Distribution independent parallel algorithms and software for hierarchical methods with applications to computational electromagnetics

by

Bhanu Hariharan

A thesis submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:
Srinivas Aluru, Major Professor
Shanker Balasubramaniam
G.M.Prabhu

Iowa State University
Ames, Iowa
2003

Copyright © Bhanu Hariharan, 2003. All rights reserved.
This is to certify that the master's thesis of
Bhanu Hariharan
has met the thesis requirements of Iowa State University

Signatures have been redacted for privacy
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF TABLES</td>
<td>v</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>vi</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>vii</td>
</tr>
<tr>
<td>CHAPTER 1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Motivation for Research</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Organization of the Thesis</td>
<td>4</td>
</tr>
<tr>
<td>CHAPTER 2. LITERATURE REVIEW</td>
<td>5</td>
</tr>
<tr>
<td>2.1 Hierarchical Data Structures</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Hierarchical Methods</td>
<td>7</td>
</tr>
<tr>
<td>2.3 Parallel Implementations</td>
<td>8</td>
</tr>
<tr>
<td>CHAPTER 3. PARALLEL ALGORITHMS AND SOFTWARE FOR COMPRESSED OCTREES</td>
<td>11</td>
</tr>
<tr>
<td>3.1 Octree Data Structure</td>
<td>11</td>
</tr>
<tr>
<td>3.1.1 Basic Notations and Octree Construction</td>
<td>11</td>
</tr>
<tr>
<td>3.1.2 Octree Vs. Compressed octree</td>
<td>12</td>
</tr>
<tr>
<td>3.2 Compressed Octree Data Structure</td>
<td>13</td>
</tr>
<tr>
<td>3.2.1 Basic Notations</td>
<td>13</td>
</tr>
<tr>
<td>3.2.2 Cell Representation and Operations</td>
<td>13</td>
</tr>
<tr>
<td>3.2.3 Construction</td>
<td>14</td>
</tr>
<tr>
<td>3.3 Tree Accumulations</td>
<td>19</td>
</tr>
<tr>
<td>3.3.1 Upward Accumulation</td>
<td>19</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 3.1</td>
<td>Size of the residual tree for 128,000 and 512,000 points as a function of the number of processors. The number of levels specified is 12.</td>
<td>22</td>
</tr>
<tr>
<td>Table 3.2</td>
<td>Run-times in seconds for the upward accumulation query for 128,000 points. The accumulation function used is addition.</td>
<td>23</td>
</tr>
<tr>
<td>Table 3.3</td>
<td>Run-time in seconds for leaf neighboring cell query for 128,000 points.</td>
<td>25</td>
</tr>
<tr>
<td>Table 4.1</td>
<td>Run-times in seconds for various stages of the algorithm for 20,000 unknowns for uniform distribution.</td>
<td>41</td>
</tr>
<tr>
<td>Table 4.2</td>
<td>Run-times in seconds for various stages of the algorithm for 20,000 unknowns for non-uniform distribution.</td>
<td>41</td>
</tr>
<tr>
<td>Table 4.3</td>
<td>Run-times in seconds for various stages of the algorithm for 1,000,000 unknowns for uniform distribution.</td>
<td>42</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Figure 3.1  A two dimensional domain and its corresponding quadtree and compressed quadtree. The term *quadtree* is used in two-dimensions because each node has at most 4 children. ........................................ 12

Figure 3.2  Canonical EMC problem................................................................. 18

Figure 3.3  Run-times of parallel tree construction as a function of the number of processors for 128,000 points and 512,000 points. ........................................... 19

Figure 3.4  Run-times for the stages of tree construction for 128,000 points. ........ 20

Figure 3.5  Run-times for the stages of tree construction for 512,000 points. ........ 21

Figure 3.6  Run-times of upward and downward accumulation queries for 128,000 points. The accumulation function uses a delay of a quarter of a millisecond. ........................................... 24

Figure 4.1  Illustration of translations. Node u, parent of node v, has node w in its near field. v generates subcells a, b, c and d of node w. Subcell a is translated to v; Subcells b and d are empty; Subcell c has a child node e and rest of the region is empty. Therefore, e should be translated to v. 39

Figure 4.2  Total run-times for solving the problem as a function of the number of processors for 20,000 and 40,000 unknowns. ........................................... 40

Figure 4.3  Run-time as a function of the problem size on 16 processors for (i) one iteration of farfield computation and (ii) total application. ........................... 42

Figure 4.4  Comparison between analytically and numerically obtained RCS data for a sphere of radius 1.0m discretized using 518,310 unknowns. .................. 43
ABSTRACT

Octrees are tree data structures used to represent multidimensional points in space. They are widely used in supporting hierarchical methods for scientific applications such as the N-body problem, molecular dynamics and smoothed particle hydrodynamics. The size of an octree is known to be dependent on the spatial distribution of points in the computational domain and is not just a function of the number of points. For this reason, run-time of an algorithm using octree that depends on the size of the octree, is unknown for arbitrary distributions.

In this thesis, we present the design and implementation of parallel algorithms for construction of compressed octrees and queries that are typically used by hierarchical methods. Our parallel algorithms and implementation strategies perform well irrespective of the spatial distribution of data, are communication efficient, and require no explicit load balancing. We also developed a software library which provides the functionality of parallel tree construction and various queries on compressed octrees. The purpose of the library is to enable rapid development of applications and to allow application developers to use efficient parallel algorithms without the necessity of having detailed knowledge of the algorithms or of implementing them. To demonstrate the performance of our algorithms and to show the effectiveness of the library, we developed a complete end-to-end parallel electromagnetics code for computing the scattered electromagnetic fields from a Perfect Electrically Conducting surface. We used the functions provided by the software library to develop a Fast Multipole Method based solution to this problem. Experimental results show that our algorithms scale well and have bounded communication irrespective of the shape of the scatterer.
CHAPTER 1. INTRODUCTION

Many scientific applications model a physical system using either grid-based simulation or particle-based simulation. In grid-based methods, space is discretized using a finite grid and the quantities of interest are determined at each of the grid points. In particle-based methods, the system is evolved by determining the quantities of interest at the particle positions, as a result of direct particle-particle interactions. The classical computational approach of these methods in a system of $N$ particles requires $O(N^2)$ computations and is prohibitive for a large system, laying the need for faster methods.

Faster methods typically employ a divide and conquer strategy, partitioning the computational domain into a hierarchy of subdivisions and using approximations at various levels of the subdivision to reduce overall computational cost. Hence they are known as hierarchical methods and the applications employing these methods are called hierarchical applications. Well known hierarchical applications are, N-body problem in astrophysics and celestial mechanics, molecular dynamics, fluid dynamics, hierarchical radiosity, volume rendering, and smoothed particle hydrodynamics, to name a few. Examples of hierarchical grid based methods are multigrid and adaptive mesh refinement methods. Particle based hierarchical methods include smoothed particle hydrodynamics (SPH) and the Fast Multipole Method (FMM). FMM [47] has become a standard numerical method with wide applications in astrophysics, computational electromagnetics, molecular dynamics, and computational fluid dynamics. The basic idea of FMM is to compute cluster-cluster interactions instead of particle-particle interactions. All the applications using FMM follow the same methodology but have different equations for computing interactions. A truncated series approximation is used to compute interactions. Some variations in computing interactions have been used. For instance, the Barnes-Hut al-

Data structures used by the hierarchical applications for representing the multidimensional point data are called hierarchical tree data structures or simply, hierarchical data structures. Examples include k-d trees [8], hyperoctrees [22, 51] (quadtrees in two dimensions and octrees in three dimensions), cell trees [16], BSP trees [23], fair split trees [13], BBD trees [5], and k-d tries [44]. All these data structures have been used in many methods and applications, sometimes the same data structure has been used by multiple methods, possibly for the same application (for example, hyperoctrees have been used for FMM [59], SPH [29], and mesh generation [10]) and also different data structures have been used for the same application, possibly for the same method (for example, N-body problem has been solved using octrees [49], hashed octrees [66] and k-d tree [19]). Samet gives a detailed description of hierarchical data structures in [52]. In this thesis, we focus on octrees and applications that use octrees.

1.1 Motivation for Research

One of the primary motivations behind this work is the large number of scientific applications that use octrees and the importance of efficient data structures in the design of algorithms for hierarchical applications. A data structure is said to be distribution dependent if its size is dependent on the spatial distribution of the points in the computational domain and not just on the number of points. An algorithm is said to be distribution dependent if its run-time depends on the spatial distribution. If the algorithm uses a distribution dependent data structure, then it inherits its distribution dependency. However, mere use of a distribution independent data structure does not guarantee a distribution independent algorithm. Experimental results are often presented as the only means to validate performance of a distribution dependent algorithm. Octrees are distribution dependent and the performance of the algorithms using octrees for arbitrary distributions is therefore unbounded.

Significant research efforts have been directed towards parallelizing applications that use octrees. Some parallelizations give good load balance only for uniform distributions and many
require explicit load balancing after the tree construction leading to a lot of communication overhead and additional software complexity. Many algorithms for octrees are application-specific and are optimized for the type of accesses required by the specific application under consideration. The techniques developed do not usually translate to performance gains in other applications and are not useful as efficient, general-purpose parallel spatial tree data structures. As a result, much of the effort involved in parallelizing an application is directed towards the design of parallel algorithms for constructing and querying octrees and their implementation. While the purpose of these applications may be different, they share the same tree data structure and therefore also share the tasks of tree construction, computing aggregate information in bottom-up and top-down traversals, and performing multiple queries in parallel. Therefore it will be extremely helpful to identify and abstract the commonalities present in various hierarchical methods using octrees, design efficient parallel algorithms for them, and encapsulate them in a software library.

In this thesis, we addressed the above mentioned issues. Firstly, we designed radically different parallel algorithms and implementation strategies to ensure theoretically provable and practically efficient performance independent of spatial distribution. In order to allow the design of provably efficient algorithms, we use compressed octrees instead of regular octrees. We also implemented algorithms for queries typical of many hierarchical applications. The main distinguishing features of our algorithms are provably good run-times, minimal communication overhead and good load balance without the need for any explicit balancing. Secondly, we developed a software library that will enable rapid development of applications, allowing application developers to use efficient parallel algorithms developed for this purpose, without the necessity of having detailed knowledge of the algorithms or of implementing them. For most of the problems considered in this thesis, we have been able to design distribution independent algorithms.

We implemented our software in C in a message passing environment. Our main goal in developing this software library using efficient algorithms was to provide a data structure for use by a wide range of applications and not on focusing on a particular application or memory
architecture. Further, the queries provided support a number of methods. Using the framework provided by the library, we solved the problem of computing scattered electromagnetic fields from a Perfect Electrically Conducting (PEC) surface using the Fast Multipole Method. The electromagnetic scattering analysis problem is important for many technologies like communications and antennas, wireless, ocean modelling, biomedical, geophysics, opto-electronics etc. The validity of our software is demonstrated using problems with analytically computable results and validated sequential software.

1.2 Organization of the Thesis

The remainder of this thesis is organized as follows: In Chapter 2, we present a review of relevant literature on hierarchical methods and data structures. Chapter 3 describes our efficient parallel algorithms and software for compressed octrees. Design of the library and experimental results are also presented. Chapter 4 introduces the computational electromagnetics problem and contains details of how we use our parallel algorithms to compute fields scattered from PEC surfaces along with the experimental results. Conclusions and future work are presented in Chapter 5.
CHAPTER 2. LITERATURE REVIEW

2.1 Hierarchical Data Structures

There is a large body of literature on the design of hierarchical tree data structures for multidimensional data. Typically these are designed for supporting either hierarchical methods or for efficient geometric searches including nearest-neighbor searches, range-queries and various other well-known problems in computational geometry and spatial databases. All of them use a hierarchical subdivision of space into boxes, differing in the shape of the box and in the way the boxes are split. Designing efficient data structures is significantly important as size of the data structure and the algorithms for data access dictate the run-time complexity and memory requirement of an application. Samet gives an in-depth description of various spatial data structures along with their applications in a series of two books [52, 53].

A popular data structure for storing multidimensional data used by many hierarchical methods is the hyperoctree, called quadtree in two dimensions and octree in three dimensions [22, 51]). In a hyperoctree, the root node represents all points and a cubic region containing them. The children of a node in the tree represent equi-volume subregions of the region represented by the node and the leaf nodes in the tree represent individual points. Octrees are known to be distribution dependent, that is, the size of the data structure or the number of nodes in the octree is dependent on the spatial distribution of the points. This is because, recursively subdividing a cubic region may result in a long chain of nodes without any branching. To rectify this dependency, Clarkson adopted the concept of compressing such paths and introduced the Compressed hyperoctree in the context of the all nearest neighbors problem [16]. He gave a randomized algorithm to construct a compressed hyperoctree in $O((c^d)N\log N)$ expected time (where $c$ is a constant and $d$ is the number of dimensions). Even though this
alleviated the distribution dependency of octrees, the algorithms for insertions and deletions on a compressed hyperoctree remained inefficient because of its worst case height of $\Omega(N)$. A number of auxiliary data structures such as centroid decomposition trees, dynamic trees and topology trees were proposed to be used with compressed hyperoctrees in order to support faster insertions and deletions. For instance, Bern [9] proposed a $O((cd)^d N \log N)$ time deterministic algorithm for construction of compressed hyperoctrees and used centroid decomposition for $O(d \log N)$ time point searches. But these auxiliary data structures are complicated and hence not widely used.

Multidimensional binary search trees or $k$-$d$ trees, proposed by Bentley [8], have $O(\log N)$ height. These trees generalize octrees by allowing non-cubical regions. Each node in a $k$-$d$ tree represents a rectilinear region and is associated with an axis aligned plane that cuts its region into two subregions. A variety of applications including graph partitioning and database searches use $k$-$d$ trees for efficient data access. They have also been used for hierarchical applications like molecular dynamics and the N-body problem [4]. But $k$-$d$ trees have poor run-times for queries like the nearest neighbor and spherical region. This is because they have arbitrarily large aspect ratios, defined as the ratio of the maximum and the minimum sidelengths along the coordinate axis, leading to $\Omega(N)$ search times in the worst case, though the run-times in practice are known to be quite good.

Another distribution independent data structure, called the fair-split tree, was introduced by Callahan and Kosaraju [13]. Like octrees, though they have a good aspect ratio, their worst case height is $\Omega(N)$. So they introduced the concept of well separated pair decomposition, built in $O(N \log N)$ sequential time using the fair-split tree, and use that to solve the $k$-nearest neighbor problem in $O(k N)$ sequential time and FMM based N-body potential fields problem in $O(N)$ sequential time [13, 34]. However, the practical performance of the data structure and its applicability to other hierarchical methods is not known.

Aluru et al. [1] introduced Distribution-Independent Adaptive Tree or DIAT, a compressed hyperoctree that uses a simple balanced binary search tree to enable efficient searches, point insertions, and point deletions. They present three $O(N \log N)$ time deterministic algorithms
for the construction of compressed octrees with applications to N-Body problem and \( O(d\log N) \) time algorithms for searches, insertions, and deletions. The algorithms proposed by Aluru et al. [1] are the fastest known for the construction of compressed hyperoctrees and any algorithm that uses hyperoctrees can be potentially implemented using compressed octrees. We therefore make use of their sequential algorithm in our implementation.

### 2.2 Hierarchical Methods

The most popular method used by hierarchical applications is the FMM [47]. FMM was developed by Rokhlin [47] in the context of solving Poisson’s equation and has been applied to the N-Body problem by Greengard [27]. FMM computes interactions between distant clusters of particles instead of computing particle-particle interactions. It uses multiple term approximation for computing interactions allowing for rigorous error bounds. It has thus become a standard numerical method. Most FMM implementations use the distribution dependent octree and are therefore distribution dependent. The run-time of their algorithms is \( O(N) \) only for uniform distributions and their performance for arbitrary distributions is not known.

The concept of using clustering in particle simulations to reduce computational complexity was also used by A. Appel [4] for applications in astrophysics. Appel introduced the use of binary tree data structure to cluster particles where each cluster was considered as a particle at the center of mass in a cluster-cluster interaction.

Barnes-Hut algorithm [7] is based on the algorithm of Appel. It uses the octree instead of binary trees and for this reason, octrees are called BH-trees in Barnes-Hut implementations. Barnes-Hut algorithm first constructs the octree and computes the center of mass and total mass for each node in the tree. Then for each particle, it traverses the tree to compute the force on it. The computational complexity of Barnes-Hut algorithm is claimed to be \( O(N\log N) \), but it is true only for uniform distributions. It uses first order approximation to compute interaction between a particle and a cluster and so it is not as accurate as FMM. However, it is easier to implement than FMM on parallel computers [60] and a number of parallel implementations of the Barnes-Hut algorithm are known. Salmon and Warren have introduced a hybrid of
Barnes-Hut and FMM in [49]. Zhao [68] and Anderson [2] have proposed other algorithms but these are variants of FMM.

2.3 Parallel Implementations

Significant research efforts have been directed at parallelizing algorithms and implementations for hierarchical data structures for the tree based methods. Advances in parallel implementations for Barnes-Hut developed especially in the context of the N-body problem can be found in [36, 49, 62, 65, 66] and parallel FMM algorithms and implementations are described in [35, 55, 60, 63]. Parallel implementation of these methods has been applied in areas such as molecular dynamics [12, 54], capacitance extraction [67] and electromagnetic scattering [20]. Irrespective of the application area, a major challenge in parallel hierarchical methods is to achieve load balance and communication efficiency independent of the spatial distribution of the points in the computational domain. Most of the known parallel hierarchical methods use the distribution dependent octree data structure and are thus distribution dependent in nature. Though some significant advances are made in developing distribution-independent algorithms, their promise has remained largely theoretical so far [13, 55]. Sevilgen et al. discuss parallel algorithms for compressed octree construction and propose parallel algorithms for the tree-based methods on distributed memory machines [55]. Their parallel compressed octree construction algorithm however uses three expensive parallel sorts leading to a lot of communication. Though they present rigorous run-time analysis of their algorithms, the practical performance of their strategy is not known. The application of fair-split trees to hierarchical tree based methods like FMM, has been reported to run in $O(\log N)$ time using $O(N/\log N)$ processors on an EREW PRAM and does not translate to an efficient algorithm on parallel computers [13].

The parallel implementations by Leathrum [35] and Lu et al. [39] for FMM in three dimensions using octrees provide provably good load balance only for uniform distribution of points. Salmon [50] and Singh [60] introduced new data partitioning heuristics to achieve good load balance for non-uniform distributions. Salmon [49, 50] parallelizes Barnes-Hut in a message
passing environment and uses Orthogonal Recursive Bisection or ORB for domain decomposition. In ORB decomposition, space is recursively divided into two subspaces such that each subspace has equal number of points. The regions are distributed to the processors and each processor constructs its local octree. Before the force computation phase, each processor collects all the data that it may need for its local particles using a single communication. Each processor determines the destination processors to which its local nodes may be sent using a simple distance criteria. If the distance between the node and the processor boundary is larger than a threshold, the node is sent to the processor. Since ORB partitioning allocates rectangular regions to processors, a processor can easily locate the destination processors. But when the criteria is not as simple and needs a knowledge of the distribution of the particles in the rectangular region, such a technique of collecting data a priori is not possible. Liu et al. [36] also use a similar scheme in their implementation of Barnes Hut method. Unlike [49], they incrementally update their octrees for the reason that it is expensive to implement ORB data partitioning scheme from scratch during every time step of the N-body simulation.

However, ORB decomposition has been shown to be inefficient for FMM by Singh et al. in [60] as it leads to significant load imbalances for certain distributions. They propose another partitioning scheme known as the costzones and have used it to parallelize FMM on shared address architecture [59, 60]. In this approach, the cost of every node as computed in the previous time step is stored with the node. The total cost of the domain is equally divided among processors. Every processor determines its local cells in an inorder traversal of the tree. In a shared memory model, since every processor has access to every memory location, such a repartitioning is possible for load balancing, but in a distributed memory model, it would lead to a lot of communication overhead. Also such an irregular partitioning leads to a lot of communication in the computation stages of FMM. Lu et al. [40] propose a partitioning scheme called weighted subtrees for the distributed memory model. After the tree construction phase, the FMM tree is partitioned into subtrees which are then assigned to processors one after the other until the number of particles on each processor is close to a known quantity. The subtrees are rooted at the level at which the number of nodes is greater than the number
of processors so that each processor is assigned at least one subtree. This algorithm requires that every processor has knowledge of the global tree and needs explicit load balancing after the tree construction phase.

In another implementation of Barnes-Hut method using octrees, Salmon and Warren introduced another kind of data partitioning that uses space filling curves for domain decomposition [66]. Each cell is assigned a unique integer key and the ordering of the cells according to their keys corresponds to Morton ordering. A hash table is used to map the key to the memory location where the cell data is stored. This kind of mapping helps in retrieving non-local data needed during tree traversals when the data required cannot be determined a priori. They use multiple threads in the force computation stage of Barnes-Hut method so that the communication latency can be hidden by switching controls between the threads. However the program structure gets overly complicated with the use of threads.

In the light of the knowledge of existing work on parallel hierarchical methods using octrees, the focus of our work has been to develop efficient parallel algorithms and software on a unifying data structure to support a wide range of applications. We use compressed octrees to allow the design of efficient algorithms. Unlike previous work, our parallel algorithms provides all of the following features: distribution independent behavior, good load balance, bounded communication, no explicit load balancing, good speed-up and simple program structure.
CHAPTER 3. PARALLEL ALGORITHMS AND SOFTWARE FOR COMPRESSED OCTREES

In this chapter, we describe our algorithms for constructing and querying compressed octrees and also present implementation strategies used in developing the library functions.

3.1 Octree Data Structure

Octrees have been used in a number of scientific applications [35, 47, 45, 66, 69] to represent point data. In an octree, the root node represents all points and a cubic region containing them. The children of a node in the tree represent subregions of the region represented by the node and the leaf nodes in the tree represent individual points.

3.1.1 Basic Notations and Octree Construction

We use the term cell to denote a cubic region, the term subcell to denote a cell that is contained in another and the term immediate subcell to denote a cell obtained as a result of immediate subdivision of another cell into 8 cells having half the side length of the original cell. The length of a cell is the span of the cell along any dimension. Let $N$ denote the number of points.

The computational domain is a cell large enough to contain all points. Recursive subdivision of this cell is naturally represented by a tree, called octree. The subdivision process is stopped on cells having exactly one point or when the smallest allowable size for a cell (fixed based on machine precision or other considerations) is reached. A cell that is not further subdivided is a leaf cell.
3.1.2 Octree Vs. Compressed octree

In an octree, when the points that lie within a cell also lie within one of its immediate subcells, chains are formed in the tree. Each node on a chain represents the same set of points and hence contains the same information duplicated, perhaps in a different form. In principle, a chain can be arbitrarily large irrespective of the total number of points, making the size of the octree dependent upon the distribution of the points. The run-time of the applications using octree data structure depend on the size of the octree which is unbounded for arbitrary distributions. Thus, algorithms using octrees are distribution dependent.

In a compressed octree, each such chain is replaced by a single node (see Figure 3.1). This not only avoids information duplication but also makes the size distribution independent. The size of a compressed octree is $O(N)$. It is not a height-balanced tree and its height could be as large as $\Omega(N)$. A compressed octree for $N$ points can be constructed sequentially in $O(N \log N)$ time, which is optimal [1]. To achieve a provably good run-time independent of the distribution, we use the compressed octree data structure. It can be directly used by applications coded to use octrees.
3.2 Compressed Octree Data Structure

3.2.1 Basic Notations

For each node in a compressed octree, we define two cells: The small cell at a node \( v \), denoted by \( S(v) \), is the smallest cell that contains all the points in its subtree. The large cell of the node, denoted by \( L(v) \), is the largest cell that contains only the points in its subtree. It can be obtained by taking the appropriate immediate subcell of the small cell at its parent node. The following subsection describes how we represent these cells in the computational domain and the various operations we use on them. A compressed octree can be directly used by applications coded to use octrees. If necessary, octrees can be constructed from compressed octrees. To do this, a chain can be created in place of a node whose large cell and small cell are not identical, such that each node in the chain represents a cell half the length of its parent cell.

3.2.2 Cell Representation and Operations

We need a unique representation for the cells that allows for an ordering of the cells such that locality is preserved, an important criteria for data distribution. Therefore, we follow [66] and represent cells using integer keys. A cell space is defined as the computational domain divided recursively into cells of the same size. Each cell in the cell space represents a cubic region that could be associated with an integer triple \((x, y, z)\) where \(x\), \(y\) and \(z\) are the coordinates of the cell in the cell space. The integer key for the cell is formed by interleaving the bits of \(x\), \(y\) and \(z\) in that order, and prepending it with a 1 bit. This unique and unambiguous representation of all the cells in the tree corresponds to ordering them according to Morton ordering, also known as Z-space filling curve ordering [42]. The following operations are performed on cells and are treated as constant time operations in the algorithms in the subsequent sections.

- Check if a cell \( C_1 \) is contained in another cell \( C_2 \). If \( C_2 \) is a prefix of \( C_1 \), then \( C_1 \) is contained in \( C_2 \), otherwise not.

- Find the smallest cell containing two cells \( C_1 \) and \( C_2 \). This is obtained by finding the
longest common prefix of $C_1$ and $C_2$ that is a multiple of 3 bits. If node $w$ is the lowest common ancestor of nodes $u$ and $v$ in a compressed octree, then $S(w)$ is the smallest cell containing $S(u)$ and $S(v)$. Thus, lowest common ancestors in compressed octrees can be computed in constant time.

- Find the immediate subcell of cell $C_1$ that contains a given cell $C_2$. If $3k + 1$ is the number of bits representing the cell $C_1$, the required immediate subcell is given by the first $3k + 4$ bits of $C_2$. This operation is useful in computing the large cell $L(v)$ of a node $v$ in the compressed octree. If $u$ is the parent of $v$, then $L(v)$ is the immediate subcell of $S(u)$ that contains $S(v)$.

**Definition 1** Let $D$ be the length of the domain and $l$ be the length of a cell. The level of the cell in the tree is defined to be $\log_2 \frac{D}{l}$.

**Definition 2** The level of a node in the compressed octree is defined to be the level of its small cell.

### 3.2.3 Construction

In this section, we present our parallel algorithm for constructing compressed octrees. Although tree construction requires an insignificant amount of the total run-time of most applications, this stage is very important because tree partitioning is the key determinant of the load balancing and communication efficiency of subsequent stages. We store the tree in postorder traversal and this helps in designing efficient algorithms for queries. No explicit load balancing is required in our implementation as will become evident later.

The algorithm takes as input the number of allowable levels ($L$) in the tree to determine the size of the smallest cell. In practice, $L$ is quite small. For instance, 20 levels allows us to have a tree with potentially $8^{20} = 2^{60}$ leaf nodes, enough to capture a wide variety of distributions for even the largest simulations. For any pair of cells, either the two cells are disjoint or one of them is completely contained in the other cell. We consider two cells to be disjoint if they merely touch at the boundaries. Define an ordering of a pair of cells as follows: If a cell is
completely contained in another, then the cell with smaller size appears first in the ordering. If they are disjoint, find the smallest cell that contains both these cells. Each of the two cells will be contained in a distinct immediate subcell of this smallest cell. Order the two cells according to the value of the keys of the corresponding immediate subcells. These operations can be done using bit arithmetic, and are taken as $O(1)$ time operations.

The compressed octree is constructed as follows: Each processor is initially given $\frac{N}{P}$ points where $P$ denotes the number of processors. For each point, the corresponding leaf cell is generated using the bit-interleaving scheme described in the previous subsection. These are sorted in parallel, eliminating duplicates. Each processor borrows the first leaf cell from the next processor and runs a sequential algorithm to construct the compressed octree for its leaf cells using the borrowed cell in $O\left(\frac{N}{P} \log \frac{N}{P}\right)$ run-time [1]. This local tree is stored in postorder traversal order in an array. Each node stores the indices of its parent and children in the array. To obtain the local tree in postorder traversal order, we order the nodes according to their small cells. Such an ordering of the leaf nodes of the compressed octree results in the left to right ordering of the leaves and ordering of all the nodes in the compressed octree corresponds to the postorder traversal of the nodes in the tree. Furthermore, given two nodes $v_1$ and $v_2$ in the compressed octree, the smallest cell containing $S(v_1)$ and $S(v_2)$ is the same as $S(u)$, where $u$ is the lowest common ancestor of $v_1$ and $v_2$, and appears after $v_1$ and $v_2$ in the postorder array.

Consider the single compressed octree for the $n$ points, which we term the global compressed octree. The following lemma establishes that every node in this octree is generated in at least one of the processors.

**Lemma 1** Each node in the global compressed octree can be found in the local compressed octree of at least one processor.

**Proof:** Every internal node is the lowest common ancestor of at least one consecutive pair of leaves. More specifically, let $u$ be a node in the global compressed octree and let $v_1, v_2, \ldots, v_k$ be its children ($2 \leq k \leq 8$). Consider any two consecutive children $v_i$ and $v_{i+1}$ ($1 \leq i \leq k - 1$). Then, $u$ is the lowest common ancestor of the rightmost leaf in $v_i$'s subtree and the leftmost
leaf in \( v_{i+1} \)'s subtree. If we generate the lowest common ancestors of every consecutive pair of leaf nodes, we are guaranteed to generate every internal node in the compressed octree. However, there may be duplicates — each internal node is generated at least once but at most seven times \((2^d - 1 \text{ times in } d \text{ dimensions})\).

Each node in a local compressed octree is also a node in the global compressed octree for the whole set of points. In order to generate the postorder traversal array of the global compressed octree, some nodes in the local tree that should actually appear later in the postorder traversal of the global tree, should be sent to the appropriate processors. This is because the rightmost leaf in their subtree may reside in some other processor. Such nodes, termed out of order nodes, can be easily identified as they appear consecutively after the borrowed leaf in the postorder traversal of the local tree. Also, the borrowed leaf is used to make sure that the lowest common ancestor of every consecutive pair of leaves is generated; it is not considered a part of the local tree. An Allgather operation is used to collect the first leaf cell in each processor into an array of size \( P \). Using a binary search in this array, the destination processor for each out of order node can be found. Nodes that should be routed to the same processor are collected together and sent in a single message. This uses Many-to-Many communication.

**Lemma 2** The number of out of order nodes in a processor is at most \( L \).

**Proof:** We show that there can be at most one node per level in the local tree that is out of order. Suppose this is not true. Consider any two out of order nodes in a level, say \( v_1 \) and \( v_2 \). Suppose \( v_2 \) occurs to the right of \( v_1 \) (\( v_2 \) comes after \( v_1 \) in cell order). Since \( v_1 \) and \( v_2 \) are generated, some leaves in their subtrees belong to the processor. Because the leaves allocated to a processor are consecutive, the rightmost leaf in the subtree of \( v_1 \) belongs to the processor. This means \( v_1 \) is not an out of order node, a contradiction.

**Lemma 3** The total number of nodes received by a processor is at most \( 7L \).

**Proof:** We first show that a processor cannot receive more than one distinct node per level. Suppose this is not true. Let \( v_1 \) and \( v_2 \) be two distinct nodes received at the same level.
Without loss of generality, let $v_2$ be to the right of $v_1$. A node is received by a processor only if it contains the rightmost leaf in the subtree of the node. Therefore, all the leaf nodes between the rightmost leaf in the subtree of $v_1$ and the rightmost leaf in the subtree of $v_2$ must be contained in the same processor. In that case, the entire subtree under $v_2$ is generated on the processor itself and the processor could not have received $v_2$ from another processor, a contradiction. Hence, a processor can receive at most $L$ distinct nodes from other processors. As there can be at most $7 \ (2^d - 1 \text{ in } d \text{ dimensions})$ duplicate copies of each node, a processor may receive no more than $7L \ ((2^d - 1)L \text{ in } d \text{ dimensions})$ nodes.

The received nodes are merged with the local postorder traversal array and their positions are communicated back to the sending processors. The net result is the postorder traversal of the global octree distributed across processors. Each node contains the position of its parent and each of its children in this array. From here onwards, we use the term local tree or local array to refer to the local portion of the global compressed octree stored in postorder traversal order. The size of the local tree is $O \left( \frac{N}{P} + L \right)$. We use the terms node (in the tree) and the cell represented by the node interchangeably, provided such usage does not result in confusion.

**Experimental Results**

We experimentally tested our tree construction algorithm using an IBM xSeries cluster. The cluster consists of 32 dual processor nodes, each running at 1.26 GHZ and containing 1GB RAM per node. The nodes are connected using Myrinet to support parallel communication at a peak rate of 2Gbits/sec, full duplex.

To verify the distribution-independent nature of the algorithm and to show that the algorithm performs well in practice, we tested our algorithm with both uniformly and non-uniformly distributed points. We used multiplicative congruential random number generator for uniform distribution and generated points that are uniformly distributed in all three dimensions in a unit cube. One of the non-uniform distributions is generated using three normal distributions with different means (0.4, 0.5, 0.7 for $x$, $y$, and $z$ respectively) and standard deviations (0.2, 0.2, 0.1 for $x$, $y$, and $z$ respectively), to generate the $x$, $y$, and $z$ coordinates of the points. For
the other non-uniform distribution, we chose a canonical EMC problem (see Figure 3.2) and generated equi-spaced points in all three dimensions. Though the distribution of the unknowns is uniform on the surface of the plate and its mirror image, and along the cable and its mirror image, the distribution within the entire computational domain (an enclosing cube) becomes non-uniform. To distinguish between the two non-uniform distributions, the latter is referred to as the EMC problem.

The run-times of parallel tree construction for 128,000 and 512,000 points for the uniform and the two non-uniform distributions as a function of the number of processors are shown in Figure 3.3. From the graphs, it is easy to see that the algorithm scales well even for non-uniform distributions, as predicted by the theoretical analysis. The breakdown of the run-times into times spent on various stages in the algorithm for 128,000 points and 512,000 points is shown in Figure 3.4 and Figure 3.5. Parallel sorting and local tree construction dominate the run-times. A negligible percentage of time is spent in communicating out of order nodes and incorporating them into the local tree. This is a direct consequence of the result that the number of out of order nodes is a function of the height of the tree, which tends to be small. It also explains why this time is almost the same for the two data sizes. Hence, the algorithm has very little overhead due to parallelization.
3.3 Tree Accumulations

The upward and downward tree accumulations described in the following subsections are typical of many hierarchical applications. In an upward accumulation query, the objective is to aggregate the information at all the descendants of each node in the tree. Typically, a function is supplied to aggregate the information at a node to its parent. To compute the upward accumulation at a node, this function is applied at each node in the tree for each of its children. In a downward accumulation query, the objective is to aggregate the information at all the ancestors of each node in the tree.

3.3.1 Upward Accumulation

Our algorithm for parallel upward accumulation is as follows: First, each processor scans its local array from left to right. During the scan, the information at each node is aggregated to its parent, provided the parent is local to the processor. As the tree is stored in postorder traversal order, if all the children of a node are present in the same processor, it is encountered only after all its children are. This ensures that such a node has received its upward accumulation when the scan reaches it. This computation takes $O \left( \frac{N}{P} + L \right)$ time.

During the scan, some nodes are labeled residual nodes based on the following rules: 1) If a node receives its upward accumulation but its parent lies in a different processor, it is labeled
a residual leaf node. 2) If a node does not have its upward accumulation when it is visited, it is labeled a residual internal node. Each processor copies its residual nodes into an array. The residual nodes form a tree (termed the residual tree) and the tree is present in its postorder traversal order, distributed across processors.

**Lemma 4** The number of residual internal nodes in a processor is at most \( L - 1 \).

**Proof:** We show that there can be at most one residual internal node per level in a processor. Suppose this is not true. Consider any two residual internal nodes in a level, say \( v_1 \) and \( v_2 \). Suppose \( v_2 \) occurs to the right of \( v_1 \) (\( v_2 \) comes after \( v_1 \) in cell order). Because of postorder property, the right most leaf in \( v_1 \)'s subtree and the rightmost leaf in \( v_2 \)'s subtree should belong to this processor. Because the leaves allocated to a processor are consecutive, all the leaves in the subtree of \( v_2 \) belong to the processor. This means \( v_2 \) is not a residual internal node, a contradiction. Therefore, there can be only one residual internal node at each level in a processor. Furthermore, no leaf can be a residual internal node. Hence, the claim.

**Corollary 5** The size of the residual tree is at most \( 8PL \).

**Proof:** From the previous lemma, the total number of residual internal nodes is at most \( P(L-1) \). It follows that the maximum number of residual leaf nodes can at most be \( 7P(L-1)+1 \).
(or \(2^d - 1)P(L - 1) + 1\) in \(d\) dimensions), for a total tree size of at most \(8\(P(L - 1) + 1 < 8PL\)
(or \(2^dP(L - 1) + 1 < 2^dPL\) in \(d\) dimensions).

The residual tree is accumulated using a parallel upward tree accumulation algorithm [55].

The number of iterations of the algorithm is the logarithm of the size of the tree. During each iteration, the algorithm uses a parallel prefix and a Many-to-Many communication. The algorithm could have been directly applied to the global compressed octree but this would require \(O(\log N)\) iterations. The residual tree can be accumulated in \(O(\log P + \log L)\) iterations. Thus, we reduce the iterations from the logarithm of the size of the tree to the logarithm of the height of the tree, which is much smaller!

### 3.3.2 Downward Accumulation

The algorithm for this query is the reverse of the algorithm employed for upward accumulation. First, the downward accumulations for the residual tree are calculated. This requires \(O(\log P + \log L)\) iterations. Then, the downward accumulations are computed for the local tree using a right-to-left scan of the postorder traversal of the local tree.
Table 3.1  Size of the residual tree for 128,000 and 512,000 points as a func-
tion of the number of processors. The number of levels specified
is 12.

<table>
<thead>
<tr>
<th>No. of procs</th>
<th>Uniform distribution</th>
<th>Non-uniform distribution</th>
<th>EMC problem</th>
<th>Uniform distribution</th>
<th>Non-uniform distribution</th>
<th>EMC problem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>128,000</td>
<td></td>
<td></td>
<td>512,000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>36</td>
<td>38</td>
<td>18</td>
<td>32</td>
<td>28</td>
<td>19</td>
</tr>
<tr>
<td>4</td>
<td>69</td>
<td>92</td>
<td>75</td>
<td>99</td>
<td>98</td>
<td>74</td>
</tr>
<tr>
<td>8</td>
<td>147</td>
<td>166</td>
<td>141</td>
<td>151</td>
<td>222</td>
<td>139</td>
</tr>
<tr>
<td>16</td>
<td>286</td>
<td>305</td>
<td>256</td>
<td>344</td>
<td>387</td>
<td>274</td>
</tr>
<tr>
<td>32</td>
<td>536</td>
<td>620</td>
<td>440</td>
<td>640</td>
<td>644</td>
<td>499</td>
</tr>
<tr>
<td>64</td>
<td>957</td>
<td>1115</td>
<td>907</td>
<td>1159</td>
<td>1177</td>
<td>948</td>
</tr>
</tbody>
</table>

**Experimental Results**

The primary advantage of using the parallel accumulation algorithms on the residual tree instead of the global compressed octree is that the size of the residual tree is significantly smaller. The residual tree sizes as a function of the number of processors for two different data sizes (128,000 and 512,000 points) and different distributions are shown in Table 3.1. The size of the residual tree is much less compared to its upper bound of $8PL$. This is because most of the nodes have their children in the same processor and only nodes at higher levels are more likely to form the residual tree. In interpreting these numbers, it should be kept in mind that the global octree for $N$ points has between $N$ and $2N - 1$ nodes. Hence, the size of the residual tree is smaller by several orders of magnitude.

Both the queries were run for uniform and non-uniform data sets using addition as the accumulation function. The results for the upward accumulation query are tabulated in Table 3.2. The results of the downward accumulation query are very similar. The total run-times are significantly smaller when compared to tree construction run-times and do not show much improvement as the number of processors is increased. This is because: 1) the local run-times are small due to the simple accumulation function chosen, and 2) the parallel part, though has only few iterations, uses the time consuming Many-to-Many communication in each iteration. In any application, tree construction will be used prior to running any of these queries. Given that the query times for accumulations are smaller by two orders of magnitude or more, the
Table 3.2  Run-times in seconds for the upward accumulation query for 128,000 points. The accumulation function used is addition.

<table>
<thead>
<tr>
<th>No. of procs</th>
<th>Uniform distribution</th>
<th>Non-uniform distribution</th>
<th>EMC problem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Local</td>
<td>Parallel</td>
<td>Total</td>
</tr>
<tr>
<td>2</td>
<td>0.08</td>
<td>0.01</td>
<td>0.09</td>
</tr>
<tr>
<td>4</td>
<td>0.04</td>
<td>0.01</td>
<td>0.05</td>
</tr>
<tr>
<td>8</td>
<td>0.02</td>
<td>0.01</td>
<td>0.03</td>
</tr>
<tr>
<td>16</td>
<td>0.01</td>
<td>0.01</td>
<td>0.02</td>
</tr>
<tr>
<td>32</td>
<td>0.01</td>
<td>0.03</td>
<td>0.04</td>
</tr>
<tr>
<td>64</td>
<td>0.01</td>
<td>0.05</td>
<td>0.06</td>
</tr>
</tbody>
</table>

The overall application run-time will still show near linear speedup.

Hierarchical applications typically use complex accumulation functions. For instance, in the Fast Multipole Method to solve the N-body problem, the upward accumulation function transfers a truncated Taylor series expansion from a child to its parent in time proportional to the fourth power of number of terms used. Studies show that the run-time of the applications is dominated by this phase (for example, see [66]). To reflect real application behavior, we used an accumulation function consisting of a small delay of 0.25ms. This increased the overall query time but the time spent in communication remained almost the same. The run-times for upward and downward accumulation as a function of the number of processors are shown in Figure 3.6. The figure confirms that 1) the algorithms scale well for complex accumulation functions and 2) the run-times are nearly identical for the upward and downward accumulation queries, as expected.

3.4 Cell Queries

In a cell query, each processor generates an array of cells whose information should be retrieved. The query cells are directed to the processors owning these cells and the cell information is communicated back to the originating processors.

Many queries used in hierarchical applications can be classified under the category of cell queries. Consider two examples of such queries – In the Fast Multipole Method, for each cell, the information within cells that are in a “doughnut region” around the query cell is
sought. In molecular dynamics, the information from points that are within a cut-off radius from each point is sought. If the smallest cell size is chosen such that its length is equal to the cut-off radius, this query translates to finding the neighboring cells of each cell. Specialized algorithms can be designed for each query type. However, they may each require a different tree representation and/or a different data distribution strategy. In order to preserve generality, we use a unified approach. The commonality between most such queries is that spatially local information is sought for each cell. The partitioning of the compressed octree using its postorder traversal order has the effect of partitioning the domain into regions, ordering the regions according to Morton ordering, and allocating them to processors. This ensures that most queries can be answered locally whereas communication is necessary only near the boundaries.

We classify queries into two categories — those that can be answered locally, and those that require remote access. Using an All-to-All communication, the non-local queries are sent to the appropriate processors. Another All-to-All communication is used to receive the answers. Here, communication for receiving remote queries can be overlapped with the computation used for answering local queries to virtually eliminate the parallel overhead.
Table 3.3 Run-time in seconds for leaf neighboring cell query for 128,000 points.

<table>
<thead>
<tr>
<th>No. of proc</th>
<th>Uniform distribution</th>
<th>Non-uniform distribution</th>
<th>EMC problem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Comm</td>
<td>Total</td>
<td>Comm</td>
</tr>
<tr>
<td>2</td>
<td>0.02</td>
<td>30.18</td>
<td>0.09</td>
</tr>
<tr>
<td>4</td>
<td>0.02</td>
<td>16.66</td>
<td>0.12</td>
</tr>
<tr>
<td>8</td>
<td>0.02</td>
<td>9.25</td>
<td>0.09</td>
</tr>
<tr>
<td>16</td>
<td>0.02</td>
<td>4.78</td>
<td>0.07</td>
</tr>
<tr>
<td>32</td>
<td>0.02</td>
<td>3.03</td>
<td>0.06</td>
</tr>
<tr>
<td>64</td>
<td>0.02</td>
<td>2.82</td>
<td>0.05</td>
</tr>
</tbody>
</table>

Experimental Results

To experimentally evaluate the run-time performance of cell queries using our approach, we used the following query: For each leaf cell in the compressed octree, all 26 of its neighboring leaf cells are sought. This query is typical of molecular dynamics applications. The breakdown of the run-times into time spent in answering queries locally and time spent in communication for 128,000 points on the IBM xSeries cluster is shown in Table 3.3. The run-times scale well with the number of processors for both uniform and non-uniform distributions. Because of locality and overlapping of communication with computation, the parallel overhead is negligible. It is important to note that this algorithm gives good load balance provided all processors generate equal number of query cells like in the approach described above. However, it may not be the case for all hierarchical methods because for some cells, the information sought may be more than that for other cells. In that sense, even though the cell query algorithm shows good practical performance, it is not entirely distribution dependent for all approaches.

3.5 Software Design

This section describes the overall design of our software library. The functions provided by the library incorporate the algorithms described in previous sections. We distinguish between two types of information stored at the nodes in a compressed octree — application specific information and the information necessary for tree construction, tree accumulations, and queries.
As an example of application specific information, in the N-body problem, the mass and velocity of the particles in a subtree may be stored at the root node of the subtree. A library intended for many applications should provide flexibility in storing such application specific information. We achieve this by separating it from the tree node information needed for maintaining the tree structure and allowing queries on it. For this, we define the tree node structure $Treenode$ to encapsulate the user defined structure $Node$. $Treenode$ also contains the small cell, large cell, parent pointer, processor containing the parent, and child pointer information for a node. An example user defined $Node$ structure and the $Treenode$ structure are shown below:

```c
typedef struct usernode {
    double mass;
    double velocity;
} Node;

typedef struct tree {
    unsigned long long small_cell;
    unsigned long long large_cell;
    int parent, processor;
    int child[];
    Node node;
} Treenode;
```

Similarly, the structure defining the application specific attributes of a point is kept separate from the information needed for tree library functions. For this, we define the library point structure, denoted by $Library\_Point$, to encapsulate the user defined structure $Point$. An example $Point$ structure is given below and is included in $Library\_Point$ that contains the $x$, $y$, $z$ coordinates and the integer key for the leaf cell it resides in. We do require that the user defined point structure contain the coordinates of the point itself as minimal information.

```c
typedef struct userpoint {
    double a, b, c;
    double mass;
    double velocity;
} Point;

typedef struct libpoint {
    unsigned long long key;
    unsigned x, y, z;
    Point point;
} Library\_Point;
```

Some of the functions provided by the library are described below:
• **Tree construction function:** It is called with the list of point coordinates, number of points, desired number of levels, derived datatype of the user defined node structure and derived datatype of the user defined point. It returns a handle to the tree, which is a structure containing a pointer to the array storing the local tree, number of levels, and number of nodes in the local tree.

```c
void tree_construct_tree(
    Point listofpoints [], /* in */,
    int numberofpoints /* in */,
    int levels /* in */,
    MPI_Datatype Node_datatype /* in */,
    MPI_Datatype Point_datatype /* in */)
```

The handle to the tree obtained from the tree construction function and the derived datatype of the user defined node structure are inputs common to the tree accumulation functions and the neighbor query function.

• **Upward accumulation function:** Apart from the common inputs, it takes as input a pointer to the user defined function which initializes a leaf node, a pointer to the user defined function which initializes an internal node, and a pointer to the user defined aggregation function that translates the information at a child node to its parent node and combines it with the information at the parent node.

```c
void upward_accumulate (
    Tree* tree /* in */,
    MPI_Datatype Node_datatype /* in */,
    void (*Initialize_Leaf)(Node*) /* in */,
    void (*Initialize_Node)(Node*) /* in */,
    void (*child_to_parent)(Node *, Node *)/* in */)
```

• **Downward accumulation function:** Additional inputs to this function include a pointer
to a user defined function which initializes an internal node and a pointer to a user
defined aggregation function that translates information at a node to a child of it and
combines it with the information at the child.

```c
void downward_accumulate(
    Tree* tree, /* in */,
    MPI_Datatype Node_datatype /* in */,
    void (*InitializeNode)(Node*), /* in */
    void (*parent_to_child)(Node*, Node*) /* in */
)
```

- **Neighbor query function:** This function implements the cell query algorithm described
  earlier. It returns the output array and size of the output array.

```c
void neighbor_query(
    Tree* tree, /* in */,
    Node* output[], /* out */
    int* outputsize /* out */,
    MPI_Datatype Node_datatype /* in */
)
```

The details of how the tree is stored, the information at the tree nodes and the algorithms
incorporated by the library functions are hidden from the user. With little knowledge of
writing parallel programs, the user can use the library functions easily. The user is expected to
implement only the initial data partitioning phase and define the application specific structures
and MPI derived datatypes.

The library can also be easily enhanced to accommodate additional queries. For instance,
we implemented the neighbor query function that seeks information from the neighboring 26
cells for each cell in the domain. This can be extended to solving for more queries like the
spherical region query or the nearest neighbor query. The function can be modified to solve
for such queries with little effort or more functions can be added to solve for such queries.
CHAPTER 4. APPLICATION TO ELECTROMAGNETIC SCATTERING

The analysis of electromagnetic scattering by quasi-planar structures is a research topic of intense current interest, as numerous devices fall under this geometric description. Some of the important technologies where computational electromagnetics (CEM) is applicable are: Communications and Antennas, Wireless, Geophysics, Remote Sensing and Ocean Modeling.

CEM solvers can be categorised into those that have been derived using either:

- Differential equations, or
- Integral equations

Integral equation based methods are typically used for solving the electromagnetic scattering problems. In these methods, the problem is first translated to integral equations and then reduced to a system of linear equations using Method of Moments (MoM). The memory required for storing the matrix for the linear system is $O(N^2)$, where $N$ is the number of unknowns that is used to model the scatterer. Direct methods such as Gaussian elimination to solve the system of linear equations require $O(N^3)$ time, making it prohibitively expensive for large problems.

The computational complexity and storage requirements of the iterative methods for the solution to the integral equation techniques require $O(N^2)$ per iteration, which is again not suited for large problems. In 1993, with the development of Fast Multipole Method (FMM) [48] for the solution to the Helmholtz equation, the computational complexity and the memory requirements are reduced to $O(N \log N)$ and $O(N)$ respectively, for uniform distributions. Subsequently, FMM augmented CEM solvers have been used for a variety of applications,
including large scale scattering [17, 18, 37, 38], scattering from quasi-planar objects [32, 33], EMC/EMI analysis [6, 15], and mine detection techniques [24, 25]. A comprehensive survey of the state of the art may be found in [15]. Consequently, classical solvers augmented with the FMM are now becoming the method of choice for electromagnetic analysis. An efficient parallel implementation for CEM greatly enhances the ability to solve larger and more realistic problems in reasonable time. Increasing the electrical size of the object or the modeling of finer features results in larger problem sizes and the analysis goes beyond the capabilities of a serial code.

4.1 Problem formulation

In this section we give the problem formulation and details can be found in [20]. Electromagnetic field with components $E^i(r)$, and $H^i(r)$ when impinged on the surface of a closed perfect electrically conducting body (or PEC), creates a current $J(r)$ on the surface of the PEC. This generates scattered electric and magnetic fields $E^s(r)$ and $H^s(r)$. The PEC or the scatterer resides in free space. The problem of analysis of the electromagnetic scattering by the PEC is translated to solving for the scattered electric fields which is further translated to solving a matrix equation.

The scatterer is enclosed in a fictitious cubic domain. Let $S$ denote the surface of the scatterer and let $\hat{n}$ denote an outward pointing position dependent normal to $S$, and $S_\pm$ denotes hypothetical surfaces that lie $\epsilon_\pm$ on either side of $S$. The electric currents on the surface are represented by a set of basis functions, i.e.,

$$J(r) = \sum_{n=1}^{N} I_n S_n(r)$$  \hspace{1cm} (4.1)

The basis functions chosen in our analysis are the RWG functions that require the surface $S$ to be represented using a set of flat triangular panels and one basis function $S_n(r)$ is associated with each edge joining two triangles:
where $l_n$ is the length of the common edge between the triangles $\Gamma_n^+$ and $\Gamma_n^-$, $A_n^\pm$ is the area of the triangle $\Gamma_n^\pm$, and $\rho_n^\pm(r)$ is the position vector with respect to the free vertex of the corresponding triangle [46].

To represent the scattered fields, we begin with an expression for the magnetic vector potential $A(r)$ that is related to the current as

$$A(r) = \frac{\mu_0}{4\pi} \int_S dr' J(r') e^{-jkR}$$

where $R = |R| = |r - r'|$, and $\mu_0$ and $\varepsilon_0$ are the permeability and permittivity of free space, respectively. The electric and magnetic fields can then be expressed in terms of the magnetic potential as follows

$$E^s(r) = -j\omega \left( I + \frac{\nabla\nabla}{k^2} \right) \cdot A(r) \quad (4.4a)$$

$$H^s(r) = \frac{1}{\mu_0} \nabla \times A(r) \quad (4.4b)$$

The electric field integral equation (EFIE) is constructed by imposing the boundary condition that the total electric field tangential to the surface is zero. In other words, $\mathbf{n} \times \mathbf{E}^s(r) = -\mathbf{n} \times \mathbf{E}^s(\mathbf{r}) \quad \forall \mathbf{r} \in S, S_-, S_+$. This condition together with (4.4a) yields

$$\mathbf{n} \times \mathbf{E}^s(r) = j\omega \mathbf{n} \times \mathbf{n} \times \left( I + \frac{\nabla\nabla}{k^2} \right) \cdot A(r) \quad \forall \mathbf{r} \in S, S_-, S_+$$

$$\L_e \{ J(\mathbf{r}) \}$$

As is well known, this equation holds true for both open and closed structures. Likewise, the magnetic field integral equation (MFIE) can be derived using the condition that the total magnetic field tangential to $S_-$ vanishes. Thus,

$$\mathbf{n} \times \mathbf{H}^t(r) = -\frac{1}{4\pi} \mathbf{n} \times \nabla \times A(r) \quad \forall \mathbf{r} \in S_-$$

$$\L_h \{ J(\mathbf{r}) \}$$
In what follows, we shall use the following notation for convenience: \( V_e \{ \mathbf{E}^i(r), \mathbf{H}^i(r) \} = \hat{n} \times \hat{n} \times \mathbf{E}^i(r) \) and \( V_h \{ \mathbf{E}^i(r), \mathbf{H}^i(r) \} = \hat{n} \times \mathbf{H}^i(r) \). It is well known that the solutions to the EFIE and the MFIE are not unique when analyzing scattering at frequencies that are close to the interior resonance frequencies of the body. This is typically overcome using the combined field integral equation (CFIE), a linear combination of the EFIE and the MFIE. The solution to the latter is unique at resonance frequencies of a closed object. CFIE is prescribed as

\[
V \{ \mathbf{E}^i(r), \mathbf{H}^i(r) \} = \mathcal{L}_c \{ \mathbf{J}(r) \} \quad \forall \ r \in S_-
\]

where \( V \{ \mathbf{E}^i(r), \mathbf{H}^i(r) \} = -\beta/\eta_0 V_e \{ \mathbf{E}^i(r), \mathbf{H}^i(r) \} + V_h \{ \mathbf{E}^i(r), \mathbf{H}^i(r) \} \) and \( \mathcal{L}_c \{ \mathbf{J}(r) \} = -\beta/\eta_0 \mathcal{L}_e \{ \mathbf{J}(r) \} + \mathcal{L}_h \{ \mathbf{J}(r) \} \). In the above equations, \( \eta_0 = \sqrt{\mu_0/\varepsilon_0} \) is the intrinsic impedance of the free space and is introduced for scaling purposes, and \( \beta \) is a positive (real) constant that is greater than zero. Solution to (4.5), (4.6), and (4.7) is usually obtained using MoM. Substituting \( \mathbf{J}(r) \) in (4.7), and using Galerkin testing leads to a matrix equation of the form

\[
\mathbf{Z}_c \mathbf{I} = \mathbf{F}_c
\]

where

\[
\mathbf{I} = [I_1, I_2, \ldots, I_N], \quad \mathbf{F}_{c,m} = \langle \mathbf{S}_m(r), \mathbf{V}_e \{ \mathbf{E}^i(r), \mathbf{H}^i(r) \} \rangle, \quad \text{and} \quad \mathbf{Z}_{c,nn} = \langle \mathbf{S}_m(r), \mathcal{L}_c \{ \mathbf{S}_n(r) \} \rangle.
\]

Here \( \langle \ldots \rangle \) denotes a standard inner product.

We describe how FMM is used to solve the matrix equation. FMM relies on the fact that the fields radiated by a spatially bound source are quasi-bandlimited, and can be expressed in terms of plane waves to an arbitrary degree of accuracy. Consider two spheres of radius \( R_s \), centered at \( \mathbf{r}_s^s \) and \( \mathbf{r}_s^o \) and labelled \( \alpha_s \) and \( \alpha_o \), that contain a set of sources and observers. The field radiated by a source basis function \( \mathbf{S}_n(r) \) for \( n \in \alpha_s \) is tested by \( \mathbf{S}_m(r) \) for \( m \in \alpha_o \). In other words, one needs to evaluate \( \langle \mathbf{S}_m(r), \mathcal{L}_c \{ \mathbf{S}_n(r) \} \rangle \). It can be shown that

\[
\langle \mathbf{S}_m(r), \mathcal{L}_c \{ \mathbf{S}_n(r) \} \rangle = \frac{1}{8\pi^2 c^2} \sum_{k=0}^{M} \sum_{p=-M}^{M} w_{kp} \left[ -\beta S_{m}^{-} \left( \hat{k}_{kp}, \hat{k}_{kp} \right) + S_{m}^{-} \left( \hat{k}_{kp}, \hat{n}_{kp} \right) \right]^T \mathbf{T} \left( \hat{k}_{kp}, M \right) \left[ S_{n}^{+} \left( \hat{k}_{kp}, \hat{k}_{kp} \right) \right] I_n
\]

In (4.9), \( M = [2k\chi R_s + 1] \) is the number of harmonics, \( \chi \) is an oversampling factor, \( w_{pq} \) are quadrature weights, and \( \hat{k}_{kp} \) are the plane wave directions [58]. The translation operator
The terms \( \mathcal{T}(\mathbf{k}_p, M) \) and the slant-stack operator are given as

\[
\mathcal{T}(\mathbf{k}_p, M) = -\omega^2 \sum_{\nu=0}^{M} (-j)^{\nu} (2\nu + 1) h_\nu^{(2)}(k|\mathbf{R}_{\alpha_0\alpha_s}^c|) P_\nu \left( \mathbf{k}_p \cdot \mathbf{R}_{\alpha_0\alpha_s}^c / |\mathbf{R}_{\alpha_0\alpha_s}^c| \right) \tag{4.10a}
\]

for \( \nu = \{m,n\} \), \( \mathbf{R}_{\alpha_0\alpha_s}^c = \mathbf{r}_e^c - \mathbf{r}_s^c \), \( P_\nu(\cdot) \) is a Legendre polynomial of degree \( \nu \), and \( h_\nu^{(2)}(\cdot) \) is a Hankel function of the second kind. Equation (4.10b) indicates that the field radiated by all \( n \in \alpha_s \) may be computed by first projecting it on to a set of plane waves that travel out of the source sphere \( \alpha_s \). These are then translated onto another set of plane waves that enter an observation sphere using a diagonal translation operator. Finally, the incoming rays are projected on to all basis \( m \in \alpha_o \). This completes the evaluation of a matrix-vector product. This procedure can be nested within itself and one can arrive at a hierarchical procedure that scales as \( O(N \log N) \) for computing scattering from surface unknown distributions.

### 4.2 FMM Based CEM Solver

The scatterer is enclosed in a fictitious cubical domain. The solution to (4.8) is typically obtained using an iterative solver like TFQMR (Transpose Free Quasi-Minimal Residual). The matrix vector product that is required by such an iterative solver is computed as follows: The matrix is decomposed into a nearfield component and a farfield component. The nearfield portion of the matrix represents the interactions between basis functions that are "close" (to be made precise later) to each other, and is computed once and used in every iteration. The product of the farfield component of the matrix multiplied by the vector is directly computed using the FMM algorithm in each iteration. Computation of the farfield component of the matrix vector product comprises of the following steps:

- **Computing interpolations**: The spectrum of electromagnetic field radiated by the sources that reside in a cell, is represented using a set of sample points. These points correspond to plane wave directions \([17, 33]\). The number of sample points is proportional to the square of the side length of the cell. The work done at each level depends on the
number of sample points and hence the problem is often referred to as frequency domain FMM, different from the static FMM used to solve the N-body problem. For the leaf nodes, sample points are computed from the basis functions in the leaf node. For each internal node, sample points are computed from sample points of its child nodes using an operation called interpolation. This involves interpolating sample points of the children to obtain values along a set of directions associated with the parent node, shifting to the center of the parent node and then merging. Interpolation can be done by using a straightforward algorithm \(O(k^2)\) time), an FFT based algorithm \(O(k \log k)\) time), or a Lagrange interpolation \(O(k)\) time), where \(k\) is the total number of original and interpolated sample points. In our CEM solver, we use an FFT/band-limited algorithm. The sample points for all the nodes are computed using a bottom-up tree traversal.

- **Computing translations:** After interpolation, the field is computed at all the nodes. A node \(a\) is said to be in the nearfield of \(b\), if \(a\) and \(b\) are of the same size and \(a\) is adjacent to \(b\), i.e, if \(a\) lies within a shell around the node \(b\). Otherwise, they are said to be in the farfield of each other. Nodes in the farfield of another node form the interaction list of the node. For each node, the fields from the nodes in its interaction list are translated. This phase is called translation. The phrase partial farfield is used to denote the result of such interactions. At the end of this phase, the partial farfield of all the nodes is known.

- **Anterpolations:** The far field at a node is the result of aggregating the results of translations in all its ancestral nodes. This is computed using a top-down traversal. For each node, the results of translations from its interaction list are combined with the parent’s far field (by approximately halving the number of sample points and shifting the field to the center of the child node) to compute its far field. This operation is known as anterpolation. The sample points at the leaf nodes are then projected back to the basis functions.

Care must be taken while performing the interpolation and anterpolation operations, especially for vector problems. Typically, one would translate all three Cartesian components;
however, note that (4.10b) can be expressed in terms of $\hat{\theta}$ and $\hat{\phi}$ components only. Thus, it is possible to evaluate (4.9) using only two components. It should be noted that if one were to use a scalar spherical filter [31] for interpolation/anterpolation, all three components are required. Alternatively, one needs to use a vector spherical filter if only two components are to be used [57].

Thus, the algorithm for FMM based CEM solver is: 1) Building the compressed octree, 2) Computing nearfield matrix, 3) Precomputing the interactions lists for all nodes, 4) Computing the nearfield matrix vector product, 5) Computing interpolations using a bottom-up traversal, 6) Computing translations for each node using its interaction list, 7) Computing the anterpolations using a top-down traversal, 8) Projecting the field at leaf nodes back to the observer basis functions using diagonal translation operator, and 9) Repeating steps 4 through 8 until convergence.

4.3 Tree Construction

Tree construction is the first step in all hierarchical applications. As will be evident later, tree construction requires an insignificant amount of the total run-time. The algorithm takes as input the wavelength, $\lambda$. The length of the smallest cell is approximately $0.4\lambda$. The size of the smallest cell and the length of the cubic region enclosing all the points, determine the number of levels ($L$) in the tree.

Each processor sets electromagnetic constants, computes $x, y, z$ components of the incident field, computes lengths of all edges and initializes arrays for storing node, patch and edge information. Each basis function is associated with a number and a type (source or observer type) that uniquely identifies it. The basis functions are distributed among the processors based on the numbers. Each processor is initially given $\frac{N}{P}$ source basis functions. Every processor is also given an equal number of observer basis functions. The location of the basis functions is used to generate the corresponding leaf node. For every source basis function, there is an observer basis function which maps to the same leaf node as the source basis function. We construct the tree in parallel using the algorithm outlined in the previous chapter.
4.4 Near Field Computation

In this section, we describe use of our neighbor query library function to compute the nearfield component. The nearfield portion of the matrix is a consequence of the interactions between basis functions from the same or adjacent leaf nodes. Direct computation is used for nearby interactions. Since the geometry is fixed throughout the number of iterations required for solving the matrix vector equation, the nearfield matrix is built only once and used in every iteration.

Let \( A \) be an array size \( O(N) \) that stores the basis functions. The basis functions carry the information about the leaf cell that they belong to. The array \( A \) is also sorted along with the leaf cell array during the parallel tree construction phase. As a result, basis functions whose averaged centers are contained in the same leaf cell occupy consecutive positions in \( A \). Hence the mapping of the leaf cells to the basis functions that belong to it, can be easily maintained. Each processor computes part of the nearfield portion of the matrix, corresponding to the basis functions in its leaf nodes. Therefore, accessing information from basis functions in adjacent leaf nodes is necessary. The basis functions required at a processor can be separated into two categories: those that are available locally within the same processor, and those that reside in other processors. In order to fetch the information at the adjacent basis functions from other processors, the neighbor query algorithm described in the previous chapter is used.

Note that a leaf node information sent to a processor may not be needed, because the adjacent leaf node prompting the communication may be non-existent. Similarly, an expected leaf node may not arrive because it is non-existent. These can be easily detected by ordering the received leaf nodes in postorder traversal order. The leaf nodes received from each processor are already in this order. Hence, merging the received lists of leaf nodes is enough to create the required array \( C \) that contains all the information required by a processor to begin near-field computation. Information in the leaf nodes not local to the processor is sought by doing binary searches in \( C \).

The number of leaf nodes sent by or received on a processor is bounded by \( O(N/p) \). Although a better bound can be established, and the number of out of processor leaf nodes necessary
will be even smaller because of the locality preserving properties of the Morton ordering, this is sufficient to prove that run-time complexity of this phase is $O\left(\frac{N}{P} \log \frac{N}{P}\right)$. This is because each leaf node has atmost 26 neighboring leaf nodes and the algorithm performs 26 binary searches for each of its leaf nodes. Work per leaf node is proportional to the number of basis functions that belong to it and the number of basis functions that belong to the leaf nodes adjacent to it. If each leaf node has atmost $c$ basis functions, then the number of interactions per leaf node in this phase is atmost $27 \cdot c^2$.

4.5 Farfield computation

4.5.1 Building Interaction Lists

For each node, the interaction list specifies a list of nodes which are in its farfield but whose parent nodes are in the nearfield of its parent. As with the nearfield matrix computation, the interaction lists are also built just once and used in every iteration. But the interaction lists are built for all the nodes in the tree and not just the leaf nodes.

In a compressed octree, the parent node, corresponding to the node of the parent node in the tree, need not necessarily be a node of double the side length and may reside on some other processor. Therefore, it is convenient to compute the parent node using the following lemma.

Lemma 6 The parent of a node is the smallest node containing it and its adjacent node in the postorder traversal order.

Proof: If the node is the rightmost child of its parent, then the parent node will be adjacent to it in the postorder traversal order. In this case, the smallest node containing the child node and its adjacent node will be the adjacent node itself, which is the parent. If the node is not the rightmost child of its parent, then the adjacent node will be the leftmost leaf node in the subtree of its next sibling. The lowest common ancestor of these two nodes will be the parent node, which proves the lemma.

Following are the steps to compute the interaction list of node, $b$: 


1. Using the above lemma, compute the parent node of $b$.

2. Obtain the nodes in the nearfield of the parent node using bit operations.

3. Compute subcells of the nodes obtained in Step 2 such that size of each subcell is equal to the size of $b$. Discard those subcells that are in the near field of $b$.

4. Partition these subcells into two arrays for convenience: an array $Local$ for subcells that are local to the processor, and an array $Remote$ for subcells that are remote to the processor. The $Remote$ array will thus have all the nodes whose information needs to be fetched at the translation phase of each iteration.

Following the building of interaction lists, multiple iterations of the remaining stages are run until convergence. At the end of each iteration, the difference between the matrix vector product and the right hand side is used to improve the solution vector.

4.5.2 Computing Interpolations

For the leaf nodes, the sample points are computed from the basis functions in the leaf node. The work involved in computing the sample points at a leaf node is proportional to the product of the number of basis functions and the number of sample points. The number of sample points are proportional to the side length of the node. The number of leaf nodes and the total number of nodes are distributed fairly well among the processors and so the work done is balanced. The interpolation operation has the associative property. Because of this property, interpolations on the residual tree can be computed using the parallel upward tree accumulation algorithm described in previous chapter.

4.5.3 Computing Translations

To compute translations at a node, the field at each of the nodes in its interaction list is needed. The interaction lists built in the preprocessing stage are used for this stage. Note that the size of the interaction list for a node depends on the spatial distribution. The neighbor query function is used to fetch fields of nodes from the interaction lists that reside on remote
processors. Because the interaction lists remain the same through all the iterations, requests for nodes in interaction lists need to be done only in the first iteration, thereby reducing the communication to one MPI_Alltoall in subsequent iterations.

A query node generated during building interaction lists could be empty or could contain one subcell with the remaining region being empty (see Figure 4.1). It is easy to modify binary search algorithm of the neighbor query function on the postorder traversal to ensure that a search for such a node retrieves the corresponding subcell. In Figure 4.1, interaction list of v contains node c. However, c is not present in the tree because it is empty except for the region given by subcell e. Instead of c, subcell e should be returned in response to the query. Once all the information is available locally, translations are conducted within each processor. Each processor performs $O\left(\frac{N}{P}\right)$ translations. Note that translations may be needed between nodes of different sizes. Oversampling is used when a smaller node is translated to a bigger node.

4.5.4 Computing Anterpolations

Similar to the interpolation operation, it can be shown that the anterpolation operation is also associative. Taking advantage of this property, anterpolations are also computed using the algorithm described earlier. The sample points at the leaf nodes are then projected back to the basis functions.
4.6 Experimental Results

We developed parallel code that takes the geometry of the surface, details of the impinging wave etc. into account and computes the electromagnetic scattering from the surface. It is a combined C and Fortran code using MPI for parallel communication. The numerical kernels, such as those for translation, interpolation and anterpolation, are coded in Fortran. The remaining part of the program is coded in C. It is a complete end-to-end application software with controllable accuracy, incorporating all the parallel algorithms described earlier and other necessary tasks such as the creation of basis functions themselves from the geometry etc. We had earlier developed a serial code, which is used both to validate the parallel software and to compute the runtime speedups achieved by it.

We experimentally tested the software on the IBM xSeries cluster using two different types of surfaces: 1) A sphere and a 2) a 3-dimensional L-shaped surface. The spherical surface is used as a representative uniform distribution and the L-shaped surfaces serves as a non-uniform distribution. The run-times for solving the complete application for 20,000 and 40,000 unknowns for both uniform and non-uniform distributions as a function of the number of processors are shown in Figure 4.2. From the graphs, we see that the algorithm works well for both types of distributions, as predicted by the theoretical analysis. It also scales well with number of processors.
Table 4.1  Run-times in seconds for various stages of the algorithm for 20,000 unknowns for uniform distribution.

<table>
<thead>
<tr>
<th>No. of procs</th>
<th>Tree const.</th>
<th>Nearfield time</th>
<th>Building interaction lists</th>
<th>Interpolation</th>
<th>Translation</th>
<th>Anterpolation</th>
<th>Total</th>
<th>Speed up</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.25</td>
<td>381.06</td>
<td>0.10</td>
<td>109.97</td>
<td>116.20</td>
<td>119.39</td>
<td>726.97</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0.65</td>
<td>174.46</td>
<td>0.76</td>
<td>55.54</td>
<td>73.93</td>
<td>60.82</td>
<td>366.16</td>
<td>1.99</td>
</tr>
<tr>
<td>4</td>
<td>0.33</td>
<td>87.72</td>
<td>0.39</td>
<td>28.44</td>
<td>46.50</td>
<td>30.66</td>
<td>194.04</td>
<td>3.75</td>
</tr>
<tr>
<td>8</td>
<td>0.16</td>
<td>44.95</td>
<td>0.20</td>
<td>14.54</td>
<td>28.11</td>
<td>16.02</td>
<td>103.98</td>
<td>6.99</td>
</tr>
<tr>
<td>16</td>
<td>0.07</td>
<td>23.13</td>
<td>0.12</td>
<td>7.66</td>
<td>15.68</td>
<td>8.30</td>
<td>54.96</td>
<td>13.23</td>
</tr>
<tr>
<td>32</td>
<td>0.05</td>
<td>12.38</td>
<td>0.06</td>
<td>4.71</td>
<td>11.78</td>
<td>4.95</td>
<td>33.93</td>
<td>21.42</td>
</tr>
<tr>
<td>64</td>
<td>0.05</td>
<td>6.41</td>
<td>0.04</td>
<td>2.77</td>
<td>7.33</td>
<td>2.77</td>
<td>19.37</td>
<td>37.53</td>
</tr>
</tbody>
</table>

Table 4.2  Run-times in seconds for various stages of the algorithm for 20,000 unknowns for non-uniform distribution.

<table>
<thead>
<tr>
<th>No. of procs</th>
<th>Tree const.</th>
<th>Nearfield time</th>
<th>Building interaction lists</th>
<th>Interpolation</th>
<th>Translation</th>
<th>Anterpolation</th>
<th>Total</th>
<th>Speed up</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.21</td>
<td>817.30</td>
<td>0.03</td>
<td>130.13</td>
<td>30.36</td>
<td>140.68</td>
<td>1118.71</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0.64</td>
<td>368.35</td>
<td>0.12</td>
<td>65.96</td>
<td>20.77</td>
<td>70.04</td>
<td>525.88</td>
<td>2.13</td>
</tr>
<tr>
<td>4</td>
<td>0.33</td>
<td>190.51</td>
<td>0.07</td>
<td>33.78</td>
<td>16.06</td>
<td>36.04</td>
<td>276.79</td>
<td>4.04</td>
</tr>
<tr>
<td>8</td>
<td>0.17</td>
<td>98.22</td>
<td>0.04</td>
<td>17.65</td>
<td>10.11</td>
<td>18.93</td>
<td>145.12</td>
<td>7.71</td>
</tr>
<tr>
<td>16</td>
<td>0.09</td>
<td>50.44</td>
<td>0.03</td>
<td>9.56</td>
<td>6.33</td>
<td>9.85</td>
<td>76.30</td>
<td>14.66</td>
</tr>
<tr>
<td>32</td>
<td>0.06</td>
<td>28.04</td>
<td>0.02</td>
<td>5.76</td>
<td>4.34</td>
<td>5.44</td>
<td>43.66</td>
<td>25.62</td>
</tr>
<tr>
<td>64</td>
<td>0.06</td>
<td>21.03</td>
<td>0.01</td>
<td>3.82</td>
<td>2.72</td>
<td>3.48</td>
<td>31.12</td>
<td>35.94</td>
</tr>
</tbody>
</table>

The breakdown of the run-times into times spent in various stages of the application for solving the problem with 20,000 unknowns are shown in Table 4.1 and Table 4.2 for uniform and non-uniform distributions, respectively. We show the breakdown of the run-times for 20,000 unknowns to illustrate that our algorithm scales very well even for a small problem size. The nearfield and farfield components take approximately half the running time each, with a negligible amount of time spent in constructing the tree and building the interaction lists. The last column in the tables lists the speedup when compared to running the serial code for the same problem on one processor of the cluster.
Table 4.3 Run-times in seconds for various stages of the algorithm for 1,000,000 unknowns for uniform distribution.

<table>
<thead>
<tr>
<th>No. of procs</th>
<th>Tree const.</th>
<th>Nearfield time</th>
<th>Building interaction lists</th>
<th>Interpolation</th>
<th>Translation</th>
<th>Anterpolation</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>6.27</td>
<td>636.32</td>
<td>14.80</td>
<td>993.66</td>
<td>5543.02</td>
<td>1051.46</td>
<td>8245.53</td>
</tr>
<tr>
<td>64</td>
<td>2.47</td>
<td>329.60</td>
<td>8.79</td>
<td>584.69</td>
<td>3276.59</td>
<td>600.88</td>
<td>4803.02</td>
</tr>
</tbody>
</table>

Figure 4.3 Run-time as a function of the problem size on 16 processors for (i) one iteration of farfield computation and (ii) total application.

Table 4.3 shows the run-times of the various stages for a problem size of 1,000,000 unknowns for a uniform distribution. As expected, with increase in problem size, most of the run-time is spent in the farfield component. The scaling of the run-time with problems for a fixed number of processors is shown in Figure 4.3. The graph to the left shows run-time for one iteration of farfield computation. The graph to the right shows the total application run-time. The number of iterations for convergence increases with increase in the number of unknowns and this behavior is reflected in the growth of the application run-time.

Finally, to demonstrate that results generated by this code are indeed valid, results generated by our parallel software are compared against analytical solutions for canonical problems. We chose to compute the field scattered by a sphere of radius 1m illuminated by an incident plane wave propagating in the $-\hat{z}$ direction, and is $\hat{x}$ polarized. The sphere is discretized using 518,310 basis functions and frequency of the incident wave is 4.2GHz. As is evident from
Figure 4.4  Comparison between analytically and numerically obtained RCS data for a sphere of radius 1.0m discretized using 518,310 unknowns.

Figure 4.4, the agreement between the analytical and numerical RCS data is excellent.
CHAPTER 5. CONCLUSIONS AND FUTURE WORK

We presented the design and implementation of provably efficient parallel algorithms for compressed octrees and various queries on them. Our algorithms perform well even for non-uniform distributions and have little communication overhead. The data partitioning technique used requires no explicit load balancing in any stage of the algorithm. Also it gives good load balance and communication efficiency for all hierarchical methods unlike certain other data partitioning schemes that work best only for a particular hierarchical method and lead to lot of communication overhead when applied to other methods. We developed a parallel software library that uses our efficient algorithms and encapsulates most of the parallelization necessary in building applications using hierarchical methods. This significantly reduces the effort needed to develop a parallel application. We demonstrated this by developing a scalable, parallel computational electromagnetics solver for computing the scattering from three dimensional perfect electrically conducting surfaces using the library. The software can produce results with controllable accuracy and has been validated against analytical solutions and serial software.

In what follows, we highlight some of the work that remains to be done:

- We achieved a remarkable reduction in the number of communication rounds for computing tree accumulations which are the most communication demanding stages in any application. The number of communication rounds for our algorithm is bounded by $O(\log P + \log L)$. For a uniform distribution, the height of the tree is $O(\log N)$, reducing the number of communications to $O(\log \log N)$. For an exponential distribution, the height of the tree is $\Omega(N)$, suggesting that the worst-case number of communications is $\Omega(\log N)$. But the actual number of communication rounds required by the algorithm for the exponential case is be just one. Based on a number of problem instances we
have seen, we conjecture that the worst-case number of communications is $O(\log \log N)$, though we do not have a proof of this.

- The functionality of the library can be enhanced by including queries like spherical region queries, nearest neighbor queries, and k-nearest neighbor queries. The library can also be enhanced to support dynamic compressed octrees by adding functions such as insertions and deletions of cells in the domain.

Some open problems and future work for the parallel electromagnetics code developed using the library functions are:

- The initial set up phase which involves preprocessing of the geometry is serially performed on every processor. This takes several hours for a large problem size. This time has not been shown intentionally in the experimental section of this thesis as our aim was to show the scalability of the algorithms designed. We do hope to parallelize the geometric set up phase in the future.

- Currently the translation operators are computed on every processor. Parallelizing this phase and precomputing the translation operators would further enhance the performance of the electromagnetics code.

- The work done for a cell or node at each level depends on the number of sample points which is proportional to the side length of the box. Therefore the work done for boxes at higher levels of the tree is more than that for lower level boxes. Our tree accumulation algorithms are distribution independent and give good load balance during upward and downward accumulation for the static FMM. However for frequency domain FMM, depending on the spatial distribution, a single processor may get too many higher level nodes leading to load imbalance.

- The size of the interaction list for a node and therefore the work done for the node during the translation phase of the FMM depends on the spatial distribution. Though the data partitioning scheme distributes the total number of nodes fairly well among the processors, the load balance during translation phase remains distribution dependent.
Bibliography


ACKNOWLEDGEMENTS

I am deeply indebted to my thesis advisor Dr. Srinivas Aluru for giving me an opportunity to work with him and for introducing me to the world of research and parallel computing. I thank him for his guidance and support in making this research possible and for being a wonderful source of inspiration and encouragement throughout. His expertise in the area of parallel algorithms and scientific computing and helpful suggestions and comments always kept up my enthusiasm for research.

I am extremely grateful to Dr. Shanker Balasubramaniam for his assistance and constant involvement in my research. I would like to thank him for spending his valuable time in making me understand the concepts involved in computational electromagnetics, for providing the numerical kernels of the electromagnetics code used in developing the parallel electromagnetics code and for all his help in developing and validating the software. Without his help and guidance, the interdisciplinary project would not have been possible. I wish to thank Dr. G.M.Prabhu for being on my committee and for his valuable time.

I would like to express my gratitude to all members of my research group who contributed in one way or the other to the success of this thesis. And most importantly, I sincerely thank my family and my fiance for being extremely supportive and for their never-ending words of motivation and encouragement needed to complete my graduate studies.