+k+1}} \left( \sum_{\gamma=0}^{i} \sum_{\beta=0}^{j} \sum_{\alpha=0}^{k} AB \right)
\]  

(3.38)

\[
A = \left( \begin{array}{c} \frac{-1}{2} \\ i+j+k-\alpha-\beta-\gamma \\ i+j+k-\alpha-\beta-\gamma \end{array} \right) (i+j+k-\alpha-\beta-\gamma)!
\]  

\[
B = \left( \begin{array}{c} i-\alpha \\ j-\beta \\ k-\gamma \end{array} \right) (2\tau_1)^{i-2\alpha}(2\tau_2)^{j-2\beta}(2\tau_3)^{k-2\gamma}
\]
\( \tau_1, \tau_2, \tau_3 \) are direction cosines of \( \overrightarrow{y^{(0)}y^{(0)}} \). Substitute (3.37) into (3.33)

\[
\Phi(X) = \sum_{\alpha, \beta, \gamma = 0}^{\infty} a_{\alpha \beta \gamma} \frac{\partial^{\alpha + \beta + \gamma}}{\partial x_1^\alpha \partial x_2^\beta \partial x_3^\gamma} \left( \sum_{i,j,k = 0}^{\infty} b_{ijk} X_1^i X_2^j X_3^k \right)
\]

\[
= \sum_{\alpha, \beta, \gamma = 0}^{\infty} a_{\alpha \beta \gamma} \sum_{i=\alpha}^{\infty} \sum_{j=\beta}^{\infty} \sum_{k=\gamma}^{\infty} b_{ijk} iljk!(i-\alpha)!(j-\beta)!(k-\gamma)! X_1^{i-\alpha} X_2^{j-\beta} X_3^{k-\gamma}.
\]  

(3.39)

The region of analyticity is the inside of the sphere with radius \((\Delta r - r_0)\) and center at \(y^{(0)}\). After making substitutions of \(i-\alpha = i', j-\beta = j', k-\gamma = k'\), we get

\[
\Phi(X) = \sum_{i', j', k' = 0}^{\infty} c_{i'j'k'} X_1^{i'} X_2^{j'} X_3^{k'}
\]  

(3.40)

where

\[
c_{i'j'k'} = \frac{1}{i'!j'!k'!} \sum_{\alpha, \beta, \gamma = 0}^{\infty} a_{\alpha \beta \gamma} b_{\alpha + \alpha, \beta + \beta, \gamma + \gamma}(i' + \alpha)(j' + \beta)(k' + \gamma)!
\]  

(3.41)

This completes the proof of (3.34).

Similar to Lemma 3.2.1, the error bound for retaining only terms of \(i + j + k \leq p\) in (3.34) is

\[
|\epsilon| \leq C' \left| \frac{R'}{\Delta r - r_0} \right|^{p+1}
\]

We also derive a formula for shifting an local expansion within a spherical region of analyticity.

**Lemma 3.2.3** Given a local expansion

\[
\Phi(x) = \sum_{i,j,k = 0}^{\infty} c_{ijk} x_1^i x_2^j x_3^k,
\]

with respect to \(y^{(0)}\), valid in the region inside the sphere of radius \(r_0\) centered at \(y^{(0)}\), we expand \(\Phi(x)\) inside the sphere of radius \(r_0 - \Delta r\) centered at \(y^{(0)}\),
CHAPTER 3. A 3-D N-BODY ALGORITHM

Figure 3.4: Lemma 3.2.3

after $y^{(0)}$ is shifted $\Delta r$ to $y'^{(0)}$ with $\Delta r < r_0$ as shown in Figure 3.4, as

$$
\Phi(X) = \sum_{i,j,k=0}^{\infty} d_{ijk} X_1^i X_2^j X_3^k
$$

with $X = (X_1, X_2, X_3)$ the new coordinates of the point $x$ with respect to $y'^{(0)}$, and

$$
d_{ijk} = \sum_{\alpha, \beta, \gamma=0}^{\infty} c_{i+\alpha, j+\beta, k+\gamma} \binom{i + \alpha}{\alpha} \binom{j + \beta}{\beta} \binom{k + \gamma}{\gamma} \Delta x_1^\alpha \Delta x_2^\beta \Delta x_3^\gamma \tag{3.42}
$$

Here $y''(0) - y'(0) = (\Delta x_1, \Delta x_2, \Delta x_3)$.

Proof.

This can be derived by re-expanding the expansion

$$
\Phi(X) = \sum_{i,j,k=0}^{\infty} c_{ijk} (X_1 + \Delta x_1)^i (X_2 + \Delta x_2)^j (X_3 + \Delta x_3)^k
$$

in powers of $X_1, X_2, X_3$. No truncation errors are introduced in the calculation of (3.42).
CHAPTER 3. A 3-D N-BODY ALGORITHM

Having derived the potential in the form of the multipole expansion

\[ \Phi(x) = \sum_{i,j,k=0}^{\infty} c_{ijk} x_1^i x_2^j x_3^k, \]

we recall from the equation (3.5) of Section 3.1 that the force is obtained by

\[ \mathbf{F} = -\nabla \Phi(x), \]

where

\[ \nabla = \left( \frac{\partial}{\partial x_1}, \frac{\partial}{\partial x_2}, \frac{\partial}{\partial x_3} \right). \]

Therefore,

\[ \mathbf{F} = \left( -\sum_{i,j,k=0}^{\infty} i c_{ijk} x_1^{i-1} x_2^j x_3^k, -\sum_{i,j,k=0}^{\infty} j c_{ijk} x_1^i x_2^{j-1} x_3^k, -\sum_{i,j,k=0}^{\infty} k c_{ijk} x_1^i x_2^j x_3^{k-1} \right). \]

3.3 A Sequential Algorithm

In this section we give a complete description of the three-dimensional \( N \)-body algorithm. This algorithm is similar to the two-dimensional algorithm of Rokhlin and Greengard described in Chapter 2, however this time, we use the analytical results obtained in the previous section. This formulation of the algorithm is for a sequential computer. We present a parallel version in Chapter 5.

The three-dimensional space under consideration is taken to be a cube (Figure 3.5). We apply the operation of subdividing a cube into eight identical subcubes repeatedly, until a subcube has only a few particles in it. When the particle distribution is relatively uniform, the level of subdivision is approximately \( n = \lfloor \log_8 N \rfloor \). By convention, we say that the initial cube is at level 0, and the atomic cubes, i.e., cubes at the finest level of the spatial refinement, are at level \( n \). As in the two-dimensional algorithm, we define for a cube its near-field, far-field and interactive-field. The near-field of a cube is defined as consisting of those cubes that are at the same level as the cube, and have distance to the center of the cube less than \( \sqrt{3} \) of the cube edge length; the far-field of the cube is the complement of its near-field and itself; and the interactive-field of the cube is the part of its far-field contained in its parent’s near-field.
Figure 3.5: Decomposition in a three-dimensional space

For a cube $i$ at level $l$, we denote by $\phi_i^l(x)$ the multipole expansion, valid in the far-field of the cube, for the potential due to particles in the cube; and by $\psi_i^l(x)$ the local expansion, valid inside the cube, for the potential due to those particles in its far-field. Both $\phi_i^l(x)$ and $\psi_i^l(x)$ are expanded relative to the center of the cube.

A description of the algorithm is given in Figure 3.6. $n$ is the total levels of spatial refinement.

Step (1) computes initially the multipole expansion $\phi_i^0(x)$ for each atomic cube. This is obtained by adding together series expansions, relative to the center of the cube, for potentials due to individual particles in the cube.

Step (2) computes the multipole expansion $\phi(x)$ for each of the cubes at all intermediate levels in an upward manner. For a cube $i$ at level $l$, the multipole expansion $\phi_i^l(x)$ is computed by summing $\phi_i^{l+1}(x)$ of cube $i$'s eight subcubes that are shifted to the center of the cube $i$.

Step (3) computes the local expansion $\psi(x)$ for each of the cubes. For a cube $i$ at level $l$, the computation consists of two parts. First it shifts $\psi_i^{l-1}(x)$ of cube $i$'s parent to the center of cube $i$, which constitutes the far-field interaction. Then, it computes interaction with its interactive-field by summing $\phi(x)$ of the interactive-field that are shifted to the center of cube $i$. 
CHAPTER 3. A 3-D N-BODY ALGORITHM

The sum of these two parts is the local expansion $\psi^l(x)$ for the cube. This is recursively performed when walking down the refinement tree.

Finally step (4) computes the desired potential for each of the particles. The far-field potential is obtained by evaluating $\psi(x)$ of the atomic cube the particle belongs to at the particle position. The near-field potential is obtained by a direct computation.

The error estimate for the algorithm is given by (3.31) and (3.36). By the definition of the near-field and far-field for each of the cubes, we have $(r_0 + \Delta r)/R^\prime < 1/2$ in (3.31) and $R^\prime/|\Delta - r_0| < 1/2$ in (3.36). For a required accuracy of $\epsilon$, we choose $p = \left\lfloor -\log_2 \epsilon \right\rfloor$.

Let's analyze the computational complexity of the algorithm. The total number of the cubes at all levels of the subdivision is

$$8^0 + 8^1 + \cdots + 8^n = \frac{8^{n+1} - 1}{7} = \frac{8N - 1}{7}.$$ 

Step (1) takes $O(N)$ work to compute $N$ series expansions for all the $N$ particles. Summing these expansions to form $\phi(x)$ for every atomic cube takes another $O(N)$ work.

In step (2), it takes one shifting and seven summations of expansions to compute $\phi(x)$ for each of the cubes. This takes constant amount of work for a prespecified $p$. Since the total number of the cubes is of order $N$, the work to compute $\phi(x)$ for all the cubes in step (2) is of order $O(N)$.

In order to compute $\psi(x)$ for each of the cubes in step (3), we need to do one shifting on the parent $\psi(x)$ and one shifting for each of the cube in its near-field, and then sum the resulting expansions. Since the number of cubes in each cube’s interactive-field is bounded by 567, it takes a constant amount of work to compute $\psi(x)$ for each of the cubes for a fixed $p$. Therefore step (3) takes $O(N)$ work in total to compute $\psi(x)$ for all the cubes.

We have chosen the level $n$ of the subdivision so that the number of particles in the near-field of each atomic cube is bounded by a small constant. Thus in step (4), for each of particles the direct computation on the interaction with its near-field takes work of order $O(1)$. Evaluation of the local expansion at the particle position takes again constant amount of work for a fixed $p$. Thus, step (4) takes $O(N)$ work to compute potentials for all the $N$ particles.

The overall complexity of the algorithm described in Figure 3.6, therefore, is the sum of those of step (1) through (4), that is, order $O(N)$. 
(1) **Initial expansions:** for each atomic cube \(i\), compute \(\phi_i^n(x)\).

for \(i\) from 1 to \(8^n\) with step 1 do begin
   compute for each of the particles in the cube \(i\) the series expansion for the potential due to the particle by using Theorem 3.2.1;
   sum these expansions for particles in the cube \(i\) to form \(\phi_i^n(x)\) for the cube \(i\).
end.

(2) **Upward-path:** for each of the cubes \(i\) at level \(l\) of the spatial refinement, compute \(\phi_i^l(x)\) in an upward manner.

for \(l\) from \(n - 1\) to 0 with step -1 do begin
   for \(i\) from 1 to \(8^l\) with step 1 do begin
      shift \(\phi_i^{l+1}(x)\) of cube \(i\)'s eight subcubes to the center of the cube \(i\) by using Lemma 3.2.1;
      Sum the shifted \(\phi_i^{l+1}(x)\) of cube \(i\)'s eight subcubes to form \(\phi_i^l(x)\) for the cube \(i\).
   end.
end.

(3) **Downward-path:** for each of the cubes \(i\) at level \(l\), compute \(\psi_i^l(x)\) in a downward manner.

for \(l\) from 1 to \(n\) with step 1 do begin
   for \(i\) from 1 to \(8^l\) with step 1 do begin
      (3a) shift \(\psi_i^{l-1}(x)\) of cube \(i\)'s parent cube to the center of the cube \(i\), by using Lemma 3.2.3;
      (3b) shift \(\phi(x)\) of the interactive-field to the center of cube \(i\), by using Lemma 3.2.2.
      sum the resulting expansions of (3a) and (3b) to form \(\psi_i^l(x)\) for the cube \(i\).
   end.
end.

Figure 3.6: A sequential algorithm
(4) Final evaluation: For each of particles, compute the potential at the particle.

for particle $p$ from 1 to $N$ with step 1 do begin
  (4a) evaluate $\psi_i^p(x)$ of the atomic cube $i$ the particle $p$ belongs to at the particle position;
  (4b) compute directly interactions with particles in its near-field and in the cube $i$.
    add (4a) and (4b) as the desired potential for the particle $p$.
end.

Figure 3.7: A sequential algorithm (con't)
Chapter 4

Numerical Verifications of the Three-dimensional Algorithm

This chapter numerically verifies the three-dimensional algorithm of Chapter 3 with respect to the achieved accuracy and speed of the algorithm. We first describe a sequential implementation of the algorithm. We discuss our choice of a 3-D tree data structure, and describe heuristic methods for increasing the efficiency of the implementation based on a detailed analysis on the time complexity of the algorithm. We then present experimental results to demonstrate that the algorithm does have linear growth and does compute the forces and potentials to within any prespecified tolerance up to machine precision. We also compare the speed of the algorithm with that of a direct computation and find that for a required average accuracy of $10^{-4}$ for potential fields our implementation will be faster when there are more than 1,000 particles.

4.1 A Sequential Implementation

4.1.1 3-D Tree Data Structure

As described in the previous chapter, the algorithm requires the data structure used for the implementation to hierarchically decompose a three-dimensional space and to support operations such as insertion, deletion, and searching. Therefore a 3-D tree data structure is an obvious choice.

A 3-D tree [knu81] is a balanced eightfold-branching tree, in which each
level of nodes corresponds to one level of the three-dimensional spatial decomposition. The 3-D tree is an extension of the two dimensional tree data structure discussed in Section 2.2 of Chapter 2. In the 2-D tree, a node and its four children correspond to a square and its four subsquares; in the 3-D tree, a node and its eight children correspond to a cube and its eight subcubes. Note that the root of the 3-D tree corresponds to the original cube of space under consideration, while the leaves of the tree correspond to atomic cubes. Since each node of the tree corresponds to a cube in the spatial decomposition, we regard the word “node” and the word “cube” as interchangeable. From now on we will refer to a node as if we refer to the corresponding cube in three-dimensional space, and use the phrase “the center of a node” instead of “the center the cube”. Each node of the tree has pointers to its children, and pointers to its near-field and its interactive-field, as defined before. The near-field and the interactive-field for each node could be computed at run time, but in practice this is too expensive. In a long-term simulation, which iterates over many time steps, these fields would have to be recomputed again and again. Therefore, we initially establish static pointers to each node’s near field and interactive-field. This approach saves time but requires extra storage space for these pointers.

Particles are initially contained in leaf nodes of the tree according to their positions. After each iteration of a simulation, information about particles, such as positions, velocities, etc. is updated. A particle is moved to a new node if it crosses boundaries of an atomic cube. Coefficients of expansions at various stages of the computation are stored in 3-D arrays held by each node. The hierarchical clustering of expansions is done by walking up and down the tree.

The level of spatial refinement is chosen approximately as \( \log_8 N \). Thus, the 3-D tree is about \( \log_8 N \) levels deep.

### 4.1.2 Implementation Details

In this section we discuss the implementation of the algorithm on a Symbolics LISP machine, and describe techniques for increasing the efficiency of the implementation.

Speed is the major concern in our implementation. One way to measure the efficiency of our implementation is to compare it with an implementation of the direct computation. For the first few implementations of our algorithm
on the LISP machine, the code ran extremely slowly. For \( p = 3 \), the crossover point where the running time of our method falls below the running time of the direct method was at about 10,000 particles.

Performance monitoring revealed opportunities for both machine-independent and machine-dependent optimizations.

We discuss the machine-independent optimizations first. Shifting expansions due to interactive-field potentials is very expensive. This accounts for most of the hidden constant in \( O(N) \). For each node in the tree, we need to do 567 shiftings on expansions, since each node has 567 nodes in its interactive-field. We can reduce this cost by grouping together nodes in the interactive-fields. More specifically, we replace eight child nodes by their common parent node (we call it a super-node) if all eight nodes are in the interaction-field of a single node. Using this heuristic, each node has only 140 nodes in its interactive-field. Numerical experiments indicate that this modification speeds up the algorithm by a factor of about 8.

Another source of inefficiency in the initial implementation was the redundancy in computing coefficients in shifting formulas. The repeated calculation of factorials and permutations was removed by storing factorials and permutations in a table. There is also repeated calculation of some common factors in the formulas. We simplified this part of the computation by extracting common factors in sums, and by canceling common factors appearing in successive stages of the computation.

For the machine-dependent optimizations, we found that a major portion of the running time is spent on manipulating coefficients of expansions in shiftings and evaluations. The bottleneck of the computation is the floating-point calculation and indexing of arrays storing coefficients of various expansions. The array reference time was reduced by representing a 3-D array as a 2-D array of 1-D arrays, or as 1-D array of 2-D arrays. This minimizes the array reference overhead since 1-D and 2-D arrays are better supported on the LISP machine\(^1\). Because of the extensive use of array reference, the above improvement was significant. Other machine-dependent optimizations included using tight loops in frequently called procedures, and avoiding the creation of new arrays in manipulating expansions whenever possible.

The above optimizations greatly reduced both the time and storage requirements for the computation. The reduction in storage in turn reduces

\(^1\)The idea is due to Rich Zippel.
the time due to disk paging. Overall, the machine-independent optimizations reduced running time by a factor of 8 and the machine-dependent optimizations provided an additional factor of 4. The running time crossover point with the direct computation is now at about 1,000 for $p = 3$ (See Table 4.4).

4.2 Experimental Verifications of the Algorithm

4.2.1 Accuracy of the Algorithm

In this section, we study the actual achieved accuracy of the algorithm, and compare it with the theoretical prediction given in Chapter 3. We first test our algorithm on the Pythagorean configuration of three bodies and observe how the error of the computation scales with $p$. Then we test the algorithm on two typical distribution models, the uniform distribution model and Plummer distribution model, each with 1,000 particles, and again determine how the error varies with $p$.

(a) The Pythagorean Configuration

The Pythagorean configuration of three bodies was first investigated by C. Burrau in 1913 [SP67]. It is Pythagorean not only in the geometric sense but also with respect to the masses. The sides of the triangle formed by the three bodies are 3, 4, and 5 and the masses of the three bodies are also 3, 4, and 5. The configuration is such that the body with mass $x$ is situated at the vertex opposite to the side of length $x$, where $x$ is one of 3, 4, or 5. The initial configuration of the problem and the coordinate system used are shown in Figure 4.1. Initially the three bodies are situated in the $z = 0$ plane and have speeds zero, consequently the motion is planar.

We use our algorithm to compute the accelerations of the three bodies and observe how error scales with $p$. The experimental results are given in Table 4.1 and plotted in Figure 4.2. We define the relative error in each of the accelerations as

$$
\epsilon_{relative}^{(i)} = 2 \frac{\| \hat{a}_{calculated}^{(i)} - \hat{a}_{true}^{(i)} \|}{\| \hat{a}_{calculated}^{(i)} \| + \| \hat{a}_{true}^{(i)} \|},
$$


CHAPTER 4. NUMERICAL VERIFICATIONS

Figure 4.1: Initial configuration of the Pythagorean configuration of three bodies

where $\vec{a}_{\text{calculated}}^{(i)}$ is the calculated acceleration on body $i$ using our method in single precision arithmetic, and $\vec{a}_{\text{true}}^{(i)}$ is the actual acceleration on that body using the direct method of force computation in double precision arithmetic and is therefore accurate to the machine round-off error of double precision arithmetic.

In Table 4.1, $\epsilon_{\text{max}}$ is the maximum error of all the relative errors in accelerations

$$
\epsilon_{\text{max}} = \text{Max}\left\{\epsilon_{\text{relative}}^{(1)}, \ldots, \epsilon_{\text{relative}}^{(n)}\right\},
$$

$\epsilon_{rms}$ is the root-mean-square error

$$
\epsilon_{rms} = \sqrt{\frac{1}{n} \sum_{i=1}^{n} \left(\epsilon_{\text{relative}}^{(i)}\right)^2},
$$

where $n = 3$ is the number of bodies in the test, and $\epsilon_{\text{potential-energy}}$ is the relative error in total potential energy.
Table 4.1: Accuracy test for the Pythagorean configuration of three bodies. \( p \) is
the highest degree of harmonics retained in an expansion.

Table 4.1 shows that the accuracy of the algorithm is improved by approx-
imately a factor of 2 when \( p \) is incremented by 1 for \( p \) greater than 2.
Consequently, the dots in Figure 4.2 are distributed nearly on a line, since
the error axis is scaled logarithmically.

Note that the near-field in this implementation has been defined so that
the ratio appearing in the error bounds (3.31) and (3.36) of Chapter 3 is
1/2. The factor of 2 decrease in errors for each increase in \( p \) is thus expected
from the formal analysis. The results presented in Table 4.1 and plotted in
Figure 4.2 match well with the predicted error bounds.

The experimental results exhibit two phenomena that warrant further
discussion. First, the error \( e_{\text{relative}}^{(i)} \) in individual accelerations does not de-
Figure 4.2: Plot of the accuracy test for the Pythagorean configuration of three bodies. \( p \) is the highest degree of harmonics retained in an expansion.

crease monotonically as \( p \) increases. Second, the calculated accelerations have nonzero components in the \( z \)-direction, which is perpendicular to the plane of the three-body configuration. Below, we discuss how these phenomena arise.

**Nonmonotonic decrease of \( \epsilon_{\text{relative}}^{(i)} \) with increasing \( p \)**

The nonmonotonic decrease of \( \epsilon_{\text{relative}}^{(i)} \) for individual accelerations with increasing \( p \) is due to the fact that a multipole expansion is a mixed-sign Taylor series. The error due to truncating a multipole expansion is also a mixed-sign series. Consequently, the error from truncating the multipole expansion does not necessarily decrease monotonically when retaining more terms in the multipole expansion. Nevertheless, the error in individual accelerations is always bounded by \( \epsilon_{\text{max}} \) in Table 4.1, which, like the predicted error bounds, decreases monotonically as \( p \) increases.

**Nonzero \( z \) component for accelerations**

The nonzero \( z \) component in our computation results from two kinds of
truncation: the primary truncation and the secondary truncation. We note that each term of the multipole expansion in Lemma 3.2.2 of Chapter 3 is approximated by a convergent Taylor series, which we will call term series. In computing the multipole expansion using Lemma 3.2.2, the primary truncation truncates the multipole expansion and the secondary truncation truncates each term series.

In the multipole expansion computed from Lemma 3.2.2, the coefficients in each dimension are determined by parameters in all three dimensions. If an infinite number of terms were retained in computing both the term series and the multipole expansion, then the $z$ component of the multipole expansion would be zero due to cancellation among nonzero $z$ terms. Since only a finite number of terms are retained, the $z$ component of the multipole expansion is not completely canceled.

Intuitively, we might expect that retaining more and more terms in the term series would reduce in general the secondary truncation error in computing the multipole expansion and in particular the error in $z$-direction. But the experimental results show that the intuition is unfounded. The reason is that the multipole expansion and the term series are mixed-sign series. The fact that each of the terms in the multipole expansion is better approximated by the term series does not necessarily guarantee that the resulting truncated multipole expansion is more accurate.

(b) Uniform Distribution of 1,000 Particles

We verify the accuracy claim of our algorithm using a configuration in which particles are distributed uniformly. In this test, the system has 1,000 particles, each of which has mass $1/1000$. The results are shown in Table 4.2. The errors $\epsilon_{\text{max}}$, $\epsilon_{\text{rms}}$, and $\epsilon_{\text{potential-energy}}$ are defined as before. The results match well with the prediction.

(c) Plummer Distribution of 1,000 Particles

We also test our algorithm to determine if the accuracy of the algorithm is sensitive to the distribution of particles. The Plummer distribution is a nonuniform distribution with the density profile [Her86]

$$\rho(r) = \frac{3M}{4\pi} \frac{r_0^2}{(r^2 + r_0^2)^{5/2}},$$

where $M$ is the mass of the system and $r_0$ is a characteristic radius.
Table 4.2: Accuracy test for the uniform model. $p$ is the highest degree of harmonics retained in an expansion.

which is spherical and isotropic. $M$ is the mass of the system and $r_0$ is the scale-length.

In the test the initial configuration of the system has two clusters, each of which is a Plummer system with 500 particles. Two clusters have the same $M$ and $r_0$ and are separated by $5r_0$. A two-dimensional projection of this configuration is shown in Figure 4.3.

The experimental results are given in Table 4.3. Again, they match very well with the theoretical prediction.

In summary, the accuracy of the algorithm is demonstrated in Table 4.1, Table 4.2, and Table 4.3. These indicate that our method can compute forces and potentials to within any prespecified tolerance up to the machine round-off precision, which is about 7 decimal digits in single precision arithmetic [Ref85]. As we retain more and more terms in the multipole expansions, the accuracy of our method improves, and when $p = 20$ the error of the computation is at the level of the machine round-off errors.

4.2.2 Running Time of the Algorithm

In this section, we test the speed of our algorithm and experimentally determine how the running time grows with the number of particles. We also compare the results with those of a direct computation and determine the
Figure 4.3: Initial configuration of two Plummer systems

<table>
<thead>
<tr>
<th>$p$</th>
<th>$\epsilon_{\text{max}}$</th>
<th>$\epsilon_{\text{rms}}$</th>
<th>$\epsilon_{\text{potential-energy}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.56</td>
<td>0.076</td>
<td>$3.3 \times 10^{-4}$</td>
</tr>
<tr>
<td>2</td>
<td>0.13</td>
<td>0.019</td>
<td>$1.8 \times 10^{-5}$</td>
</tr>
<tr>
<td>3</td>
<td>0.032</td>
<td>0.0053</td>
<td>$7.7 \times 10^{-6}$</td>
</tr>
<tr>
<td>4</td>
<td>0.014</td>
<td>0.0018</td>
<td>$1.2 \times 10^{-6}$</td>
</tr>
<tr>
<td>5</td>
<td>0.0066</td>
<td>$7.3 \times 10^{-4}$</td>
<td>$7.5 \times 10^{-7}$</td>
</tr>
<tr>
<td>6</td>
<td>0.0030</td>
<td>$3.3 \times 10^{-4}$</td>
<td>$2.6 \times 10^{-7}$</td>
</tr>
<tr>
<td>7</td>
<td>0.0013</td>
<td>$1.6 \times 10^{-4}$</td>
<td>$6.4 \times 10^{-8}$</td>
</tr>
<tr>
<td>8</td>
<td>$5.3 \times 10^{-4}$</td>
<td>$7.9 \times 10^{-5}$</td>
<td>$2.1 \times 10^{-8}$</td>
</tr>
<tr>
<td>9</td>
<td>$2.8 \times 10^{-4}$</td>
<td>$4.0 \times 10^{-5}$</td>
<td>$2.1 \times 10^{-8}$</td>
</tr>
</tbody>
</table>

Table 4.3: Accuracy test for the Plummer model. $p$ is the highest degree of harmonics retained in an expansion.
Table 4.4: Speed test for the Multipole-expansion algorithm with $p=3$. Results of a direct computation are also presented for the purpose of comparison. The running time excludes paging time.

<table>
<thead>
<tr>
<th>Number of Particles</th>
<th>Running Time (sec.)</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Multipole-expansion</td>
<td>Direct</td>
</tr>
<tr>
<td>64</td>
<td>12.60</td>
<td>2.30</td>
</tr>
<tr>
<td>128</td>
<td>22.73</td>
<td>9.08</td>
</tr>
<tr>
<td>256</td>
<td>51.38</td>
<td>39.04</td>
</tr>
<tr>
<td>512</td>
<td>409.95</td>
<td>156.34</td>
</tr>
<tr>
<td>1024</td>
<td>620.92</td>
<td>616.62</td>
</tr>
<tr>
<td>2048</td>
<td>1050.65</td>
<td>2468.72</td>
</tr>
<tr>
<td>4096</td>
<td>5221.29</td>
<td>9858.48</td>
</tr>
<tr>
<td>8192</td>
<td>7203.37</td>
<td>39433.92</td>
</tr>
<tr>
<td>16384</td>
<td>11411.68</td>
<td>157735.69</td>
</tr>
</tbody>
</table>

Figure 4.4: Running time growth rate of the Multipole-expansion algorithm, plotted against that of the direct computation.
running time crossover point. In the test, particles are initially distributed uniformly within a spherical region. The space under consideration is the cube of space containing the spherical region.

The results of the test on the speed of the algorithm are given in Table 4.4, and are plotted in Figure 4.4. The running time growth rate of the direct computation is also plotted for comparison. In the plot, both the running-time axis and the number-of-particles axis are scaled logarithmically. The parameter $p$ of the algorithm is chosen as 3 and the calculated potentials are accurate in average to about $10^{-4}$.

Figure 4.4 shows that the running time of our sequential implementation of the algorithm grows linearly with the number of particles. The jumps in the curve when the number of particles are 500 and 4,000 are due to the overhead of maintaining the tree structure when the tree grows one level deeper. The height of the tree grows logarithmically with the number of particles to enforce the constraint that the number of particles in each atomic cube is bounded by a predetermined constant.

The running time crossover point of our algorithm with the direct computation is at about 1,000 particles. This means that for a required average accuracy of $10^{-4}$ for the potentials, our sequential algorithm is faster than direct computation when there are more than 1,000 particles.

### 4.3 Discussion

We have numerically verified the three-dimensional algorithm with respect to the achieved accuracy and speed of the algorithm in the previous sections. In this section, we discuss some additional issues concerning the use of our algorithm.

(a) Tree Overhead

We note the overhead of maintaining the 3-D tree data structure in Section 4.2.2. The tree has a complicated pattern of links for the near-fields and the interactive-fields at each of the nodes. Much of the computation time is spent on tracing these pointers. When the tree is not fully loaded, this part of the overhead will dominate. This accounts for the jumps in the curve in Figure 4.4.
(b) Distribution of Particles

We have assumed in Chapter 3 that the distribution of particles is relatively uniform within the region under consideration. For very clumpy distributions, the performance of the algorithm will degenerate. If we still construct the decomposition tree with height of $\log_b N$, step (4) of the algorithm would take more than $O(N)$ time to compute all the near-field interactions. Otherwise, to maintain the condition that the number of particles in each atomic cube is bounded by a predetermined constant, we would have to construct the tree with height of $O(N)$ in the worst case. Then step (2) and step (3) of the algorithm in Figure 3.7 of Chapter 3 would take time exponential in $N$ to compute all the multipole expansions. One way to improve the efficiency is to dynamically prune empty tree branches, that is, if no particles are founded in a branch, the branch will be pruned from the tree.

(c) Choice of $p$

We have demonstrated in Section 4.2.1 that the accuracy of our method improves as we retain more and more terms in the multipole expansions. How big should the parameter $p$ be in practice? This question depends on the specific nature of a problem. The goal of a numerical simulation is not always "accuracy" in a strictly mathematical sense, but rather "fidelity" to the underlying physics in a sense that is looser and more pragmatic [Pe86]. Often, as in galaxy simulations, the conserved system quantities such as total energy and total momentum are of more interest, so monitoring on these quantities will better reflect how good an algorithm is. Sometimes we only want to look at the statistical behavior of a system. In such context, some types of errors are much more tolerable than others. Another reason is that the error introduced by the discrete integration time step of a simulation is itself comparable with that of truncation [App85]. Therefore choosing small $p$ often suffices. This will greatly reduce the constant factor in our algorithm.
Chapter 5

A Parallel Algorithm

5.1 Introduction

It is clear that the $N$-body method of Chapter 3 can take significant advantage of parallelism. With a massively parallel computer, where we can allocate a separate processing element for each particle, we can compute from expansions the potentials for all of the particles in one parallel step. We can also propagate values up and down the spatial-decomposition tree, as in steps (2) and (3) of the algorithm in Figure 3.6, in parallel, generating expansions for all the nodes at a given level in one parallel step. The depth of the tree grows as $\log N$, so a complete propagation will require time at least on the order of $\log N$.

In this chapter, we present an extremely fast implementation of the $N$-body code—a parallel implementation of the algorithm, whose measured running time does indeed grow as $O(\log N)$. The code runs on the Connection Machine, which is a massively parallel SIMD computer, consisting of up to 65,536 processors, each with its own local memory, connected in a communications network. On the Connection Machine model CM-2 that we have been using, each processor is a one-bit ALU with floating-point unit and 64K bits of local memory.

In designing a practical algorithm for a large parallel machine, there are two issues that must be addressed in addition to the issue of deciding which steps should be performed concurrently. For many large parallel computations, local computations at each processing element are relatively cheap, while communications among processing elements are expensive. The bot-
CHAPTER 5. A PARALLEL ALGORITHM

tleneck of the computation is the interprocessor communication. Thus, an efficient program must be designed to minimize both the need for interprocessor communication and the contention generated by whatever communication is required. For the parallel $N$-body code, we exploit the locality of the algorithm to reduce the need for communication. In addition we exploit the regularity in the communication patterns of the algorithm in order to substantially reduce contention for the underlying communication resource.

5.2 Why a Parallel Computer?

In the tree walking of our sequential algorithm, computations on expansions could be done concurrently within each level of the tree. Tremendous speed-up can be expected if we exploit this parallelism. The question is how to map our problem onto a parallel machine, either a machine specially designed for this problem or a commercially available one, such as the Connection Machine.

Let us take a close look at the sequential algorithm we developed in Chapter 3. First, the tree computation has regularity over all nodes at each level of the tree. A SIMD machine suffices for computation of this pattern. Second, the tree computation has concurrency. Computation at each of the nodes within a single level of the tree can proceed at the same time. Third, the tree computation has spatial locality. In the upward-path of tree walking, each of the nodes talks only to its parent node. In the downward-path of tree walking, each of the nodes talks not only vertically to its parent, but also laterally to those in its interactive-field. The lateral communication with a node’s interactive-field is local to a subtree rooted the node’s predecessor four levels up in the tree. Depending on how we construct the tree on a parallel machine, we might be able to exploit this locality. Lastly, the tree computation has dependencies among levels of the tree. More specifically, computation at one level of the tree cannot proceed unless the computation one level up or down is finished. This determines that the optimal performance we can achieve from the tree computation is proportional to the height of the tree, that is, $O(\log N)$, where $N$ is the number of leaves in the tree.

We choose the Connection Machine to implement a parallel version of the algorithm developed in Chapter 3.
5.3 3-D Tree on the Parallel Computer

We use a 3-D tree as our data structure, as in the sequential implementation. The goal of the algorithm design is to distribute the components of the 3-D tree structure among many processors so that the concurrency in the tree computation can be maximally exploited.

We allocate a processor to each node of the tree. The upward-path and downward-path of the tree walking of the algorithm require that each node in the tree has links to its parent as well as to its children. Explicitly storing these links would be expensive, since each node has one parent and eight children. However, since the tree has a fixed branching factor, a regular allocation of processors for the tree will impose an implicit relationship among the processor addresses, as long as the tree is allocated within a well-defined region of processors. This scheme is called the address-induced representation of the tree structure [Hil85] [Chr84].

More precisely, we allocate consecutive processors to nodes, starting at the root of the tree. We first make a sequential breadth-first scan of the tree to be constructed, and label each node as we encounter it, starting from 0 at the root. Then we assign to every node a processor (or a fixed number of processors) whose address is the breadth-first scan label of the node. We therefore have a formula to induce the parent-child relationship for every processor in the tree. Given the address \( i \) of a processor, we know that its parent processor has address of \( [(i - 1)/8] \), and its child processors have addresses from \( 8i + 1 \) to \( 8i + 8 \). By representing the tree in this way, we avoid having to store pointers at every tree node. However, the saving in storage is achieved at the cost of computing the tree structure every time we trace pointers.

We also allocate a processor to each of the particles in a test \(^1\). Each leaf node of the tree remembers addresses of those particles that belong to the node spatially, so that those particles can be accessed simply by referring to the leaf node during the computation of its near-field. Similarly those particles also remember the leaf node for the initial expansion stage of the computation. This representation of particle-leaf-node links has an additional advantage. After every time step of the simulation, the tree may be updated for particles that move to new cells. This process is easy, since only particle-

\(^1\) It is possible that the allocation of the tree overlaps with that of the particles.
5.4 A Parallel Algorithm

In this section, we will describe a parallel implementation of the algorithm we developed in Chapter 3. The basic patterns of communication include passing and combining data among processors vertically and laterally with respect to the embedded tree structure. The description of the algorithm is summarized in Figure 5.1. Notations are used in the same way as that in the sequential algorithm of Figure 3.6. The number of levels in the tree is $n$.

In Step (1) of the algorithm, each node of the tree is mapped to a processor. Since the tree only needs to be built once, the sequential construction is not a serious drawback as long as we amortize its cost over many time frames in a simulation.

Step (2) finds for every node of the tree its near-field and interactive-field nodes. This could be done at run time, since the addresses for these nodes take a fair amount of memory spaces. But this means that we have to find these nodes every time we want to access them. We know that there are 81 nodes in a node's near-field, and 567 nodes (using heuristics this can be reduced to 140 nodes) in each interactive-field (we ignore the boundary conditions). Experiments have shown that the calculation of these fields is very expensive. So we precompute these fields and store them. Step (3) inserts all the particles into the tree.

Step (4) initially expands potentials for all particles and forms the expansions $\Phi$ for leaf nodes of the tree. An expansion for the potential of a particle is valid outside the leaf node the particle belongs to, and has the center of the node as the reference point. All expansions of a single leaf node are added together. This is done all in parallel. ($\Phi$'s and $\Psi$'s are expansions as defined in Section 3.3, and are represented as arrays of parallel variables.)

Step (5) implements the upward path of the tree walking, which computes $\Phi$ for every node of the tree. The computation proceeds in this way: nodes at one level of the tree in parallel shift the $\Phi$'s to the centers of their respective parents, combine the shifted $\Phi$'s according to whether they have the same parents, and send the results to the parents.

Step (6) walks down the tree and computes $\Psi$'s for every node of the tree. Every node in parallel fetches $\Psi$ at its parent node, and shifts it to its center.
(1) **Tree embedding:** allocate one processor for every tree node, starting at the root.

allocate processor 0 for the root;
for i from 1 to n do begin
   for j from 1 to \(2^i\) do begin
      allocate processor \(8(2^{i-1} - 1) / 7 + j\) for the jth to the leftmost node at level i of the tree.
   end.
end.

(2) **Find for each of nodes its near-field and interactive-field nodes:**

for all nodes of the tree in parallel do begin
   each node finds its near-field and interactive-field nodes and stores the addresses.
end.

(3) **Insert N particles:**

for i from 1 to N do begin
   insert particle i into the tree.
end.

(4) **Initial expansion at leaf nodes:**

for all particles in parallel do begin
   compute expansion \(\Phi\) for the potential due to each of particles relative to center of the leaf node the particle belongs to; add together those \(\Phi\)'s due to particles of the same leaf node and form \(\Phi\) for that leaf node.
end.

Figure 5.1: A parallel algorithm
(5) Upward-path of tree walking:

for i from n down to 1 do begin
    for all nodes at level i in parallel do begin
        at each node
            shift the $\Phi$ to the center of its parent node;
            sum the resulted expansions of those who have the
            same parent node and form $\Phi$ for that parent node.
        end.
    end.
end.

(6) Downward-path of tree walking:

for i from 1 to n do begin
    for all nodes at level i in parallel do begin
        at each node $d_1$
            (6a) shift $\Psi$ at $d_1$'s parent node to $d_1$'s center;
            (6b) for node $d_2$ in $d_1$'s interactive-field do begin
                    shift $\Phi$ at $d_2$ to $d_1$'s center;
                end.
                add the shifted $\Phi$'s together.
                add (6a) and (6b) to form $\Psi$ for node $d_1$.
            end.
        end.
end.

(7) Final evaluation: compute near-field and far-field interactions for each of
particles.

for all particles in parallel do begin
    at each particle
        (7a) evaluate at the particle position the $\Psi$ of the leaf
        node the particle belongs to;
        (7b) compute directly potentials due to its near-field.
        add (7a) and (7b) to form the desired potential.
    end. 

Figure 5.2: A parallel algorithm (con't)
CHAPTER 5. A PARALLEL ALGORITHM

This is done level by level, as in Step (5).

Finally, step (7) finishes the computation. Every particle evaluates in parallel the corresponding \( \Psi \)'s at the particle position. This part of the potential is due to the interaction with its far-field. Then every particle computes in parallel the near-field interactions directly. The sum of the far-field and near-field interactions is the desired potential at this particle position.

We have seen from the above description that computation at a given level of the tree in steps (4)–(7) takes one parallel step. Thus, step (4) and step (7) each take constant time. Since the tree has depth of \( \log N \), and computations at different levels of the tree have dependencies among them, a complete propagation in step (5) and step (6) takes \( O(\log N) \) time. Therefore, the time complexity of the parallel algorithm, excluding the initial set-up time for the tree and communication costs, is \( O(\log N) \).

Let us now look at the memory usage in the computation. When we retain in an expansion only the terms of spherical harmonics with degree less than four, the total number of terms in an expansion is 20. As verified by the experimental results from the sequential implementation, this guarantees in average a four decimal-digit accuracy in potentials. Each of the coefficients in an expansion is a 32-bit single precision floating-point number. We have three different arrays of coefficients for each node of the tree in the computation. Thus expansions alone take about 2K bits of memory at each of nodes. Storing the interactive-field and near-field takes another 4K bits of memory. We also need some stack space for intermediate computation. Therefore, about 10K bits of local memory at each node suffice for our purpose. In case a machine has insufficient amount of local memory per processor, we can remedy this by simulating a node of the tree with several processors. In such a case, however, the cost of the communication overhead would be high.

5.5 Experimental Results

We have implemented the parallel algorithm on the Connection Machine. The experimental results are summarized in Table 5.1. The running time growth rate of the implementation is plotted in Figure 5.3, along with that of the sequential implementation. The results experimentally verify that the parallel implementation of the algorithm on the Connection Machine scales
### Table 5.1: Experimental results on the Connection Machine. The running time excludes paging time.

<table>
<thead>
<tr>
<th>Number of Particles</th>
<th>Running Time (sec.)</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Lisp Machine</td>
<td>Connection Machine</td>
</tr>
<tr>
<td>64</td>
<td>12.60</td>
<td>9.06</td>
</tr>
<tr>
<td>128</td>
<td>22.73</td>
<td>14.99</td>
</tr>
<tr>
<td>256</td>
<td>51.38</td>
<td>23.72</td>
</tr>
<tr>
<td>512</td>
<td>409.95</td>
<td>33.86</td>
</tr>
<tr>
<td>1024</td>
<td>620.92</td>
<td>51.78</td>
</tr>
<tr>
<td>2048</td>
<td>1050.65</td>
<td>60.65</td>
</tr>
<tr>
<td>4096</td>
<td>5221.29</td>
<td>71.83</td>
</tr>
<tr>
<td>8192</td>
<td>7203.37</td>
<td>79.63</td>
</tr>
<tr>
<td>16384</td>
<td>11411.68</td>
<td>94.17</td>
</tr>
</tbody>
</table>

### Figure 5.3: Running time growth rate of the parallel algorithm, with that of the sequential implementation.
logarithmically in the total number of particles in the simulation, even when we take the communication into the consideration. We find that the parallel algorithm runs faster than the sequential algorithm, even on small examples. Compared with the sequential implementation, the parallel algorithm exhibits a speed-up factor of about 10 for $N=1,000$ and a speed-up factor of about 100 for $N=10,000$.

The complexity issue of the parallel algorithm, which is complicated by the need for communication, will be discussed in more detail in the following section.

### 5.6 Communication Patterns

We have discussed the complexity issue due to the floating-point computation in Section 5.4. Our experiments have shown that the floating-point computation takes about 20% of the total running time and the interprocessor communication consumes a major portion of the rest.

A careful study at the communication patterns reveals that there are two major kinds of communication going on in the computation. The first kind of communication occurs between adjacent levels of the tree. We call this *vertical communication*. The second kind of communication occurs between nodes and their interactive-fields and near-fields, and takes place within a single level of the tree. We call this *lateral communication*. The experimental results show that, when the tree is averagely loaded, the lateral communication constitutes about 20% of the total running time, and the vertical communication accounts for another 25%. In the following sections, we describe various optimizations we have made to reduce the communication overhead.

---

2. The parallel computation has about 200 million floating-point calculations in a test of 1,000 particles, and has about 1,600 million floating-point calculations in a test of 10,000 particles.

3. The results are obtained on the Connection Machine using the metering program provided by Shawn Mclean.

4. Recall from Section 4.1.2 that a super-node is a node formed by grouping on interactive-field nodes. Communicating with a super-node in the lateral communication involves two adjacent levels of the tree. Although this situation is similar to the vertical communication, for the sake of simplicity we will consider it to be the lateral communication.
5.6.1 Reducing Communication Bottlenecks

The lateral communication required for computing interactions with near-fields and interactive-fields is one of the most expensive parts of the parallel computation. The cost of the lateral communication is due largely to message collisions in which many processors try to access a single processor at the same time. Since processor access on the Connection Machine is handled in an exclusive reading and exclusive writing way, the time of a successful message routing is proportional to the maximum number of message collisions.

We tried to ease the communication bottleneck by reducing the number of collisions in the lateral communication. We know that each node has 140 nodes in its interactive-field, and those nodes are represented as a list of addresses. A close look shows that many processors often have the same node as their interactive-field node at the same time, due to the way the interactive-field list is constructed. Our algorithm, therefore, randomizes the order in which one’s interactive-field nodes nodes are accessed, in the hope that two processors are less likely to access a single processor at the same time. As a result, the randomization improves the performance of the algorithm.

5.6.2 Localizing Communication

In computing the interaction with a node’s interactive-field, we need to use coefficients of expansions stored in an interactive-field node again and again. Instead of fetching these every time we use them, we fetch them once and store them on a temporary local stack. This can reduce the traffic by a factor of 5, when computing expansions whose highest degree is three.

5.6.3 Combining and Delegating Messages

The Connection Machine router has fairly good performance when the network is averagely loaded, but the routing time scales up as the number of collisions in the routing process. Collisions are mostly due to concurrent read or concurrent write to a single processor. To reduce the collisions, we can use the scan operation, which does the segmented parallel prefix computation on
the Connection Machine\textsuperscript{5}. To resolve collision due to the concurrent read, we first do an exclusive read, and then do a \textit{scan-copy} operation. In case of the concurrent write combined with addition, we do a \textit{scan+} and then an exclusive write instead. If elements of a segment to be scanned are not in consecutive processors, in order to apply the \textit{scan} operator we have to first project the elements to a new segment so that the new segment has elements in consecutive processors. This extra projection pays off when there are more than thirty processors in a segment.

![Figure 5.4: Concurrent write](image)

In our implementation of the algorithm, the final direct interactions with a node's near-field often gives rise to 10–20 collisions. Therefore we use a different method to reduce the collisions. The idea is to structure the message routing through a tree that distributes the message collisions over many intermediate nodes. This tree-like routing allows many of the collisions to be handled in parallel, thereby reducing the total time delay due to the collisions.

Consider an example in which 16 processors access a single processor for concurrent write (Figure 5.4). Using the tree routing scheme outlined above,

\textsuperscript{5}Given a binary associative operator $\ast$ and a segment of sequence of elements $x_1, x_2, \ldots, x_m$, the parallel prefix operation computes $x_1, x_1 \ast x_2, \ldots, x_1 \ast x_2 \ast \cdots \ast x_m$ in $O(\log(m))$ time [HJ86]. In particular, there are \textit{scan+} and \textit{scan-copy} operations. The \textit{scan} operation is very efficiently implemented on the Connection Machine. For simple binary operations such as $+$ and \textit{copy}, it takes the same order of time as a routing cycle does, which is the fundamental unit for time measurement, and is therefore considered to take constant amount of time.
Figure 5.5: Combining messages

Figure 5.6: Delegating messages
we can use a two-level tree to combine every four of the writes together, and then combine every four of the resulting writes (Figure 5.5). Even though there are 20 total collisions in this tree, the time delay is only proportional to 8 collisions because the nodes at the intermediate level of the tree handle their collisions in parallel. This process is called message-combining. The dual process, message-delegating, reduces collisions due to concurrent read, as illustrated in Figure 5.6.

In general, suppose there are \( N \) collisions in total. Each stage of combining or delegating reduces the number of messages by a factor of \( m \). Therefore the number of stages is \( \log_m N \). What is the optimal value of \( m \) such that the function

\[
\text{Routing-time}(m, N) = m \log_m(N)
\]

is minimal? Using elementary calculus, we find that the optimal \( m = e \), where \( e \) is the base of the natural logarithm 2.718... Of course, we must actually choose an integral value for \( m \) (2 or 3).

Message-combining and message-delegating have been very effective, and in practice have given a factor of 2 improvement in speed.

5.7 Discussion

In this section, we discuss trade-offs we made in our parallel implementation, and suggest alternatives to this implementation.

5.7.1 Grid Representation

We have used an address-induced scheme to construct the tree on the Connection Machine. This mapping scheme has the merit of simplicity. It also saves memory space, by avoiding explicitly storing pointers in the tree. As a result of this consecutive mapping scheme, only a subset of processors are active at a time, since the computation proceeds level by level in the tree.

The alternative to the address-induced scheme is to use the Connection Machine grid-like communication network called the NEWS grid. This structure provides fast communication for local or highly structured communication patterns [Hil85]. Every processor is assigned a grid coordinate and can be addressed by specifying this coordinate. The tree can be embedded in the
Figure 5.7: Superimposed mapping. Each square in the grid represents a processor, and each number represents a node in the tree.

Figure 5.8: Non-superimposed mapping. Each square in the grid represents a processor, and each number represents a node in the tree.
network in two different ways. One way is to superimpose levels of the tree so that nodes at one level may share processors with those at other levels, as illustrated in Figure 5.7. As a result, some processors require more memory than others do. This scheme makes the processor utilization at most 100%. However, since all processors on the Connection Machine must have the same amount of local memory, the nonuniform memory requirement across processors wastes memory in some processors. The alternative is to unfold levels of the tree and to assign to every node a processor, as in Figure 5.8. This makes the memory utilization at most 100% but wastes some processors.

By using the NEWS grid, we can take advantage of the fast grid routing for highly structured communication patterns, such as the communication with interactive-fields discussed below.

5.7.2 Exploiting Regularities in Communication Patterns

To further reduce the number of collisions discussed in Section 5.6.1, Alan Ruttenberg has suggested a scheme to exploit the regularities in the communication patterns. We have not implemented this, so we do not know how much it decreases the running time of the algorithm. To keep the discussion simple, we present the scheme in two dimensions. The extension to three dimensions is straightforward.

We classify squares of the spatial decomposition into four classes. Suppose all the squares of smallest size of Figure 5.9 are at level $l$. We first group every four squares that share a common parent at level $l-1$. Then we classify the four squares of a group as class 1, 2, 3, and 4 respectively according to the relative position of a square in the group, as labeled in the Figure 5.9.

We find that for every group there is spatial symmetry among the interactive-fields of its four squares. In Figure 5.10, square $a'$ is an interactive-field square of square $a$. A clockwise rotation of 90 degrees of the square $a'$ with respect to the point $O$ results in another square $b'$, which is an interactive-field square of square $b$, and so on. As a result of the spatial symmetry, the four squares $a'$, $b'$, $c'$, and $d'$ are in different classes, as shown in Figure 5.10. This symmetry property can be used to completely eliminate message collisions in the communication with interactive-field squares. The idea is as follows: each of four squares $a$, $b$, $c$, and $d$ interacts with squares $a'$, $b'$, $c'$, and $d'$ respectively at the same time so that the communication
Figure 5.9: Classification of squares

Figure 5.10: Symmetric communication pattern
pattern is symmetric with respect to four squares; the same is true for all other groups. Since the four squares \(a', b', c', d'\) are all in different classes and communication patterns of two different groups are identical except that they are shifted from each other by some distance, none of the four squares \(a', b', c', d'\) will be fetched by more than one square at a time. Thus, lateral communication can take place without collisions.

In Section 4.1.2, we discussed super nodes in the interactive-fields. Since the super nodes are one level higher in the decomposition tree than those ordinary nodes in the interactive-field, interactions with a super node using the above scheme will give rise to collisions. However, the maximum number of interactions to a super node will not exceed 4 at a time. Therefore the collisions can be avoided by spreading the data to be fetched at a super node to its four children, and afterwards, access to the super node will be directed to the four children.

In the above discussion we also ignored the boundary case. We can introduce dummy squares outside the boundaries so that boundary squares still preserve the symmetry property of interactive-fields.

5.8 Conclusion

We have described the parallel implementation of the three-dimensional \(N\)-body algorithm that runs on the Connection Machine in order \(O(\log N)\) time in this chapter. Compared with the previous \(N\)-body methods, our method has a demonstrated advantage in simulating large number of particles. The parallel implementation achieves tremendous speedup in running time and computes the forces and potentials to within any prespecified tolerance up to machine precision.

Combining the results of this chapter with the analytical results derived in Chapter 3 and the experimental results of the sequential implementation in Chapter 4, we have presented a complete analysis of the three-dimensional \(N\)-body algorithm both theoretically and experimentally. Because of the superior speed and accuracy of our algorithm, we expect that it will find applications in astrophysics, plasma physics, fluid dynamics, and molecular dynamics. Nevertheless, there are additional improvements that we believe will enhance the practicality of the algorithm and, therefore, are worth further research effort.
We have observed in Section 5.6 that local computation—mainly floating-point calculation at each processing element of the parallel implementation—takes only about 20% of total running time. The rest of the running time is mostly spent on communication among processing elements. The results indicate that the bottleneck is the interprocessor communication. Therefore, an efficient implementation of the algorithm should explore the communication requirements of the algorithm.

In Section 5.7 we described alternatives to the current parallel implementation to improve the performance of the algorithm. By using the NEWS grid, we can exploit the localities of the algorithm and take advantage of the fast grid routing. The use of the NEWS grid will also enable us to exploit the regularities in the lateral communication of the algorithm. We expect that the resulting implementation will ease the bottleneck of the interprocessor communication.
Appendix A

Sequential Code

This appendix contains a sequential code implementing the three-dimensional $N$-body algorithm on the Symbolics LISP machine. The code is written in Zetalisp. Appendix A.1 provides data structure abstraction layer for the sequential code, and Appendix A.2 implements the algorithm.

A.1 Data Structure Abstraction

;;; -*- Base: 10; Mode: LISP; Package: User; Syntax: Zetalisp;  
;;; Lowercase: T; Fonts: CPTFONT,HL10,HL12B,HL12I; -*-  
;;; (c) Copyright 1986 Massachusetts Institute of Technology

;;; ===================================================================
;;; POINTS
;;; ===================================================================

(defstruct (pt :named :conc-name (:constructor pt (x y z)))
  x y z)

; toss out any info beyond third decimal place
(defsubset point-niceify (x)
  (* (round x .001) .001))

(defsubset pt-component-=(x1 x2)
  (= (point-niceify x1) (point-niceify x2)))
(defselect ((:property pt named-structure Invoke))
  (:print-self (pt stream ignore ignore)
    (si:printing-random-object (pt stream :no-pointer)
      (format stream "Pt: " "D, " "D" (pt-x pt) (pt-y pt) (pt-z pt))))))

(defun pt-zero-p (pt)
  (and (zerop (pt-x pt)) (zerop (pt-y pt)) (zerop (pt-z pt))))

(defun pt-equal (p1 p2)
  (and (= (pt-x p1) (pt-x p2))
       (= (pt-y p1) (pt-y p2))
       (= (pt-z p1) (pt-z p2))))

; check if p1 and p2 are the same object
(defun pt-same (p1 p2)
  (eq p1 p2))

(defun pt-plus (p1 p2)
  (pt (+ (pt-x p1) (pt-x p2))
       (+ (pt-y p1) (pt-y p2))
       (+ (pt-z p1) (pt-z p2))))

(defun pt-difference (p1 p2)
  (pt (- (pt-x p1) (pt-x p2))
       (- (pt-y p1) (pt-y p2))
       (- (pt-z p1) (pt-z p2))))

; the square of the Euclidean distance; the arguments are not points
; but coords
(defun pt-dist (x1 y1 z1 x2 y2 z2)
  (let ((delta-x (- x1 x2))
         (delta-y (- y1 y2))
         (delta-z (- z1 z2))
         (+ (* delta-x delta-x) (* delta-y delta-y) (* delta-z delta-z))))

; if one of the arguments is NIL, returns NIL.
(defun careful-+ (p1 p2)
  (if (and p1 p2)
      (+ p1 p2)))

(defun pt-average (p1 p2)
  (pt (careful-// (+ (pt-x p1) (pt-x p2)) 2)
       (careful-// (+ (pt-y p1) (pt-y p2)) 2)
       (careful-// (+ (pt-z p1) (pt-z p2)) 2)))
;;; CAREFUL--// is a function that does division carefully.
;;; If the arguments are both integers, it will do integer division
;;; provided X is an integer multiple of Y. Otherwise it will return
;;; the correct floating-point result. Thus it doesn’t have lossage
;;; due to doing an integer division when X is not an integer multiple
;;; of Y. (If either X or Y is not an integer, it does the regular
;;; polymorphic //).

(defun careful--// (x y)
  (cond ((and (fixp x) (fixp y))
         (if (zerop (mod x y))
             (// x y)
             ((/ (float x) (float y))))
         (t
             (// x y))))

(defun pt-scalar-times (scalar pt)
  (pt (* scalar (pt-x pt))
       (* scalar (pt-y pt))
       (* scalar (pt-z pt))))

(defun pt-times (p1 p2)
  (pt (* (pt-x p1) (pt-x p2))
       (* (pt-y p1) (pt-y p2))
       (* (pt-z p1) (pt-z p2))))

(defun distance-between-points (p1 p2)
  (sqrt
   (pt-dist
    (pt-x p1) (pt-y p1) (pt-z p1) (pt-x p2) (pt-y p2) (pt-z p2))))

(defconst pt-0-0-0 (pt 0 0 0)
  "The origin")

(defun printable-point ((x y z) pt &body body)
  (once-only (pt)
    '(let ((x (pt-x ,pt))
           (y (pt-y ,pt))
           (z (pt-z ,pt))
           ,@body))
  (defun printable-point (pt)
'(pt , (pt-x pt), (pt-y pt), (pt-z pt))

;;; -------- Thinking of Points as Vectors --------

(defun vector-norm (pt)
  (sqrt (vector-norm-squared pt)))

(defun vector-norm-squared (pt)
  (let ((x (pt-x pt))
        (y (pt-y pt))
        (z (pt-z pt)))
    (+ (* x x) (* y y) (* z z))))

; be warned: will die if the norm is zero
(defun unit-vector (pt)
  (let ((norm (vector-norm pt)))
    (pt (careful-/- (pt-x pt) norm)
        (careful-/- (pt-y pt) norm)
        (careful-/- (pt-z pt) norm))))

(defun within-box? (pt pt1 pt2)
  (and (< (pt-x pt) (pt-x pt2)) (> (pt-x pt) (pt-x pt1))
       (< (pt-y pt) (pt-y pt2)) (> (pt-y pt) (pt-y pt1))
       (< (pt-z pt) (pt-z pt2)) (> (pt-z pt) (pt-z pt1))))
;;; 3D-TREES

(deflavour item-not-found ()
  (error))

(deflavour 3D-tree-item-not-found
  (3D-tree key item)
  (item-not-found)
  :initable-instance-variables
  :gettable-instance-variables)

(defmethod (3D-tree-item-not-found :report)
  (&optional (stream standard-output))
  (format stream
    "Could not find item for key ~S in the 3D-tree ~S."
    key 3D-tree))

;;; 3D-tree's are implemented as binary branching trees. The X levels
;;; are separated by the Y levels and Z levels. The keys used here must
;;; be points. This version maintains multiple items per location.
;;; find-by-key will now multi-value return: node item-list position.
;;; item-list is ALWAYS a list and count is the length of the list

(deflavour 3D-tree (root
  (lower-left-front-vertex)
  (upper-right-behind-vertex)
  (half-leaf-diag-vec)
  ; precision requirement for multipole expansions.
  (precision-p 0)
  (softening-parameter 0)
  ; # items inserted into the tree
  (number-of-items 0)
  ; # nodes in the tree
  (number-of-nodes 0)
  (total-mass 0)
  (CM-position pt-0-0-0)
  (CM-velocity pt-0-0-0)
  (energy-T 0)
  (energy-U 0)
APPENDIX A. SEQUENTIAL CODE

(true-energy-U 0)
; the depth of the tree
(depth 0)
(item-template)
; # of x levels in the tree
(height-of-tree 0))
()
:gettable-instance-variables
:settable-instance-variables)

(defstruct (3D-node :named :conc-name)
    (left ())
    (right ())
    (key ())
    ; true if this node is on someone else's interactive-field
    (mark-bit ())
    (item-list ())
    (phi)
    (psi)
    (potential-at-this-point nil)
    (true-potential-at-this-point nil)
    (neighbor-list ())
    (near-field-list ())
    (interaction-list ())
    (count 0)
    (sort-by-y 3) ; 0 indicates the node is on X level,
                 ; 1 on Y level, 2 on Z level, and 3 for root.
)

(defun make-3D-tree (REST init-options)
    (lepr-funcall #\'make-instance '3D-tree init-options))

(defmethod (3D-tree :after :init) (ignore)
    (send self :clear)
    (setf (3D-node-count root) 0))

(defmethod (3D-tree :clear) ()
    (setq root (make-3D-node))
    (setq number-of-items 0)
    (setq number-of-nodes 0))
(setq total-mass 0)
(setq CM-position pt-0-0-0)
(setq CM-velocity pt-0-0-0)
(setq energy-T 0)
(setq energy-U 0)
(setq true-energy-U 0)
(setq precision-p 0)
(setq softening-parameter 0)
(setq depth 0)
(setq height-of-tree 0)

(defmethod (3D-tree :empty-p) ()
  (null (3D-node-right root)))

(defmethod (3D-tree :print-self) (stream ignore ignore)
  (si:printing-random-object (self stream)
    (format stream "3D-tree, "D nodes and "D items"
            number-of-nodes number-of-items)))

(defselect ( (:property 3D-node named-structure-invokes))
  (:print-self (node stream ignore ignore)
    (si:printing-random-object (node stream :no-pointer)
      (format stream "3D node "A: item-list:"S"
        (if (3D-node-key node)
          (format nil "("D, "D, "D)"
            (pt-x (3D-node-key node))
            (pt-y (3D-node-key node))
            (pt-z (3D-node-key node)))
          
          "(??, ??, ??)"
        )
      )
    )
  )
)

(defmethod (3D-tree :grind) (optional (stream standard-output))
  (labels ((grind (node indent side)
    (unless (null node)
      (format stream ""X"vI"[Root":L":R"] ("[X":Y":Z"]): "S"
        indent
        side
        (3D-node-sort-by-y node)
        node)
      (grind (3D-node-left node) (+ 2 indent) 1))
  )
)
(grind (3D-node-right node) (+ 2 indent) 2))))
(grind root 0 0)))

(defmethod (3D-tree :insert) (item key &rest *neighbor-list*)
  (let ((key-x (pt-x key))
        (key-y (pt-y key))
        (key-z (pt-z key))
        (my-depth 1)
        (neighbor-list (copytree (car *neighbor-list*))))
    (labels
      ((insert-internal (node sort)
          (let ((key-index (nth sort (list key-x key-y key-z))))
            (pt-index (nth sort (list #'pt-x #'pt-y #'pt-z))))
            (incf my-depth) ; keep track of how deep we are
            (if (<= key-index (funcall pt-index (3D-node-key node)))
                ;; the item either belongs here or the left node
                (cond
                  ;; the item goes to left node, either make left node
                  ;; or search for proper place
                  ((null (3D-node-left node))
                    (cond (item
                              (setf (3D-node-item-list node)
                                    (push item (3D-node-item-list node))))
                              (incf (3D-node-count node))
                              (incf number-of-items))
                              (t (setq depth (max depth my-depth))
                                (incf number-of-nodes))
                                (setf
                                  (3D-node-left node)
                                  (make-3D-node
                                   neighbor-list neighbor-list
                                   key key
                                   sort-by-y (remainder (+ sort 3) 3))))))
                  ; left node exists search for the proper place for
                  ; the item
                  (t (insert-internal
                       (3D-node-left node) (remainder (+ sort 3) 3)))
                      ; the item belongs to the right of here either make
                      ; right node or search for proper place
                      (cond ((null (3D-node-right node))
                              (cond
                                (item (setf (3D-node-item-list node)
                                           (push item
                                           (incf (3D-node-count node))
                                           (incf number-of-items)))
                                t
                                (t (insert-internal
                                    (3D-node-right node) (remainder (+ sort 3) 3))))))))
      )
    )
  )
)
(DEFMETHOD (3D-TREE :FIND-BY-KEY)
  (KEY &OPTIONAL (EQUAL-PREDICATE #'PT-EQUAL))
  (LET ((KEY-X (PT-X KEY))
         (KEY-Y (PT-Y KEY))
         (KEY-Z (PT-Z KEY)))
    (LABELS ((FIND-X (NODE))
              (COND ((NULL NODE) NIL)
                     (COND (FUNCTION EQUAL-PREDICATE KEY (3D-NODE-KEY NODE))
                           (VALUES NODE (3D-NODE-ITEM-LIST NODE) KEY))
                     ((< KEY-X (PT-X (3D-NODE-KEY NODE)))
                       (FIND-Y (3D-NODE-LEFT NODE)))))))

;;;; body of insert starts here
(cond ((AND ITEM (ATOM ITEM))
       (FORMAT T ""˜%ITEM "D not a list!" ITEM")
       ; if the tree is empty, start the tree
       ((NULL (3D-NODE-RIGHT ROOT))
        (SETF (3D-NODE-RIGHT ROOT)
              (MAKE-3D-NODE NEIGHBOR-LIST NEIGHBOR-LIST
                             KEY KEY
                             SORT-BY-Y 0))
        (SETF NUMBER-OF-NODES 1)
        (INCFF DEPTH))
       ; otherwise start the search from the top of the tree
       (T (INSERT-INTERNAL (3D-NODE-RIGHT ROOT) 0))))

(DEFMETHOD (3D-NODE-ITEM-LIST NODE))
(INCFF (3D-NODE-COUNT NODE))
(INCFF NUMBER-OF-ITEMS)
(T (SETQ DEPTH (MAX DEPTH MY-DEPTH))
 (INCFF NUMBER-OF-NODES)
 (SETF (3D-NODE-RIGHT NODE)
       (MAKE-3D-NODE
        NEIGHBOR-LIST NEIGHBOR-LIST
        KEY KEY
        SORT-BY-Y (REMAINDER (+ SORT 3))))))
; right node exists, search for the proper place
; for item
(T (INSERT-INTERNAL
     (3D-NODE-RIGHT NODE) (REMAINDER (+ SORT 3))))
))))

APPENDIX A. SEQUENTIAL CODE

71
(t (find-y (3D-node-right node))))

(find-y (node)
  (cond ((null node)
           nil)
          ((funcall equal-predicate key (3D-node-key node))
           (values node (3D-node-item-list node) key))
          ((<= key-y (pt-y (3D-node-key node)))
           (find-z (3D-node-left node)))
          (t (find-z (3D-node-right node))))

(find-z (node)
  (cond ((null node)
          nil)
          ((funcall equal-predicate key (3D-node-key node))
           (values node (3D-node-item-list node) key))
          ((<= key-z (pt-z (3D-node-key node)))
           (find-x (3D-node-left node)))
          (t (find-x (3D-node-right node))))

(find-x (3D-node-right root))))

(defun (3D-tree :map-over-items) (func)
  (labels ((mapper (node)
             (when (not (null node))
               (when (3D-node-left node)
                 (mapper (3D-node-left node)))
               (loop for item in (3D-node-item-list node) do
                  (funcall func item (3D-node-key node)))
               (when (3D-node-right node)
                 (mapper (3D-node-right node))))
             (mapper (3D-node-right root))))
A.2 Sequential Code

;;;; --- Mode: LISP; Syntax: Zetalisp; Lowercase: Yes; Package: USER;
;;;; Base: 10; Fonts: CPTFONT,HL12,HL12B,HL12I; ---
;;;; (c) Copyright 1986 Massachusetts Institute of Technology

;;;; ================================
;;;; Math Procedures
;;;; ================================

;;;; i
;;;; Procedures for computing permutation: ( ).
;;;; j
;;;; Using the property: \((i \text{ choose } j) = (i \text{ choose } (i-j)),\)
;;;; which is valid when \(i\) is a nonnegative integer!

(defun perm (i j)
  (let ((result 1)
        (j (min j (- i j))))
    (do ((m i (1- m))
         (n 1 (1+ n)))
      (> n j) (float result))
    (setq result (* (cl:sqrt m n) result))))

;;;; Procedure for computing the factorial \(i!\).

(defun factorial (i)
  (labels ((factorial-internal (n)
             (if (< n 2)
                 1
                 (* n (factorial-internal (1- n)))))
            (float (factorial-internal i)))))

;;;; Storing the products of factorials \(i!, j!\) and \(k!\) in a table.

(defun make-fac-table (p)
  (let ((fac-table-internal (make-array (list (1+ p) (1+ p))))
        (declare (sys:array-register-id fac-table-internal)))
    (loop for i from 0 to p
do
(loop for j from 0 to (- p i)
  do
    (let ((c (make-array (1+ (- p i j)))))
      (declare (sys:array-register c))
      (setf (sys:%id-aref fac-table-internal (+ i (* (1+ p) j))) c)
      (loop for k from 0 to (- p i j)
        do
          (setf (aref c k)
            (* (factorial i) (factorial j) (factorial k))))))
  fac-table-internal))

(defun make-fac-table-1 (p)
  (labels ((fac-1 (i)
    (let ((result 1.0e0)) ;; i -> 1.0e0
      (do ((m -0.5 (1- m))
          (n 1 (1+ n)))
        ((> n i) result)
        (setq result (* m result))))))
  (let ((fac-table-internal-1 (make-array (1+ p))))
    (loop for i from 0 to p
      do (aset (fac-1 i) fac-table-internal-1 i)
    fac-table-internal-1))

(defun make-fac-table (make-fac-table-1 20))
APPENDIX A. SEQUENTIAL CODE

;;; Expansion Theorems

;;; Theorem 3.2.1:
;;; produces an expansion for the potential valid outside a sphere of
;;; radius |pt| due to a charge at pt.

(defun expansion1 (pt p charge)
  (labels
    ((coeff (i j k y1 y2 y3)
      (let* ((i (coerce i 'real))
                (y1 (coerce y1 'real))
                (y2 (coerce y2 'real))
                (y3 (coerce y3 'real))
                (array-1 (coerce array-1 'real))
                (array-2 (coerce array-2 'real)))
     (declare (type array-1 double precision array-2 double precision)
              (type coeff double precision)
              (type point real array-1 real array-2 real))
     (if pt
       (let* ((y1 (pt-x pt))
               (y2 (pt-y pt))
               (y3 (pt-z pt))
               (array-1 (coerce array-1 'real))
               (array-2 (coerce array-2 'real)))
       (declare (type array-1 double precision array-2 double precision)
                (type coeff double precision)
                (type point real array-1 real array-2 real))
       (loop for i from 0 to p
             do (loop for j from 0 to (- p i)
                      do (let ((c (make-array (- array-1 i j)))
                             (declare (type array-1 double precision)
                                      (type coeff double precision)
                                      (type point real array-1 real array-2 real))
                             (loop for k from 0 to (- p i j)
                                   do (let ((k (coerce k 'real))))
                                        (setf (aref c k) (coeff i j k y1 y2 y3)))))))))

;;; This procedure is essentially the same as the previous one except
;;; that it produces a expansion for a set of charges.
;;; Pair-list is a list of cons (point-position . charge).

(defun expansion1-1 (p pair-list)
  (labels
    ((coeff (i j k pair-list)
      (loop for pair in pair-list
             for pt = (car pair)
             sum

    (loop for pt in p
          do (coeff i j k pt)))))))

(defun expansion1-1 (p pair-list)
  (labels
    ((coeff (i j k pair-list)
      (loop for pair in pair-list
             for pt = (car pair)
             sum

    (loop for pt in p
          do (coeff i j k pt))))))
(* (cdr pair)
  (expt (pt-x pt) i)
  (expt (pt-y pt) j)
  (expt (pt-z pt) k))
into accum
finally
{return (/ (expt -1 (+ i j k)) accum)
  (aref (sys:%id-aref fac-table (+ i (* 21 j)))
    k))
}

(if pair-list
(let* (((array-l (1+ p))
    (array (make-array (list array-l array-l))))
  (declare (sys:array-register-1d array))
  (loop for i from 0 to p
    do
    (loop for j from 0 to (- p i)
      do
      (let ((c (make-array (- array-l i j))))
        (declare (sys:array-register c))
        (setf (sys:%id-aref array (+ i (* array-l j))) c)
        (loop for k from 0 to (- p i j)
          do
          (setf (aref c k) (coeff i j k pair-list))))))
  array))))

;;; Theorem 3.2.2:
;;; produces an expansion for the potential valid inside the sphere of
;;; radius |pt| due to a charge at pt.
;;; This procedure actually computes b[i,j,k]*(i!)*(j!)*(k!) instead!

(defun expansion2 (pt p)
(labels
  ((phase-1 (i j k)
    (/ (aref fac-table-1 (+ i j k))
      (aref (sys:%id-aref fac-table (+ i (* 21 j))) k)))
    ;; computes coefficients b''[i,j,k]
  (phase-2 (i j k array 2t1 2t2 2t3)
    (loop for l from 0 to (/ // i 2)
      sum
      (loop for m from 0 to (/ // j 2)
        sum

(loop for n from 0 to (// k 2)
  sum
  (* (aref (sys:%id-aref array
    (+ (- i 1) (* (1+ p) (- j m))))
    (- k n))
  (perm (- i 1) l)
  (perm (- j m) m)
  (perm (- k n) n)
  (expt 2t1 (- i (* 2 l)))
  (expt 2t2 (- j (* 2 m)))
  (expt 2t3 (- k (* 2 n))))
into accum
finally (return (* (expt -1 (+ i j k)) accum))))

(if pt
  (let* ((array-1 (1+ p))
    (array0 (make-array array-1))
    (array1 (make-array (list array-1 array-1)))
    (array2 (make-array (list array-1 array-1)))
    (pt-norm (vector-norm pt))
    (2t1 (* 2 (// (pt-x pt) pt-norm)))
    (2t2 (* 2 (// (pt-y pt) pt-norm)))
    (2t3 (* 2 (// (pt-z pt) pt-norm))))

  (declare (sys:array-register array0))
  (declare (sys:array-register-%id array1 array2))
  ;; computes coefficients b'[i,j,k]
  (loop for i from 0 to p
    do
    (setf (aref array0 i) (expt pt-norm (+ i 1)))
  (loop for j from 0 to (- p i)
    do
    (let ((c (make-array (- array-1 i j)))
      (declare (sys:array-register c))
      (setf (sys:%id-aref array1 (+ i (* array-1 j))) c)
      (loop for k from 0 to (- p i j)
        do
        (setf (aref c k) (phase-1 i j k)))))))

  ;; computes coefficients b[i,j,k]*(i!)*(j!)*(k!): (loop for i from 0 to p
    do
    (loop for j from 0 to (- p i)
do
(let ((c (make-array (- array-l i j))))
 (declare (sys:array-register c))
(setf (sys:%id-aref array2 (+ i (* array-l j))) c)
(loop for k from 0 to (- p i j)
  do
    (setf (aref c k)
      (aref
       (sys:%id-aref fac-table (+ i (* 21 j)))
       k))
    (aref array0 (+ i j k)))))))

array2))))
(defun shifting1 (array delta-pt)
  (labels
    ((conversion i j k array-l array0 array1)
      (loop for l from 0 to i
        sum
        (loop for m from 0 to j
          sum
          (loop for n from 0 to k
            sum
            (* (aref
                (sys:%id-aref
                  array0 (+ 1 (* array-l m)))
                n)
                (aref
                  (sys:%id-aref
                    array1
                    (+ (- i 1) (* array-l (- j m)))
                    (- k n))))))))
  (if array
    (let* ((p (1- (array-dimension-n 1 array)))
      (array-l (1+ p))
      (array0 array)
      (array1 (expansion1 delta-pt p 1)) ;; a'-array
      (array2 (make-array (list array-l array-l)))
      (declare (sys:array-register-id array0 array1 array2))
      (loop for i from 0 to p
        do
        (loop for j from 0 to (- p i)
          do
          (let ((c (make-array (- array-l i j)))
            (declare (sys:array-register c))
            (setf (sys:%id-aref array2 (+ i (* array-l j))) c)
            (loop for k from 0 to (- p i j)
              do
              (setf (aref c k)
                (conversion i j k array-l array0 array1))))))
      array2))))
;;; Pair-list is a list of cons (delta-pt . a-array)

(defun shifting1-1 (pair-list)
  (labels
    ((conversion (i j k array-1 array0 array1)
      (loop for l from 0 to i
        sum
        (loop for m from 0 to j
          sum
          (loop for n from 0 to k
            sum
            (* (aref
                  (sys:%id-aref
                   array0 (+ 1 (* array-1 m))))
               n)
          (aref
           (sys:%id-aref
            array1
            (+ (- i 1) (* array-1 (- j m)))
            (- k n))))))))
    (if pair-list
      (let* ((p (1- (array-dimension-n 1 (cdar pair-list))))
               (array-1 (i+ p))
             ; new pair: (a-array . a'-array)
             (pair-list
              (loop for pair in pair-list
                append
                (list (cons (cdr pair)
                          (expansion1 (car pair) p 1)))))))
      (array2 (make-array (list array-1 array-1))))))

(declare (sys:array-register-id array2))
(loop for i from 0 to p
do
  (loop for j from 0 to (- p i)
do
    (let ((c (make-array (- array-1 i j))))
      (declare (sys:array-register c))
      (setf (sys:%id-aref array2 (+ i (* array-1 j))) c)
      (loop for k from 0 to (- p i j)
do
        (setf (aref c k)
(loop for pair in pair-list
  sum
    (conversion i j k array-1 (car pair)
      (cdr pair)))))))

;;; Shifting Lemma 3.2.2:
;;; it computes c[i,j,k]i!j!k! instead; pair-list is a list of cons
;;; (delta-pt . a-array).

(defun shifting2 (pair-list)
  (labels
    ((conversion (i j k p array0 array1)
      (loop for l = 0 then (1+ l)
        while (<= (+ l i j k) p)
        sum
          (loop for m = 0 then (1+ m)
            while (<= (+ l m i j k) p)
            sum
              (loop for n = 0 then (1+ n)
                while (<= (+ l m n i j k) p)
                sum
                  (* (aref
                      (sys:%1d-aref
                        array0 (+ 1 (* (1+ p) m)))
                    n)
                    (aref
                      (sys:%1d-aref
                        array1
                        (+ (+ l 1) (* (1+ p) (+ j m)))
                        (+ k n))))))))
    (if pair-list
      (let* ((p (1- (array-dimension-n 1 (cdr pair-list))))
        (array-1 (1+ p))
        ; new pair: (a-array . b-array)
        (pair-list
          (loop for pair in pair-list
            append
              (list (cons (cdr pair)
                (expansion2 (car pair) p))))
            (array2 (make-array (list array-1 array-1))))
        (declare (sys:array-register-1d array2))
        (loop for i from 0 to p
          ...)
do
  (loop for j from 0 to (- p i)
    do
      (let ((c (make-array (- array-l i j))))
        (declare (sys:array-register c))
        (setf (sys:|%id-aref array2 (+ i (* array-l j)))) c)
      (loop for k from 0 to (- p i j)
        do
          (setf (aref c k)
            (loop for pair in pair-list
              sum
              (conversion i j k p (car pair)
                (cdr pair))))))))

;;; This computes c[i,j,k] and is used for shiftings at leaf nodes;
;;; pair-list is a list of conses (delta-pt . a-array).

(defun shifting2-1 (pair-list)
  (labels
    ((conversion (i j k p array0 array1)
      (loop for l = 0 then (1+ l)
        while (<= (+ l i j k) p)
        sum
        (loop for m = 0 then (1+ m)
          while (<= (+ l m i j k) p)
          sum
          (loop for n = 0 then (1+ n)
            while (<= (+ l m n i j k) p)
            sum
            (* (aref
                (sys:|%id-aref
                  array0 (+ l (* (1+ p) m)))
              n)
              (aref
                (sys:|%id-aref
                  array1
                  (+ (+ i l) (* (1+ p) (+ j m)))
                (+ k n))))))))
      (if pair-list
        (let* ((p (1- (array-dimension-n 1 (cdar pair-list))))
          (array-l (1+ p)))
          (conversion i j k p array0 array1)
        p))))
(pair-list ; new pair: (a-array b-array)
  (loop for pair in pair-list
    append
    (list (cons (cdr pair)
      (expansion2 (car pair) p))))
  (array2 (make-array (list array-1 array-1))))
(declare (sys:array-register-1d array2))
(loop for i from 0 to p
  do
    (loop for j from 0 to (- p i)
      do
        (list ((c (make-array (- array-1 i j))))
          (declare (sys:array-register c))
          (setf (sys:%id-aref array2 (+ i (* array-1 j))) c)
          (loop for k from 0 to (- p i j)
            do
              (setf
                (aref c k)
                /// (loop for pair in pair-list
                  sum
                  (conversion i j k p (car pair)
                    (cdr pair)))
                (aref (sys:%id-aref fac-table (+ i (* 21 j)))
                  k))))))))

;;; Shifting Lemma 3.2.3:
;;; it computes coefficient d[i,j,k]i!j!k! from c[i,j,k]i!j!k!

(defun shifting3 (array delta-pt)
  (labels
    ((conversion (i j k p delta-x delta-y delta-z)
        (loop for l = 0 then (i+ l)
          while (<= (+ 1 i j k) p)
          sum
          (loop for m = 0 then (i+ m)
            while (<= (+ 1 m i j k) p)
            sum
            (loop for n = 0 then (i+ n)
              while (<= (+ 1 m n i j k) p)
              sum
              ///
              (* (aref

```
(sys: %id-arref
  array
  (+ (+ i l) (* (i+ p) (+ j m)))
  (+ k n))
(expt delta-x l)
(expt delta-y m)
(expt delta-z n))
(aref
 (sys: %id-arref
  fac-table (+ 1 (* 21 m))) n)))))))

(if array
  (let* ((delta-x (pt-x delta-pt))
         (delta-y (pt-y delta-pt))
         (delta-z (pt-z delta-pt))
         (p (i- (array-dimension-n 1 array)))
         (array-l (i+ p))
         (array1 (make-array (list array-l array-l)))
         (declare (sys: array-register-id array1))
         (loop for i from 0 to p
do
          (loop for j from 0 to (- p i)
             do
              (let ((c (make-array (- array-l i j))))
                 (declare (sys: array-register c))
                 (setf (sys: %id-arref array1 (+ i (* array-l j))) c)
                 (loop for k from 0 to (- p i j)
                    do
                      (setf
 (aref c k)
                        (conversion i j k p delta-x delta-y delta-z)))))))
array1))))))

;;; it computes coefficient d[i,j,k]i!j!k! from c[i,j,k]i!j!k!

(defun shifting3-1 (array delta-pt)
  (labels
    ((conversion (i j k p delta-x delta-y delta-z)
     (loop for l = 0 then (i+ l)
        while (<= (+ 1 i j k) p)
        sum
        (loop for m = 0 then (i+ m)
           while (<= (+ 1 m i j k) p)
           sum
})
(loop for n = 0 then (i n)
while (<= (+ l m n i j k) p)
sum

  ;;
  ;; (setf (sys:lambda @ (lambda x y z)
  ;;                  (+ x y z)))
  ;; (declare (sys:array-register-id array)
  ;;          (sys:array-register-id array1)
  ;;          (sys:array-register-id array2)
  ;;          (sys:array-register-id array3))
  ;; (loop for i from 0 to p
do
  ;; (loop for j from 0 to (- p i)
do
  ;; (let ((x (- y z)))
  ;;       (declare (sys:array-register-id x))
  ;;       (setf (sys:array-register-id x) (+ x y z))
  ;;       (loop for k from 0 to (- p i j)
do
  ;;             (setf (sys:array-register-id k)
  ;;                   (+ x y z)))
  ;;       (array1))))

;;; Evaluate the resulted multipole expansion at point pt.
(defun evaluate-at-pt (pt array)
  (if array
      (let* ((array array)
              (p (i- (array-dimension-n 1 array)))
              (array-1 (i+ p))
              (x (pt-x pt))
              (y (pt-y pt))
              (z (pt-z pt)))
        (declare (sys:array-register-1d array))
        (loop for i from 0 to p
              for temp2 =
                (loop for j from 0 to (- p i)
                      for temp1 =
                        (loop for k from 0 to (- p i j)
                              for temp =
                                (* (expt x i) (expt y j)
                                   (expt z k))
                              for d =
                                (aref (sys:%id-aref
                                       array
                                       (+ i (* array-1 j)))
                                       k)
                              when (< (+ i j k) p)
                          sum
                            (* (1+ i)
                               (aref (sys:%id-aref
                                      array
                                      (+ i 1 (* array-1 j)))
                                      k)
                               temp)
                          into ax
                          and sum
                            (* (1+ j)
                               (aref (sys:%id-aref
                                      array
                                      (+ i
                                         (* array-1 (1+ j)))
                                      k)
                               temp)
                          into ay
                          and sum
                            (* (1+ k))
(aref
(sys:%id-aref
  array
  (+ i (* array-1 j)))
(1+ k))
temp)
  into az
  sum (* d temp) into po
finally (return (list ar ay az po)))
sum (first temp1) into ax
sum (second temp1) into ay
sum (third temp1) into az
sum (fourth temp1) into po
finally (return (list ax ay az po)))
sum (first temp2) into ax
sum (second temp2) into ay
sum (third temp2) into az
sum (fourth temp2) into po
finally (return (list ax ay az po)))
(list 0 0 0 0)))
APPENDIX A. SEQUENTIAL CODE

;;; Constructing 3-D Tree

;;; The data structure is a 3-D tree. In the implementation, nodes
;;; at level x represent cubes at various levels of the spatial
;;; refinement, while those at levels y and z are dummy nodes for
;;; brunching. Each node maintains an item-list, which is a
;;; list of lists:
;;;   ((pos vel acc mass true-acc) ... ),
;;; a PHI slot and a PSI slot for storing arrays of coefficients of
;;; the multipole expansions.

;;; The following procedure takes as arguments the height of 3-D tree
;;; (the total level of the spatial decomposition), and the
;;; lower-left-front vertex and up-right-behind vertex of the original
;;; cube to start with. It returns the 3-D tree constructed.

(defmethod (3D-tree :build-tree) (height pt1 pt2)
  (labels
    ((build-tree-internal (level pt1 pt2)
      (let*
        ((key (pt-average pt1 pt2))
         (diagonal (pt-difference pt2 pt1))
         (length (abs (pt-x diagonal)))
         (li (list (- length) 0 length))
         (neighbor-list
          (loop for x in li
            append
            (loop for y in li
              append
              (loop for z in li
                append
                (if (within-box?
                  (pt-plus key (pt x y z))
                  lower-left-front-vertex
                  upper-right-behind-vertex)
                (list
                  (pt-plus key (pt x y z)))))
                into n-list
              finally (return (del #'pt-equal key n-list))))))
      (send self :insert nil key neighbor-list))))
; insert two additional levels of nodes: y level and z level
(cond
  ((= level height) nil)
  (t
   (let*
    (dummy-key-y1 (pt ( + (pt-x pt1) (pt-x key)) 2)
                 (pt-y key) (pt-z key)))
    (dummy-key-y2 (pt ( + (pt-x pt2) (pt-x key)) 2)
                 (pt-y key) (pt-z key)))
    (dummy-key-z1 (pt (pt-x dummy-key-y1)
                      ( + (pt-y pt1) (pt-y key)) 2)
                 (pt-z key)))
    (dummy-key-z2 (pt (pt-x dummy-key-y1)
                      ( + (pt-y pt2) (pt-y key)) 2)
                 (pt-z key)))
    (dummy-key-z3 (pt (pt-x dummy-key-y2)
                      ( + (pt-y pt1) (pt-y key)) 2)
                 (pt-z key)))
    (dummy-key-z4 (pt (pt-x dummy-key-y2)
                      ( + (pt-y pt2) (pt-y key)) 2)
                 (pt-z key)))
    (dummies (list dummy-key-y1 dummy-key-y2 dummy-key-z1
           dummy-key-z2 dummy-key-z3 dummy-key-z4))
    (half-diagonal (pt-scalar-times 0.5 diagonal))
    (choices
     (list 0 ( // length 2)))))

; put dummy nodes on y and z levels
(loop for *pt* in dummies
   do (send self :insert nil *pt*))

(loop for x in choices
  do
    (loop for y in choices
      do
        (loop for z in choices
          do
            (let ((base-pt (pt-plus pt1 (pt x y z)))
                  (build-tree-internal
                   (1+ level) base-pt
                   (pt-plus base-pt half-diagonal)))))))))

(let ((pt1 (pt (float (pt-x pt1)) (float (pt-y pt1)))))
(float (pt-z pt1)))
(pt2 (pt (float (pt-x pt2)) (float (pt-y pt2))
   (float (pt-z pt2))))
(setq height-of-tree height
   lower-left-front-vertex pt1
   upper-right-behind-vertex pt2
   half-leaf-diag-vec
   (pt-scalar-times
      (/* 1.0 (expt 2 (+ height-of-tree 1)))
      (pt-difference pt2 pt1)))
(build-tree-internal
   0 lower-left-front-vertex upper-right-behind-vertex)
))
;;; This procedure finds for each node its near-field and interactive-
;;; field.

(defmethod (3D-tree :find-neighbor-and-interaction) ()
(labels
  ((internal-find (node parent)
    (when node
      (cond
        ((= (3D-node-sort-by-y node) 0)
          (when parent
            (let* ((key (3D-node-key node))
                       (parent-near-field-list
                        (3D-node-near-field-list parent))
                       (dist
                        (* 5.5
                           (abs (- (pt-x key)
                                   (pt-x (3D-node-key parent)))))))
               (setf
                (3D-node-interaction-list node))
               (loop for p-near-field in parent-near-field-list
                 for sons =
                 (loop for dummy-son in
                   (list (3D-node-left p-near-field)
                         (3D-node-right p-near-field))
                   for sons2 =
                   (loop for dummy-grandson in
                     (list (3D-node-left dummy-son)
                           (3D-node-right dummy-son))
                     for sons3 =
                     (loop for child in
                       (list (3D-node-left dummy-grandson)
                             (3D-node-right dummy-grandson))
                       append
                       (if child
                         (if (> (distance-between-points
                           key (3D-node-key child))
                             dist)
                           (list child)
                           (progn
                           (setf
                            (3D-node-near-field-list node)
                            (cons child (3D-node-near-field-list node)))
                           nil))))))

APPENDIX A. SEQUENTIAL CODE
append (if (= (length sons3) 2)
  (list dummy-grandson) sons3))

append
  (if (and (= (length sons2) 2)
    (= (3D-node-sort-by-y (first sons2))
      2))
    (= (3D-node-sort-by-y
        (second sons2))
      2))
  (list dummy-son) sons2)
append (if (and (= (length sons) 2)
  (= (3D-node-sort-by-y
        (first sons))
    1)
    (= (3D-node-sort-by-y
        (second sons))
    1))
  (list p-near-field sons)))

;;; coloring all the nodes on the interaction list.
(loop for node1 in (3D-node-interaction-list node)
  do (setf (3D-node-mark-bit node1) t))
  (internal-find (3D-node-left node) node)
  (internal-find (3D-node-right node) node))
(t (internal-find (3D-node-left node) parent)
  (internal-find (3D-node-right node) parent)))
(setf (3D-node-near-field-list (3D-node-right root))
  (list (3D-node-right root)))
  (internal-find (3D-node-right root) nil)))

;;; insert particles into the 3-D tree

(defun (3D-tree :insert-points) (number-of-particles)
  (labels
    ((random-vec ()
      (loop for vec =
        (pt-plus
          (pt -1 -1 -1)
          (pt (// (random 10000001) 500000.0)
            (// (random 10000001) 500000.0)
            (// (random 10000001) 500000.0)))
        until (< (vector-norm-squared vec) 1.0))
    )))

(defun (3D-tree :insert-points) (number-of-particles)
  (labels
    ((random-vec ()
      (loop for vec =
        (pt-plus
          (pt -1 -1 -1)
          (pt (// (random 10000001) 500000.0)
            (// (random 10000001) 500000.0)
            (// (random 10000001) 500000.0)))
        until (< (vector-norm-squared vec) 1.0))
    )))
finally (return vec))))
(loop for mass = (// 1.0 number-of-particals)
  for i below number-of-particals
  for pos = (random-vec)
  for vel = (random-vec)
    ;; now item list is a list of: (pos vel acc mass true-acc).
    do (setf (3D-node-near-field-list root)
              (cons (list pos vel nil mass nil)
                    (3D-node-near-field-list root)))))))

(defmethod (3D-tree :clean-and-reinsert-points) ()
(labels ((map-over (node)
              (cond ((null node) nil)
                 (t (when (3D-node-item-list node)
                      (setf (3D-node-item-list node) nil))
                      (progn (map-over (3D-node-left node))
                             (map-over (3D-node-right node)))))))
  (setq number-of-items 0)
  (map-over root)
  (let* ((ll (length (3D-node-near-field-list root)))
          (mass (// 1.0 ll)))
    (setq total-mass 1.0)
    (loop for item1 in (3D-node-near-field-list root)
       for item = (append (copytree item1) (list mass))
       do (send self :insert item (first item)))))
APPENDIX A. SEQUENTIAL CODE

;;; -----------------------------------------------
;;; Multiple expansion Method
;;; -----------------------------------------------

;;; Array+ takes as arguments two arrays of the same size, and returns
;;; an array of which each element is the sum of the corresponding
;;; ones of two arrays in the arguments. If one of the argument is nil,
;;; it returns the other non-nil argument. If both are nil's, it
;;; returns nil.

(defun array+ (array1 array2)
  (when (or array1 array2)
    (cond
      ((null array2) array1)
      ((null array1) array2)
      (t
       (let* (((p* (array-dimension-n 1 (if array1 array1 array2)))
                (p (1- p*))
                (array (make-array (list p* p*)))))
         (declare (sys:array-register-id array1 array2 array))
         (loop for i from 0 to p
do
           (loop for j from 0 to (- p i)
do
             (let ((c (make-array (- p* i j))))
              (declare (sys:array-register c))
              (setf (sys:%id-aref array (+ i (* p* j))) c)
              (loop for k from 0 to (- p i j)
do
                (setf
                 (aref c k)
                 (+ (aref (sys:%id-aref array (+ i (* p* j))) k)
                    (aref (sys:%id-aref array2 (+ i (* p* j))) k)))
               array))))))

;;; List+ takes two lists as its arguments and returns a list in which
;;; each element is the sum of the correspondent elements in argument
;;; lists. the length is that of the shorter one in the arguments.

(defun list+ (list1 list2)
(loop for i in list1
     for j in list2
     collect (+ i j)))

(defun list-@* (list1 list2)
  (loop for i in list1
     for j in list2
     collect (* i j)))

(defun list-scale (list x)
  (loop for i in list
     collect (* i x)))

;;; a procedure computes PHI for each node in the upward-path of the
;;; algorithm

(defun (3D-tree :upward-path) ()
  (labels
    ((calculate-phi-internal (node)
       (cond
         ((null node) nil)
         ((3D-node-item-list node)
           (let* ((pair-list
                       (loop for i in (3D-node-item-list node)
                         append
                         (list (cons (pt-difference
                                      (first i) (3d-node-key node))
                               (fourth i))))))
            (accum (expansion-i i precision-p pair-list)))
          (setf (3D-node-phi node) accum)
          (list (cons (3D-node-key node) accum))))
        (t (let* ((pair-list
                   (append
                    (calculate-phi-internal (3D-node-left node))
                    (calculate-phi-internal (3D-node-right node)))))
            result))
      (cond
        (; is the node on some interaction list?
         (and (3D-node-mark-bit node)
              pair-list)
        ; shift result to the cube's center
        (setq pair-list
              (loop for i in pair-list
                 ...))

      ...
APPENDIX A. SEQUENTIAL CODE

append
   (list (cons (pt-difference
                (car i) (3d-node-key node))
           (cdr i)))))
    (setq result (shifting1-1 pair-list))
    (setq (3D-node-phi node) result)
    (list (cons (3D-node-key node) result)))
    (t (setq (3D-node-phi node) (if pair-list t)
           pair-list)))))))
   (calculate-phi-internal (3D-node-right root)))

;;; a procedure computes PSI for each node in the downward-path of
;;; the algorithm and evaluates PSI at particle positions

(defmethod (3D-tree :downward-path) ()
     (labels
     ((final-evaluation (node)
         (let ((psi (3D-node-psi node))
                (key (3D-node-key node))
                (near-field-item-list
                    (loop for near-field-neighbor in
                           (3D-node-near-field-list node)
                           append (3D-node-item-list near-field-neighbor)))))
        (loop for i in (3D-node-item-list node)
              for list1 = (list+ (evaluate-at-pt
                                  (pt-difference (first i) key) psi)
                                  (loop for j in near-field-item-list
                                          for temp =
                                          (direct-evaluate-at-pt
                                           (first i) (first j) (fourth j)
                                           softening-parameter)
                                          sum (first temp) into ax
                                          sum (second temp) into ay
                                          sum (third temp) into az
                                          sum (fourth temp) into po
                                          finally (return (list ax ay az po))))
              do (setq CM-position
                   (pt-plus
                    CM-position
                    (pt-scalar-times (fourth i) (first i)))))
              (setq energy-U
                    (+ energy-U (* 0.5 (fourth i) (fourth list1))))
              (setf (third i)
(pt (first list1) (second list1) (third list1)))

))
(calculate-psi-internal (node parent-psi parent-key level)
 (if (or (null node) (null (3D-node-psi node)))
 nil
 (if (= (3D-node-sort-by-y node) 0)
 (let* (((key (3D-node-key node))
         (tilde-psi parent-psi)
         (sum-of-phis-on-interactive-list
          (loop for node1 in
                (3D-node-interaction-list node)
                append
                (if (3D-node-psi node1)
                  (list (cons (pt-difference
                               (3D-node-key node1)
                               key)
                         (3D-node-psi node1)))))))
      (return
       ; shift phi to the center of new cube
       (if (= level (* 3 height-of-tree))
          (shifting2-1 pair-list)
          (shifting2 pair-list)))))

; shift psi of the parent node to the center of
; its son, and form psi for the son
(psi
  (if parent-key
    (array+ (if (= level (* 3 height-of-tree))
             (shifting3-1
              tilde-psi
              (pt-difference key parent-key))
             (shifting3
              tilde-psi
              (pt-difference key parent-key)))
             sum-of-phis-on-interactive-list))))
(setf (3D-node-psi node) psi)
(cond ((= level (* 3 height-of-tree))
    ; final evaluation at the point.
    ; the result is the sum of direct evaluation
    ; and multipole expansion evaluation
    (final-evaluation node))
  (t
(calculate-psi-internal
 (3D-node-left node) psi key (1+ level))
 (calculate-psi-internal
 (3D-node-right node) psi key (1+ level))))

(progn
 (calculate-psi-internal
 (3D-node-left node) parent-psi parent-key (1+ level))
 (calculate-psi-internal
 (3D-node-right node) parent-psi parent-key
 (1+ level)))))

(if (3D-node-right root)
 (calculate-psi-internal (3D-node-right root) nil nil 0))))
APPENDIX A. SEQUENTIAL CODE

;;;==========================================================
;;; Direct Method
;;;==========================================================

;;; compute potential at pt due to mass at pt1

(defun direct-evaluate-at-pt (pt pt1 mass softening)
  (if (pt-same pt pt1)
      (list 0 0 0 0)
      (let* (;;;square of dist
            (dd (+ (pt-dist
                   (pt-x pt) (pt-y pt) (pt-z pt) (pt-x pt1)
                   (pt-y pt1) (pt-z pt1))
                   (* softening softening)))
            (d (sqrt dd))
            (tx (/ (- (pt-x pt1) (pt-x pt)) dd))
            (ty (/ (- (pt-y pt1) (pt-y pt)) dd))
            (tz (/ (- (pt-z pt1) (pt-z pt)) dd))
            (po (/ mass d)))
      (list (* tx po) (* ty po) (* tz po) po))))

;;; compute potential at pt due to all the other masses

(defmethod (3D-tree :direct-evaluation-at-this-point) (pt)
  (labels ((sum-over (node))
            (cond ((null node) (list 0 0 0 0))
                  ((3D-node-item-list node)
                   (loop for i in (3D-node-item-list node)
                         for temp =
                         (direct-evaluate-at-pt
                          pt (first i) (fourth i)
                          softening-parameter)
                         sum (first temp) into ax
                         sum (second temp) into ay
                         sum (third temp) into az
                         sum (fourth temp) into po
                         finally (return (list ax ay az po)))))
            (t
             (list+ (sum-over (3D-node-left node))
                    (sum-over (3D-node-right node)))))
      (sum-over (3D-node-right root))))
;;; compute all the pair-wise interactions

(defmethod (3D-tree :direct-evaluation) ()
  (labels
    ((map-over (node)
      (when (not (null node))
        (cond
          ((3D-node-item-list node)
            (loop for i in (3D-node-item-list node)
              for list1 =
                (send self :direct-evaluation-at-this-point
                  (car i))
              do
                (setq true-energy-U
                  (- true-energy-U (* 0.5 (fourth i) (fourth list1))))
                (setq (fifth i)
                  (pt (first list1) (second list1) (third list1))))
            (t (map-over (3D-node-left node))
              (map-over (3D-node-right node))))))))
  (map-over (3D-node-right root))))
;;; Error calculation

;;; Compare the result of the multipole expansion method with that of
;;; the direct method, and compute the maximum errors and
;;; root-mean-square errors.

(defun (3D-tree :error-calculation) ()
  (labels
    ((tree-walk (node))
      (cond
        ((null node) (values -1 0))
        ((3D-node-item-list node)
          (loop for item in (3D-node-item-list node)
            for a-diff = (pt-difference (third item) (fifth item))
            for a-sum-norm = (+ (vector-norm (third item))
                                (vector-norm (fifth item)))
            for relative-error =
              (/ (* 2 (vector-norm a-diff)) a-sum-norm)
            maximize relative-error into e1
            sum (* relative-error relative-error) into e2
do
            (format t "Position = ("D, "D, "D), "
                    (pt-x (first item)) (pt-y (first item))
                    (pt-z (first item)))
            (format t "Mass = "D," (fourth item))
            (format t "Calculated Acc = ("D, "D, "D),"
                    (pt-x (third item)) (pt-y (third item))
                    (pt-z (third item)))
            (format t "True Acc = ("D, "D, "D),"
                    (pt-x (fifth item)) (pt-y (fifth item))
                    (pt-z (fifth item)))
            (format t "Acc Difference = ("D, "D, "D),"
                    (pt-x a-diff) (pt-y a-diff) (pt-z a-diff))
          finally (return (values e1 e2)))))
    (tree-walk (3D-tree-left node))
    (multiple-value-bind (max1 accum1)
      (3D-tree-left node))
    (multiple-value-bind (max2 accum2)
      (3D-tree-right node))
    (values (max max1 max2) (+ accum1 accum2)))))
  (tree-walk (3D-tree-right root))))
Appendix B

Parallel Code

This appendix contains a parallel code implementing the three-dimensional N-body algorithm on the Connection Machine. The code is written in *Lisp.

;;;; -e- Package: *LISP; Base: 10; Syntax: Common-lisp -e-

(IN-PACKAGE "*LISP")

;;;; This version of parallel implementation runs on CM-2, which
;;;; has floating point hardware and 64K bit local memory.

;;;; ===============================
;;;; POINTS
;;;; ===============================

;;;; This does the function typing:
(*proclaim '(ftype (function (t) float-pvar)
    pt-x pt-y pt-z vector-norm-squared!! vector-norm!!))

(defstruct (pt :named :conc-name (:constructor pt (x y z))
    x y z)

(defun printable-point (pt)
    '(pt , (pt-x pt), (pt-y pt), (pt-z pt)))

(*defun pt-zero-p!! (pt)
    (and!! (zerop!! (pt-x pt)) (zerop!! (pt-y pt)) (zerop!! (pt-z pt))))
(*/defun pt-equal!! (p1 p2)
  (and!! (=!! (pt-x p1) (pt-x p2))
    (=!! (pt-y p1) (pt-y p2))
    (=!! (pt-z p1) (pt-z p2))))
)

(defun pt-plus (p1 p2)
  (pt (+ (pt-x p1) (pt-x p2))
    (+ (pt-y p1) (pt-y p2))
    (+ (pt-z p1) (pt-z p2))))

;;; the square of the Euclidean distance
(*defun pt-dist!! (p1 p2)
  (*let ((delta-x (-!! (pt-x p1) (pt-x p2)))
    (delta-y (-!! (pt-y p1) (pt-y p2)))
    (delta-z (-!! (pt-z p1) (pt-z p2))))
    (+!! (*!! delta-x delta-x) (*!! delta-y delta-y) (*!! delta-z delta-z))))

(*defun distance-between-points!! (p1 p2)
  (sqrt!! (pt-dist!! p1 p2)))

;;; ------- Thinking of Points as Vectors -------

(*defun vector-norm-squared!! (pt)
  (let ((x (pt-x pt))
    (y (pt-y pt))
    (z (pt-z pt)))
    (+!! (*!! x x) (*!! y y) (*!! z z))))

(*defun vector-norm!! (pt)
  (sqrt!! (vector-norm-squared!! pt)))

;;; ------- Parallel Exponentiation -------

(*proclaim 'ftype (function (t) float-pvar) expt!!))

(*defun expt!! (pvari expos) ; pvari: float-pvar; expos: integer
  (declare (type float-pvar pvari))
  (let ((result (!! 1)))
    (declare (type float-pvar result))
    (loop for i below expos
      do (*set result (*!! pvari result))
    finally (return result)))
  )
APPENDIX B. PARALLEL CODE

;;; ===================================================================
;;; Global Variables
;;; ===================================================================

(*proclaim '(type (signed-pvar 4) level))

(*proclaim '(type (field-pvar CM:*CUBE-ADDRESS-LENGTH*) parent-pro-add))

(*proclaim '(type (float-pvar 23 8) item-list potential))

(defun p)

(defun height-of-tree)

(defun number-of-particles 0)

;;; breadth-first scan label of the last tree node
(defun last-node-add)

;;; address of the last processor allocated for tree node
(defun last-pro-add)

;;; address of the last processor allocated for particles
(defun last-leaf-add)

(defun average-error 0)

(defun max-error 0)

(defun phi) ;array of pvars holding PHI's

(defun psi) ;array of pvars holding PSI's

;;; array of pvars holding PHI-template's and PSI-template's.
(defun array-template)

;;; array of pvars temporarily storing PHI's of a node's interactive-
;;; fields.
(defun array-template1)

(defun p-key) ;(pt x y z) where x y z are pvars.

(*defvar item-list (!! 0)) ;a pvar holding mass of each particle
APPENDIX B. PARALLEL CODE

(*defvar parent-pro-add (!! 0))

;;; a pvar whose component indicates the level a node is on. By
;;; convention, the root is at level 0. A leaf node is at level -1,
;;; just for the purpose of distinguishing itself from the rest.
(*defvar level (!! 0))

(*defvar potential (!! 0))

;;; list of pvars dynamically created holding the addresses of particles
;;; entering each node of the tree.
(defvar son-add-list nil)

;;; an array of pvars that has 81 entries of near-field
(defvar near-field-list)

;;; an array of pvars that has 140 entries of interactive-field
(defvar interaction-list)
Math Procedures

;i; Procedures for computing permutation: ( ).
;i; ;j
;i; Using the property: (i choose j)=(i choose (i-j)),
;i; which is valid when i is a nonnegative integer!

(defun perm (i j)
 (let ((result 1)
       (j (min j (- i j))))
 (do ((m i (1- m))
     (n i (1+ n)))
    (> n j) (float result))
 (setq result (* (/ m n) result))))

;i; Procedure for computing the factorial i!.

(defun factorial (i)
 (labels ((factorial-internal (n)
           (if (< n 2)
               i
               (* n (factorial-internal (1- n)))))
            (float (factorial-internal i))))

;i; Storing the products of factorials i!, j! and k! in a table.

(defun make-fac-table (p)
 (let ((fac-table-internal (make-array (list (1+ p) (1+ p) (1+ p))))
       (loop for i from 0 to p
             do (loop for j from 0 to (- p i)
                do (loop for k from 0 to (- p i j)
                  do (setf (aref fac-table-internal i j k)
                         (* (factorial i)
                            (factorial j)
                            (factorial k))))))
     fac-table-internal))

(defvar fac-table (make-fac-table 20))
;;; The table for [(-0.5)*(-1.5)*(-2.5)...*(-0.5-i+1)]:

(defun make-fac-table-1 (p)
  (labels ((fac-1 (i)
    (let ((result 1.0e0)) ;; i -> 1.0e0
      (do ((m -0.5 (1- m))
           (n 1 (1+ n)))
          (> n i) result)
        (setq result (* m result))))))
  (let ((fac-table-internal-1 (make-array (1+ p))))
    (loop for i from 0 to p
e          do (setf (aref fac-table-internal-1 i) (fac-1 i)))
    fac-table-internal-1)))

(defvar fac-table-1 (make-fac-table-1 20))
APPENDIX B. PARALLEL CODE

;;; ==============================================================
;;; Expansion Theorems
;;; ==============================================================

;;; Theorem 3.2.1

(*proclaim '(ftype (function (t) float-pvar) coeff!! phase-2))

(*defun coeff!! (i j k pt)
  (let ((y1 (pt-x pt))
     (y2 (pt-y pt))
     (y3 (pt-z pt))
     (/!! (+!! (expt!! (-!! y1) i) (expt!! (-!! y2) j) (expt!! (-!! y3) k))
      (the float-pvar (!! (aref fac-table i j k)))))

(*defun expansion!! (pt) ; resulted array of pvar's put in array-template.
  (loop for i from 0 to p
    do (loop for j from 0 to (- p i)
      do (loop for k from 0 to (- p i j)
        do (*set (the float-pvar
                    (aref array-template i j k))
             (coeff!! i j k pt))))))

;;; Theorem 3.2.2

;;; Procedure computing coefficient b'[i,j,k]
(defun phase-1 (i j k)
  (/ (aref fac-table-1 (+ i j k)) (aref fac-table i j k)))

;;; Procedure computing coefficient b''[i,j,k]
(*defun phase-2 (i j k array 2t1 2t2 2t3)
  (*let ((template3 (!! 0)))
    (declare (type float-pvar template3))
    (loop for l from 0 to (floor (/ i 2))
      do (loop for m from 0 to (floor (/ j 2))
        do (loop for n from 0 to (floor (/ k 2))
          do (*set template3
             (+!!
              template3
            )
        )))
  ))
(*!! (the float-pvar
   (!! (aref array (- i l) (- j m) (- k n))
    (perm (- i l) l)
    (perm (- j m) m)
    (perm (- k n) n))())
  (expt!! 2t1 (- i (* 2 l)))
  (expt!! 2t2 (- j (* 2 m)))
  (expt!! 2t3 (- k (* 2 n))))))))
finally (return (*!! (!! (expt -1 (+ i j k))) template3)))))

;;;; b[i,j,k]*(i!)*(j!)*(k!) is computed here
(*defun expansion2!! (pt) ; results put in array-template.
 (let* ((array-1 (1+ p))
    (array0 (make-array array-1)) ;for 1/r^-(i+j+k+1).
    ;for b'[i,j,k]. will be !+-ized later.
    (array1 (make-array (list array-1 array-1 array-1)))
    (*let* ((pt-norm (vector-norm!! pt))
      (2t1 (*!! (!! 2) (!! (pt-x pt) pt-norm)))
      (2t2 (*!! (!! 2) (!! (pt-y pt) pt-norm)))
      (2t3 (*!! (!! 2) (!! (pt-z pt) pt-norm)))
    (declare (type float-pvar pt-norm 2t1 2t2 2t3))
    ; computing coefficients b'[i,j,k].
    (loop for i from 0 to p
do (setf (aref array0 !:)
      (allocate!! (expt!! pt-norm (+ i 1))
      nil '(type float-pvar)))
    (loop for j from 0 to (- p i)
do (loop for k from 0 to (- p i j)
do (setf (aref array1 i j k)
      (phase-1 i j k))))))

;;;; computing coefficients b[i,j,k]*(i!)*(j!)*(k!)
(declare (type float-pvar array-template i j k))
(allocate!! (aref array-template i j k)
  nil '(type float-pvar))
(declare (type float-pvar array0 (+ i j k)))
(expt!! (phase-2 i j k array1 2t1 2t2 2t3)
  (!! (aref facetable i j k)))
(expt!! (expt -1 (+ i j k)))
(expt!! (expt -1 (+ i j k))))))

(loop for i from 0 to p
do (*deallocate (aref array0 i))))
APPENDIX B. PARALLEL CODE

;;; ===============================================================================
;;; Shifting lemmas
;;; ===============================================================================

;;; Lemma 3.2.1:

(*proclaim '(ftype (function (t) float-pvar)
             conversion1!! conversion2!! conversion3!!))

(*defun conversion1!! (i j k)
  (*let ((template3 (!! 0)))
    (declare (type float-pvar template3))
    (loop for l from 0 to i
      do
      (loop for m from 0 to j
        do
        (loop for n from 0 to k
          do
          (**set template3
           (**!! template3
            (**!! (the float-pvar
                   (aref phi 1 m n))
            (aref phi 1 m n))
            (aref array-template (~ i l) (~ j m) (~ k n))))
        finally (return template3)))))

(*defun shifting1!! ()
  (*let ((parent-add-pvar (floor!! (self-address!!) (!! i)) (!! 8)))
    (declare (type (field-pvar CW:*CUBE-ADDRESS-LENGTH*) parent-add-pvar))
    (**when (not!! (zerop!! (the float-pvar (aref phi 0 0 0))))
      (**let ((d-x (neg!! (pt-x p-key) (pref!! (pt-x p-key) parent-add-pvar))))
      (d-y (neg!! (pt-y p-key) (pref!! (pt-y p-key) parent-add-pvar))))
      (d-z (neg!! (pt-z p-key) (pref!! (pt-z p-key) parent-add-pvar))))
    (declare (type float-pvar d-x d-y d-z))
    ; calculate a' array and put results in array-template.
    (expansion!! (pt d-x d-y d-z))
    (loop for i from 0 to p
      do (loop for j from 0 to (- p i)
        do (loop for k from 0 to (- p i j)
          do (**set :add (conversion1!! i j k)
                 (the float-pvar (aref phi i j k))
        parent-add-pvar))))))))
(*defun leaf-expansion!! () ; item-list has charge-pvar.
  (*all
   (*when (not!! (zerop!! item-list)))
     (*let ((d-x (x item-list))
           (d-y (y item-list))
           (d-z (z item-list))
           (d (d item-list)))
      (declare (type float-pvar d x y z))
      (loop for i from 0 to p
            do (loop for j from 0 to (- p i)
                 do (loop for k from 0 to (- i j)
                      do
                          (*set :add
                             (/*! item-list
                                (coeff!! i j k (pt d-x d-y d-z)))
                             (the float-pvar (aref phi i j k))
                             (parent-pro-add)))))
    )))

;;; Lemma 3.2.2:

(*defun conversion2!! (i j k interaction-pvar)
 (declare (type self-adr-pvar interaction-pvar))
 (*let ((template3 (! 0))
         (declare (type float-pvar template3))
         (loop for l = 0 then (1+ l)
               while (< (+ l i j k) (1+ p))
               do
               (loop for m = 0 then (1+ m)
                     while (< (+ l m i j k) (1+ p))
                     do
                     (loop for n = 0 then (1+ n)
                           while (< (+ l m n i j k) (1+ p))
                           do
                           (*set template3
                              (/*! template3
                                 (*!!
                                  (the float-pvar
                                   (aref array-template1 l m n))
                                  (the float-pvar
                                   (aref array-template (+ i 1) (+ j m) (+ k n)))))))))
      finally (return template3)))))
(defun shifting2!! ()
  (declare (type self-adr-pvar interaction-pvar))
  (loop for q from 0 to 139 ; iterating over interaction-list.
      for interaction-pvar = (aref interaction-list q)
      while (not (zerop (*sum interaction-pvar)))
      do
        (*when (and!! (not!! (zerop!! interaction-pvar))
                 (not!! (zerop!! (the float-pvar (aref phi 0 0 0))))
                 (not!! (zerop!! (pref!!
                                 (the float-pvar (aref phi 0 0 0))
                         interaction-pvar))))
        (*let ((d-x (-!! (pref!! (pt-x p-key) interaction-pvar)
                        (pt-x p-key))))
             (d-y (-!! (pref!! (pt-y p-key) interaction-pvar)
                     (pt-y p-key))))
             (d-z (-!! (pref!! (pt-z p-key) interaction-pvar)
                   (pt-z p-key))))
             (declare (type float-pvar d-x d-y d-z))
             (expansion2!! (pt d-x d-y d-z)) ;b[i,j,k] in array-template.

      (loop for i from 0 to p
do
        (loop for j from 0 to (~ p i)
do
          (loop for k from 0 to (~ p i j)
do
            (*set (the float-pvar (aref array-template i j k))
                   (pref!! (the float-pvar (aref phi i j k))
                     interaction-pvar))))))

      (loop for i from 0 to p
do
        (loop for j from 0 to (~ p i)
do
          (loop for k from 0 to (~ p i j)
do
            (*set (the float-pvar (aref psi i j k))
                  (+!! (the float-pvar (aref psi i j k))
                       (/!! (conversion2!! i j k interaction-pvar)
                            (the float-pvar
                             (!! (aref fac-table i j k)))))))))}
APPENDIX B. PARALLEL CODE

;;; Lemma 3.2.3:

(*defun conversion3!! (i j k d-x d-y d-z parent-add-pvar)
 (declare (type self-adr-pvar parent-add-pvar))
 (*let ((template3 (!! 0)))
    (declare (type float-pvar template3))
    (loop for l = 0 then (1+ l)
      while (< (+ l i j k) (1+ p))
      do
    (loop for m = 0 then (1+ m)
      while (< (+ l m i j k) (1+ p))
      do
    (loop for n = 0 then (1+ n)
      while (< (+ l m n i j k) (1+ p))
      do
    (*set
     template3
 (+!! template3
    (/!!
     (*!!
      (pref!!
       (the float-pvar (aref psi (+ i l) (+ j m) (+ k n)))
     parent-add-pvar)
    (the float-pvar
     (!! (aref fac-table (+ i l) (+ j m) (+ k n))))
    (expt!! d-x l)
    (expt!! d-y m)
    (expt!! d-z n))
    (the float-pvar (!! (aref fac-table l m n)))))))
    finally (return template3))))

(*defun shifting3!! ()
 (*let
    ((parent-add-pvar (floor!! (-!! (self-address!!) (!! i)) (!! 8)))
    (d-x (-!! (pt-x p-key) (pref!! (pt-x p-key) parent-add-pvar)))
    (d-y (-!! (pt-y p-key) (pref!! (pt-y p-key) parent-add-pvar)))
    (d-z (-!! (pt-z p-key) (pref!! (pt-z p-key) parent-add-pvar)))
    (select?
     (and!! (not!! (zerop!! (the float-pvar (aref phi 0 0 0))))
     (not!! (zerop!! (pref!! (the float-pvar (aref psi 0 0 0))
     parent-add-pvar)))))))

; results stored in PSI.
(Declare (type (field-pvar CM:*CUBE-ADDRESS-LENGTH*) parent-add-pvar))
(declare (type float-pvar d-x d-y d-z))
(declare (type boolean-pvar select?))

(*when select?
 (loop for i from 0 to p
  do
    (loop for j from 0 to (- p i)
      do
        (loop for k from 0 to (- p i j)
          do
            (*set (the float-pvar (aref psi i j k))
               (/*!! (conversion3!! i j k d-x d-y d-z parent-add-pvar)
                   (the float-pvar (!! (aref fac-table i j k)))))
          )))

;;; point to point direct interaction

(*defun talk!! (x1 y1 z1 item1)
 (declare (type float-pvar x1 y1 z1 item1))
 (*all
  (*when (and!! (not!! (zerop!! item-list))
     (not!! (zerop!!
       (un*compiled-for-now
        "pref!! has a bug"
        (pref!! item1 parent-pro-add))))))
  (*let* ((d-x)
     (d-y)
     (d-z)
     (charge)
     (d-norm))
   (declare (type float-pvar d-x d-y d-z charge d-norm))
   (un*compiled-for-now "pref!! has a bug"
    (*set d-x (-!! (pt-x p-key) (pref!! x1 parent-pro-add))))
    (*set d-y (-!! (pt-y p-key) (pref!! y1 parent-pro-add))))
    (*set d-z (-!! (pt-z p-key) (pref!! z1 parent-pro-add))))
    (*set charge (pref!! item1 parent-pro-add))))
  (*set d-norm
    (sqrt!! (+!! (*!! d-x d-x) (*!! d-y d-y) (*!! d-z d-z))))
  (*set potential
   (+!! potential
    (if!! (zerop!! d-norm)
      (!! 0))
(*defun final-evaluation1!! ()
  (*all
   (*when (not!! (zerop!! item-list))
    (*let ((d-x (-!! (pt-x p-key) (pref!! (pt-x p-key) parent-pro-add)))
           (d-y (-!! (pt-y p-key) (pref!! (pt-y p-key) parent-pro-add)))
           (d-z (-!! (pt-z p-key) (pref!! (pt-z p-key) parent-pro-add)))
           (template3 (!! 0)))
     (declare (type float-pvar d-x d-y d-z template3))
     (loop for i from 0 to p
       do
         (loop for j from 0 to (- p i)
           do
             (loop for k from 0 to (- p i j)
               do
                 (*set template3
                  (++! template3
                   (*!!
                    (pref!! (the float-pvar (aref psi i j k))
                    parent-pro-add)
                    (expt!! d-x i)
                    (expt!! d-y j)
                    (expt!! d-z k))))))
       finally (*set potential template3)))
  )))

(*defun final-evaluation2!! ()
  (declare (type self-adr-pvar son-add near-field))
  (*all (*when (=!! level (!! (the integer height-of-tree)))
    (*let ((near-field-son (!! 0))
      (xi (!! 0))
      (yi (!! 0))
      (zi (!! 0))
      (itemi (!! 0)))
      (loop for i from 0 to 80
        for near-field = (aref near-field-list i)
        while (not (zerop (*sum near-field)))
        do
          (loop for j from 0 to (1- (length son-add-list))
            for son-add = (nth j son-add-list)
            do
              (*let ((near-field-son (!! 0))
                (xi (!! 0))
                (yi (!! 0))
                (zi (!! 0))
                (itemi (!! 0))))
APPENDIX B. PARALLEL CODE

(declare (type self-adr-pvar near-field-son))
(declare (type (float-pvar 23 8) x1 y1 z1 item1))
(*when (not!! (zerop!! near-field))
  (un*compiled-for-now
   "pref!! has a bug"
   (*set near-field-son (pref!! son-add near-field))
   (*when (not!! (zerop!! near-field-son))
    (*set x1 (pref!! (pt-x p-key) near-field-son))
    (*set y1 (pref!! (pt-y p-key) near-field-son))
    (*set z1 (pref!! (pt-z p-key) near-field-son))
    (*set item1 (pref!! item-list near-field-son))))
  (talk!! x1 y1 z1 item1))))

(*defun final-evaluation!! ()
 (final-evaluation1!!)
 (final-evaluation2!!))
### APPENDIX B. PARALLEL CODE

```lisp
;;; ==============================================================
;;; Tree Construction
;;; ==============================================================

;; I use the induced address representation to construct the tree structure.
;; As in the sequential case, the data structure is a 3-D tree (an eightfold-
;; branching tree). If we allocate nodes of the tree sequentially to
;; consecutive processors, starting at the root of the tree, we have a
;; formula to induce the parent-child relationship for each node of the tree.
;; More specifically, given the address i of a node, we know that its parent
;; has address of (\text{floor} ((i - 1)/8)), and its children have addresses of
;; ranging from\((8i - 6)\) to\((8i + 1)\) if the root has address of \text{1, and}
;; ranging from\((8i + 1)\) to\((8i + 8)\) if the root has address of \text{0}.

;;; building the tree, starting at processor 0 (root).

(defvar build-tree (pt1 pt2)
  (labels ((build-tree-internal (*level* node-address key length)
            (when (< node-address (1+ last-node-add))
              (setf (pref (pt-x p-key) node-address) (pt-x key))
              (setf (pref (pt-y p-key) node-address) (pt-y key))
              (setf (pref (pt-z p-key) node-address) (pt-z key))
              (setf (pref level node-address) +level*)
              (let ((choices (list (- length) length))
                  (loop for i in '(0 4)
                      for x in choices
                        do
                          (loop for j in '(0 2)
                              for y in choices
                                do
                                  (loop for k in '(1 2)
                                      for z in choices
                                        do
                                          (let ((new-key (pt-plus key (pt x y z)))
                                              (new-node-address (+ i j k (* 8 node-address))))
                                            (build-tree-internal
                                              (*level* new-node-address new-key
                                               (/ length 2.0))))))
                  (let ((length (/ (abs (- (pt-x pt2) (pt-x pt1))) 4.0))
                    (key (pt (/ (+ (pt-x pt1) (pt-x pt2)) 2.0)
                        (/ (+ (pt-y pt1) (pt-y pt2)) 2.0)
                        (/ (+ (pt-z pt1) (pt-z pt2)) 2.0))
                    (build-tree-internal 0 0 key length)))))
```
;;; this procedure finds the next pvar in son-add-list who has an empty
;;; slot in the processor addressed by <address>, and stores <value> in it;
;;; otherwise appends a new pvar to the list and stores <value>.

(defun set-add (address value)
  (let ((length (length son-add-list)))
    (loop for i below length
      when (zerop
        (pref (the self-adr-pvar (nth i son-add-list)) address))
      do (setf
        (pref (the self-adr-pvar (nth i son-add-list)) address)
        value)
    (return nil)
    finally
      (setq
        son-add-list
        (cons
          (allocate!! (!! 0) nil
            '((type (field-pvar ,CM:*CUBE-ADDRESS-LENGTH*))
              son-add-list))
          (pref (the self-adr-pvar (car son-add-list)) address)
          value)))))

(defun insert (node-address parent-node-address key item)
  (cond
    ((< node-address (i+ last-node-add))
      (let*
        ((processor-key-x (pref (pt-x p-key) node-address))
         (processor-key-y (pref (pt-y p-key) node-address))
         (processor-key-z (pref (pt-z p-key) node-address)))
      (if (not (> (pt-x key) processor-key-x))
        (if (not (> (pt-y key) processor-key-y))
          (if (not (> (pt-z key) processor-key-z))
            (insert (+ 1 (* 8 node-address)) node-address key item)
            (insert (+ 2 (* 8 node-address)) node-address key item))
          (insert (+ 3 (* 8 node-address)) node-address key item))))
APPENDIX B. PARALLEL CODE

(insert (+ 4 (* 8 node-address)) node-address key item))
(if (not (> (pt-y key) processor-key-y))
  (if (not (> (pt-z key) processor-key-z))
    (insert (+ 5 (* 8 node-address)) node-address key item)
    (insert (+ 6 (* 8 node-address)) node-address key item))
  (if (not (> (pt-z key) processor-key-z))
    (insert (+ 7 (* 8 node-address)) node-address key item)
    (insert (+ 8 (* 8 node-address)) node-address key item)))
)
(t (setq last-leaf-add (1+ last-leaf-add))
 (set-add parent-node-address last-leaf-add)
 (setf (pref parent-pro-add last-leaf-add) parent-node-address)
 (setf (pref (pt-x p-key) last-leaf-add) (pt-x key))
 (setf (pref (pt-y p-key) last-leaf-add) (pt-y key))
 (setf (pref (pt-z p-key) last-leaf-add) (pt-z key))
 (setf (pref item-list last-leaf-add) item)))

(defun insert-points (number-of-particles1)
 (setq number-of-particles (+ number-of-particles number-of-particles1))
 (loop for i from 1 to number-of-particles1
   do
   (let ((position (pt-plus (pt -1 -1 -1)
     (pt (/ (random 10000001) 500000.0)
       (/ (random 10000001) 500000.0)
       (/ (random 10000001) 500000.0))))
     (insert 0 0 position (/ 1.0 number-of-particles))))
This procedure finds for each node the near-field and interactive-field. It does use the heuristics of grouping nodes to form supernodes.

(defun find-neighbor-and-interaction-list ()
  (labels
    ((vector-norm (pt)
       (let ((x (pt-x pt))
            (y (pt-y pt))
            (z (pt-z pt)))
        (sqrt (+ (* x x) (* y y) (* z z))))
    ; put <value-list> onto <dest-list>
    (random-put-on (value-list address dest-list)
      (when (> address 0)
        (let ((length1 (length value-list)))
          (loop with next-empty-slot = 0
            for n from 0 to (1- length1)
            for random-address = (random length1)
            do
              (cond ((zerop (pref (the self-adr-pvar
                                   (aref dest-list random-address))
                                    address))
                    (setf (pref (the self-adr-pvar
                                (aref dest-list random-address))
                                address)
                          (nth - . s-list,))
                          (t
                           (loop for m from next-empty-slot to (1- length1)
                             when (zerop (pref (the self-adr-pvar
                                                 (aref dest-list m))
                                              address))
                             do
                               (setf (pref (the self-adr-pvar
                                             (aref dest-list m))
                                             address)
                                      (nth n value-list))
                               (setq next-empty-slot (+ m))
                               (return nil))))))))))
  ; set the near-field of the root to be itself.
  (setf (pref (the self-adr-pvar (aref near-field-list 0)) 0) 0)
  (loop for i from i to last-node-add
do
    (let*
(let* ((parent-near-field (aref near-field-list j))
       (parent-processor-address))
  (d-x0 (- (pref pt-x p-key) processor-address)
        (pref pt-x p-key) parent-near-field)))
  (d-y0 (- (pref pt-y p-key) processor-address)
        (pref pt-y p-key) parent-near-field)))
  (d-z0 (- (pref pt-z p-key) processor-address)
        (pref pt-z p-key) parent-near-field))))
(if (> (vector-norm pt d-x0 d-y0 d-z0)) dist0)
  (setq interaction-list1 (cons parent-near-field interaction-list1))
(loop for k from 1 to 8
  do
    (let* ((son-add (+ (* 8 parent-near-field) k))
           (d-x (- (pref pt-x p-key) processor-address)
                 (pref pt-x p-key) son-add)))
      (d-y (- (pref pt-y p-key) processor-address)
            (pref pt-y p-key) son-add)))
    (d-z (- (pref pt-z p-key) processor-address)
          (pref pt-z p-key) son-add)))
    (if (> (vector-norm pt d-x d-y d-z)) dist)
      (setq interaction-list1 (cons son-add interaction-list1))
      (setq near-field-list1 (cons son-add near-field-list1)))))))
  (loop with near-field-list1
        with interaction-list1
        for j from 0 to (if (= parent-processor-address 0) 0 80)
        while (if (= parent-processor-address 0)
                 t
                 (not (zerop (pref (the self-adr-pvar
                                    (aref near-field-list j))
                                    parent-processor-address))))
        do
          (let* ((parent-near-field
                   (aref near-field-list j))
                  (parent-processor-address))
            (d-x0 (- (pref pt-x p-key) processor-address)
                  (pref pt-x p-key) parent-near-field)))
            (d-y0 (- (pref pt-y p-key) processor-address)
                  (pref pt-y p-key) parent-near-field)))
            (d-z0 (- (pref pt-z p-key) processor-address)
                  (pref pt-z p-key) parent-near-field))))
          (if (> (vector-norm pt d-x0 d-y0 d-z0)) dist0)
            (setq interaction-list1 (cons parent-near-field interaction-list1))
            (loop for k from 1 to 8
                  do
                    (let* ((son-add (+ (* 8 parent-near-field) k))
                           (d-x (- (pref pt-x p-key) processor-address)
                                 (pref pt-x p-key) son-add)))
                       (d-y (- (pref pt-y p-key) processor-address)
                             (pref pt-y p-key) son-add)))
                       (d-z (- (pref pt-z p-key) processor-address)
                             (pref pt-z p-key) son-add)))
                       (if (> (vector-norm pt d-x d-y d-z)) dist)
                         (setq interaction-list1 (cons son-add interaction-list1))
                         (setq near-field-list1 (cons son-add near-field-list1)))))))
  finally
(progn
  (cond
    ((> (length near-field-list1) 81)
      (format t
        ""%ERROR in find-near-field-list. LENGTH "D > 81"
        (length near-field-list1)) (zl:dbg))
    ((> (length interaction-list1) 140)
      (format t
        ""%ERROR in find-interaction-list. LENGTH "D > 140"
        (length interaction-list1))
        (zl:dbg))
      (t
        (random-put-on
          near-field-list1 processor-address
          near-field-list)
        (random-put-on
          interaction-list1 processor-address
          interaction-list)))
  ))))))
(*defun upward-path ()
  (leaf-expansion!!)
  (loop for i from height-of-tree downto 1
    do (*all (*when (=!! level (the (signed-pvar 4) (!! i)))
              (shiftingi!!)))))

(*defun downward-path ()
  (loop for i from 1 to height-of-tree
    do (*all (*when (=!! level (the (signed-pvar 4) (!! i)))
              ; since shifting3!! clears psi before storing values,
              ; it goes first.
              (shifting3!!)
              (shifting2!!))))

(final-evaluation!!))
(defun error-hack ()
(*all
 (*when (not!! (zerop!! item-list)) ;;(not!! (zerop!! item-list))
 (*let (true-potential diff)
 (declare (type float-pvar true-potential diff))
 (loop for i from (i+ last-pro-add) to last-leaf-add
   do
 (*when (not!! (=!! (self-address!!))
 (the (field-pvar CM:*CUBE-ADDRESS-LENGTH*)
 (!! i)))
 (*let*
 ((address
   ((the (field-pvar CM:*CUBE-ADDRESS-LENGTH*) (!! i)))
 (d-x (-!! (pt-x p-key)
   (pref!! (pt-x p-key) address :many-collisions)))
 (d-y (-!! (pt-y p-key)
   (pref!! (pt-y p-key) address :many-collisions)))
 (d-z (-!! (pt-z p-key)
   (pref!! (pt-z p-key) address :many-collisions)))
 (d-norm (vector-norm!! (pt d-x d-y d-z))))
 (declare (type (field-pvar CM:*CUBE-ADDRESS-LENGTH*) address))
 (declare (type float-r``x d-x d-y d-z d-norm))
 (*set :add
 (if!! (not!! d-norm) (!! 0) (/!! item-list d-norm))
 true-potential address nil :many-collisions))))
 (*set diff
 (/!! (abs!! (-!! true-potential potential)) true-potential))
 (setq average-error (/ (*sum diff) number-of-particles))
 (setq max-error (*max diff))))))
;;; Initialization

(*defun local-init *)
(*all
 (*set potential (!! 0))
 (loop for i from 0 to p
do
 (loop for j from 0 to (- p i)
do
 (loop for k from 0 to (- p i j)
do
 (*set (the float-pvar (aref phi i j k)) (!! 0))
 (*set (the float-pvar (aref psi i j k)) (!! 0))
 (*set (the float-pvar (aref array-template i j k)) (!! 0))
 (*set (the float-pvar (aref array-templatei i j k)) (!! 0)))))

(defun global-init (precision height-of-tree1)
 (setq height-of-tree height-of-tree1)
 (setq p precision)
 (setq last-node-add (+ 8 (/ (- (expt 0 height-of-tree) 1) 7)))
 (setq last-leaf-add last-node-add last-pro-add last-node-add)
 (setq son-add-list nil)
 (setq p-key (pt (allocate!! nil nil '(type (float-pvar 23 8))))
 (allocate!! nil nil '(type (float-pvar 23 8)))
 (allocate!! nil nil '(type (float-pvar 23 8))))
 (setq near-field-list (make-array 81))
 (setq interaction-list (make-array 140))

(setq phi (make-array (list (1+ p) (1+ p) (1+ p))))
(setq psi (make-array (list (1+ p) (1+ p) (1+ p))))
(setq array-template (make-array (list (1+ p) (1+ p) (1+ p))))
(setq array-templatei (make-array (list (1+ p) (1+ p) (1+ p))))

(loop for i from 0 to 80
do
 (setf (aref near-field-list i)
 (allocate!! (!! 0) nil
 '(%type (field-pvar ,CM:*CUBE-ADDRESS-LENGTH*))))
(loop for i from 0 to 139
do
 (setf (aref interaction-list i)
APPENDIX B. PARALLEL CODE

```
(allocate!! (!! 0) nil
  '(type (field-pvar ,CM:*CUBE-ADDRESS-LENGTH*))))
(loop for i from 0 tc p ;initial allocation of pvar's.
  do
    (loop for j from 0 to (- p i)
      do
        (loop for k from 0 to (- p i, j)
          do
            (setf (aref phi i j k)
                  (allocate!! (!! 0) nil '(type (float-pvar 23 8))))
            (setf (aref psi i j k)
                  (allocate!! (!! 0) nil '(type (float-pvar 23 8))))
            (setf (aref array-template i j k)
                  (allocate!! (!! 0) nil '(type (float-pvar 23 8))))
            (setf (aref array-template1 i j k)
                  (allocate!! (!! 0) nil '(type (float-pvar 23 8))))))

(defun prologue
  (number-of-processors height-of-tree1 precision pt1 pt2)
  (*cold-boot :initial-dimensions number-of-processors)
  (global-init precision height-of-tree1)
  (build-tree pt1 pt2)
  (find-neighbor-and-interaction-list1))

(defun epilogue ()
  (error-hack))
```
APPENDIX B. PARALLEL CODE

;;; Test functions

;;; Run time of body in seconds

(defvar how-long (&body body)
  '(let ((initial-time (zl:time)))
    (progn
     ,@body
     (/ (- (zl:time) initial-time) 60.0))))

;;; To make a <form>'s output go to an editor buffer named <buf>, do a
;;; (with-output-to-buffer <buf> <form>)

(defvar with-output-to-buffer (name &body body)
  '(let ((zl:standard-output
           (zwei:interval-stream (zwei:find-buffer-named ,name t)))
         ,@body))

;;; An example run

(defvar run (number-of-processors pt1 pt2)
  (loop for height-of-tree1 in '(2 3 4)
    ; h=2: 64 leaves, h=3: 512 leaves, and h=4: 4096 leaves
    for r1 = (how-long
               (prolv
                number-of-processors height-of-tree1 3 pt1 pt2))
    do
    (format t "Height-of-tree="D" height-of-tree1)
    (format t "Construct Decomposition Tree takes "D Seconds" r1)
    (setq number-of-particles 0)
    (loop for number-of-particles1 in
           '(64 64 128 256 512 1024 2048 4096 8192)
             ; '(64 128 256 512 1024 2048 4096 8192 16384)
      do
      (insert-points number-of-particles1)
      (when (or (and (= height-of-tree1 2) (< number-of-particles 513))
                 (and (= height-of-tree1 3) (< number-of-particles 4097)
                      (> number-of-particles 256))
                 (and (= height-of-tree1 4) (> number-of-particles 1023)))
          (local-init)
(format t "%-Number-of-particles="D;" number-of-particles)
(cm:time (progn (upward-path) (downward-path)))
;if number-of-particles <= 400, calculate errors.
(when (< number-of-particles 401)
  (epilogue)
  (format t "Max-error="D, Average-error="D"
            max-error average-error)))))

(defun test (name-of-output-buffer)
  (with-output-to-buffer
    name-of-output-buffer
    (run 32768 (pt 1.0 1.0 1.0) (pt -1.0 -1.0 -1.0))))
Bibliography


