# A Provably Optimal, Distribution-Independent Parallel Fast Multipole Method*

Fatih E. Sevilgen
Syracuse University
School of EECS
Syracuse, NY 13244
sevilgen@ecs.syr.edu

Srinivas Aluru
Iowa State University
Dept. of ECpE
Ames, IA 50011
aluru@iastate.edu

Natsuhiko Futamura
Syracuse University
School of EECS
Syracuse, NY 13244
nfutamur@ecs.syr.edu

## Abstract

*The Fast Multipole Method (FMM) is a robust technique for the rapid evaluation of the combined effect of pairwise interactions of n data sources. Parallel computation of the FMM is considered a challenging problem due to the dependence of the computation on the distribution of the data sources, usually resulting in dynamic data decomposition and load balancing problems. In this paper, we present the first provably efficient and distribution-independent parallel algorithm for the FMM on distributed memory parallel computers. Our algorithm does not require any dynamic data decomposition or load balancing step. We present our algorithm in terms of a few basic and well understood primitive operations such as sorting and parallel prefix.*

## 1 Introduction

The Fast Multipole Method (FMM) is a robust technique for the simulation of the evolution of $n$ data sources under the influence of pairwise interactions. The method was first developed by Rokhlin [11] and shot into prominence with Greengard's application of the FMM [6] to solve the N-body problem. Since then, the FMM has been applied to solve a number of problems in fluid dynamics, molecular dynamics, elliptic partial differential equations, integral equations, electromagnetic scattering, and even computer graphics.

A straightforward simulation by considering all pairwise interactions requires $O(n^2)$ work per simulation step. The rapid growth with $n$ effectively limits the number of particles that can be simulated by this method. The essential idea of the FMM is to approximate interactions between distant clusters of particles instead of computing individual pairwise interactions. To enable a recursive application of this idea, a hierarchical tree structure, typically an octree is used [6].

The octree-based FMM is often incorrectly claimed to reduce the computation required per iteration from $O(n^2)$ to $O(n)$. However, it has been shown that the run-time depends upon the distribution of the particles [1]. Furthermore, the costs involved in computing the data structure itself are often ignored. Aluru *et al.* [2] use compressed octrees and modify the FMM algorithm so that it runs in $O(n)$ time. The construction of the compressed octree itself takes $O(n \log n)$ time. Callahan and Kosaraju [4] developed a different data structure known as the Fair Split Tree, and present an FMM algorithm using this data structure having the same time bounds. These two are the only sequential FMM algorithms known to be optimal and distribution-independent.

Considerable research effort is directed at developing parallel implementations of the FMM. Certain fundamental characteristics of the FMM translate to difficult challenges for efficient parallelization. The FMM computation consists of a tree construction phase followed by a force computation phase. The data decomposition required for efficient tree construction may conflict with the data decomposition required for force computation. The distribution of the particles changes from iteration to iteration, which has the effect of changing the workload distribution. Due to these difficulties, the algorithms usually depend upon heuristic methods to perform data decomposition and entail unstructured and irregular communication. Such heuristic methods make it impossible to rigorously analyze the run-time of the parallel algorithms.

Most of the parallelizations employ the octree-based FMM computation, and thus inherit the distribution-dependent nature of the algorithm. The run-times of these algorithms are either not analyzed due to aforementioned difficulties or analyzed only for uniform distributions. The only exception is an algorithm developed by Teng [12], in which he defines a communication graph to capture the communication pattern underly-

*Research supported in part by ARO under DAAG55-97-1-0368, NSF CAREER under CCR-9702991 and Sandia National Laboratories.

ing the FMM computation and presents an algorithm to partition this graph to achieve balanced load and communication simultaneously. The degree of non-uniformity in the distribution of the particles is parameterized by $\mu$, such that the height of the octree for the particles is $O(\log n + \mu)$. The number of operations executed by the algorithm is a function of $\mu$ and hence the algorithm is distribution-dependent.

Callahan and Kosaraju developed a distribution-independent parallel algorithm for the FMM using their sequential framework. The algorithm runs on the CREW PRAM model in $O(\log^2 n)$ time, using $O(n)$ processors [4]. The PRAM model ignores problems associated with data decomposition and cost of communication, and hence does not capture the essential characteristics of parallel computers. A straightforward translation of their algorithm does not result in an efficient implementation on parallel computers.

Our interest lies in designing a parallel FMM algorithm that is distribution-independent and rigorously analyzable. We present such an algorithm in this paper. It is a parallelization of the sequential framework proposed by Aluru *et al.* [2] and is developed taking into detailed account the physically distributed nature of memory in parallel computers. It is presented in an architecture-independent fashion, using only well-understood and basic communication operations such as parallel prefix, all-to-all communication and sorting. It uses only a static data decomposition and does not require any explicit dynamic load balancing, either within an iteration or across iterations. The algorithm can be efficiently implemented on any model of parallel computation that admits an efficient sorting algorithm.

## 2 Preliminaries

We use the *permutation network* as our model of parallel computation. In this model, each processor is allowed to send and receive at most one message during a communication step. The cost is modeled as $\tau + \mu l$, where $\tau$ is the start-up time to initiate a communication, $\frac{1}{\mu}$ is the data rate supported by the network, and $l$ is the length of the largest message. This corresponds to the assumption that communication corresponding to any permutation can be realized simultaneously. The permutation network model closely reflects the behavior of most multistage interconnection networks.

Our algorithms are stated in terms of the following well-known parallel primitive operations ($p$ denotes the number of processors). For a detailed description and run-time analysis, the reader is referred to [8].

**Parallel Prefix:** Consider $n$ data items $x_1, x_2, \ldots, x_n$ and a binary associative operator $\otimes$. The prefix sums problem is to compute $s_1, s_2, \ldots, s_n$, where $s_i = x_1 \otimes x_2 \otimes x_3 \otimes \ldots \otimes x_i$. The $s_i$'s are often called partial sums. Parallel prefix is the problem of computing the prefix sums in parallel. This problem can be solved in $O(\frac{n}{p} + (\tau + \mu)\log p)$ time.

**Segmented Parallel Prefix:** Segmented prefix computation is a sequence of prefix computations using an associative operator $\otimes$. Consider $n$ data items $x_{1,1}$, $x_{1,2}$, ..., $x_{1,n_1}$, $x_{2,1}$, $x_{2,2}$, ..., $x_{2,n_2}$, ..., $x_{m,1}$, $x_{m,2}$, ..., $x_{m,n_m}$, where $\Sigma_{i=1}^{m} n_i = n$. We want to compute $s_{i,j}$, where $s_{i,j} = x_{i,1} \otimes x_{i,2} \otimes x_{i,3} \otimes \ldots \otimes x_{i,j}$ for $1 \leq i \leq m, 1 \leq j \leq n_i$. This operation can be done using one parallel prefix operation and takes $O(\frac{n}{p} + (\tau + \mu)\log p)$ time.

**All-to-All Communication:** In this operation each processor sends a distinct message of size $m$ to every processor. This operation takes $O((\tau + \mu m)p)$ time.

**Transportation Primitive:** The transportation primitive performs many-to-many personalized communication with possibly high variance in message sizes. If the total length of the messages being sent out or received at any processor is bounded by $t$, the transportation primitive performs the communication using two all-to-all communications with a uniform message size of $\frac{t}{p}$ [10].

**Sorting:** Using sample sort [8] in conjunction with bitonic sort for sorting splitters identified during sample sort, $n$ elements can be sorted in $O\left(\frac{n \log n}{p} + \tau p + \mu\left(\frac{n}{p} + p\log^2 p\right)\right)$ time for $n > p^2 \log p$.

## 3 Fast Multipole Method

In this section, we describe the sequential FMM, compressed octrees and the adaptation of the sequential FMM to compressed octrees (for a detailed discussion, see [2, 6]). Following Greengard [6], the FMM is presented in the context of the N-body problem.

We use the term *cell* to denote a cubical 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 the immediate subdivision of another cell into 8 cells having half the side length of the original cell. The terms *supercell* and *immediate supercell* are defined similarly. The *length of a cell* is the span of the cell along any dimension. Consider a cell large enough to contain the $n$ particles. Subdivide the cell into its immediate subcells. Discard cells that do not contain any particle. Recursively subdivide the cells that contain more than one particle. Stop the subdivision process on cells having exactly one particle. This recursive subdivision of the space into cells is represented by a tree, which is called the octree. A

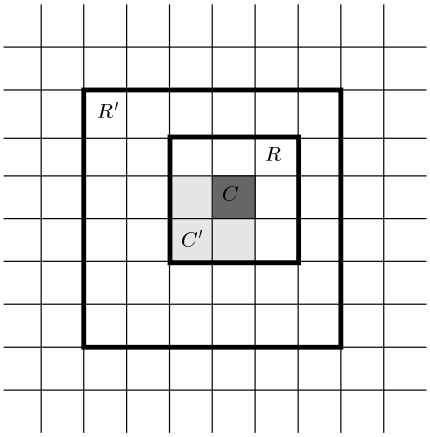


**Figure 1. Illustration of proximity set and interaction set in two dimensions.**

cell that is not further subdivided is a *leaf cell*.

In FMM, each cell $C$ in the octree is associated with a multipole expansion and a local expansion, denoted by $\phi(C)$ and $\psi(C)$, respectively. $\phi(C)$ is a truncated infinite series that describes the effect of the particles within cell $C$ at a distant point. The multipole expansions are computed using a bottom-up traversal in time linear in the size of the tree. The multipole expansion at a leaf cell is directly computed. The multipole expansion for an internal cell is computed by aggregating the multipole expansions of its immediate subcells.

The local expansion $\psi(C)$ is a truncated infinite series that describes the effect of all distant particles on the points within the cell $C$. It is obtained by appropriately combining (referred to as 'addition', for simplicity) those multipole expansions that converge at every point within cell $C$. In Figure 1, $\psi(C)$ should include the effect of all particles outside region $R$. However, the effect of all particles outside region $R'$ are included in $\psi(C')$, where $C'$ is the parent of $C$. When considering cell $C$, we only compute the effect of particles that are outside region $R$ but within region $R'$. We call this the *partial local expansion* at cell $C$. This is added to $\psi(C')$ to compute $\psi(C)$. To compute local expansions for all cells, a top-down traversal is used. The time for computing all the local expansions is proportional to the size of the octree.

In an octree, all the particles that lie within a cell may also lie within one of its immediate subcells. This leads to *chains* in the tree, where each node on a chain (except the last node) has exactly one child. Such a chain can be arbitrarily large irrespective of the total number of particles. Therefore, the size of the octree and the time for multipole and local expansion calculations depends upon the distribution of the particles.

**Algorithm 1 Compute-Local-Exp ($v$)**

I. Find the proximity set $P(v)$ and the interaction set $I(v)$ for $v$
  $E(v) = P(parent(v))$
  $I(v) = \emptyset$; $P(v) = \emptyset$
  While $E(v) \neq \emptyset$ do
    Pick some $u \in E(v)$
    $E(v) = E(v) - \{u\}$
    If $well\text{-}sep(S(v), S(u))$
      $I(v) = I(v) \cup \{u\}$
    Else if $S(u)$ is smaller than $S(v)$
      $P(v) = P(v) \cup \{u\}$
    Else $E(v) = E(v) \cup children(u)$
II. Calculate the local expansion at $v$
  Initialize $\psi(v)$ using $\psi(parent(v))$
  For each node $u \in I(v)$
    Incorporate $\phi(u)$ into $\psi(v)$
III. Calculate the local expansions at the children of $v$ with recursive calls
  For each child $w$ of $v$
    **Compute-Local-Exp ($w$)**

**Figure 2. Algorithm for calculating local expansions of all nodes in the tree rooted at $v$.**

To achieve optimal run-time independent of the distribution, we use the compressed octree data structure. Let $v_1, v_2, ..., v_k$ $(k \geq 2)$ be a *maximal chain* in the octree such that each node of the chain represents the same set of particles. In a compressed octree, each such chain is replaced by the last node on the chain, i.e., $v_k$. The size of a compressed octree is $O(n)$, where $n$ is the number of particles. It is not a height-balanced tree and its height could be as large as $\Omega(n)$. For each node in a compressed octree, we define two cells: The *small cell* at a node is the smallest subcell that contains all the particles in its subtree. The *large cell* at a node is the largest cell the node is responsible for. It is obtained by taking the appropriate immediate subcell of the small cell at its parent node. For a node $v$, we use $L(v)$ to denote the large cell and $S(v)$ to denote the small cell.

Our compressed octree algorithm is as follows: For each node $v$, compute the multipole expansion $\phi(v)$ and the local expansion $\psi(v)$, with respect to the cell $S(v)$. The multipole expansions can still be computed by a simple bottom-up traversal in $O(n)$ time.

In the octree-based FMM algorithm, cells of the same length are used to compute local expansions. For the compressed octree, a capability to deal with different cell lengths is needed. For two cells $C$ and $D$,

define a predicate *well-sep*$(C, D)$ to be true if $D$'s multipole expansion converges at any point in $C$, and false otherwise. If two cells are not well-separated, they are *proximate*. Similarly, for two nodes $v_1$ and $v_2$ in the compressed octree, $v_2$ is said to be *well-separated* from $v_1$ if and only if *well-sep*$(S(v_1), S(v_2))$. Otherwise, we say that $v_2$ is *proximate* to $v_1$. In the octree-based FMM, the set of cells that are proximate to the cell $C$ is called *the proximity set of* $C$ and is defined by $P^{=}(C) = \{D \mid length(C) = length(D), \neg well\text{-}sep(C, D)\}$. The superscript "=" is used to indicate that cells of same length are being considered. For a node $v$ in the compressed octree, define the *proximity set* $P(v)$ as the set of all nodes proximate to $v$ and having the small cell no larger than and the large cell no smaller than $S(v)$. More precisely, $P(v) = \{u \mid \neg well\text{-}sep(S(v), S(u)), length(S(u)) \leq length(S(v)) \leq length(L(u))\}$.

The algorithm to compute the local expansions is given in Figure 2. The computations are done using a top-down traversal of the tree. To compute partial local expansion at a node $v$, we have to consider the set of nodes that are proximate to its parent, i.e., $P(parent(v))$. We recursively decompose nodes in $P(parent(v))$ until each node is either 1) well-separated from $v$ or 2) proximate to $v$ and the length of the small cell of the node is smaller than the small cell of $v$. Partial local expansion is computed by adding the multipole expansions of the nodes in category 1, and the set of these nodes is the *interaction set* of $v$. Each node $w$ in the interaction set of $v$ is either in $P(parent(v))$ or is in the subtree of a node in $P(parent(v))$ such that no ancestor of $w$ is well separated from $v$. It follows that the interaction set $I(v)$ is

$$\begin{aligned} I(v) \quad = \quad & \{w \mid well\text{-}sep(v, w),\ [w \in P(parent(v)) \\ & \vee \{\neg well\text{-}sep(v, parent(w)), \\ & \exists u \in P(parent(v)), w \text{ is a descendant of } u, \\ & length(S(v)) < length(S(parent(w)))\}]\} \end{aligned}$$

The amount of work necessary to compute the partial local expansion at a node need not be constant and can be as high as $O(n)$, but the total work to compute all partial expansions is still $O(n)$ [2]. In this paper, we present an efficient parallelization of this distribution-independent sequential framework.

# 4 Parallel Fast Multipole Method

Our parallel FMM algorithm consists of four steps in each iteration: 1) Constructing the compressed octree for the particles, 2) Calculating multipole expansions, 3) Calculating partial local expansions, 4) Calculating local expansions. The local expansions at the leaf nodes are used to compute the force acting on each particle. The particle positions are updated and this completes one iteration of the N-body problem.

The data decomposition used by our algorithm is extremely simple: Initially, the $n$ particles are distributed such that each processor has approximately the same number of particles. Throughout the algorithm, the compressed octree data structure is stored in an array according to its postorder traversal. This array is uniformly distributed across the processors.

## 4.1 Compressed Octree Construction

Our algorithm for constructing compressed octrees is based on Bern *et al.*'s algorithm [3] for constructing quadtrees on the PRAM model. We also use the following result by Clarkson [5]. Note that the procedure uses floor, logarithm and bitwise exclusive-or operations.

**Lemma 1** *Let $R$ be the product of $d$ intervals $I_1 \times I_2 \times \ldots \times I_d$, i.e., $R$ is a hyperrectangular region in $d$ dimensional space. The smallest cell containing $R$ can be found in $O(d)$ time.*

Let $f$ be a bijective function that maps the 8 immediate subcells of a cell to the set $\{1, 2, \ldots, 8\}$. We extend this ordering to an arbitrary collection of cells. From the recursive decomposition of the domain into cells it is easy to see that, for any pair of cells, either the two cells are disjoint or one of them is completely contained in the other cell. Two cells are considered 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 way $f$ orders the corresponding immediate subcells. Using Lemma 1, we can order two cells in constant time.

Nodes in the tree can be ordered according to their small cells. Ordering of all nodes in the compressed octree corresponds to its postorder traversal. 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 least common ancestor of $v_1$ and $v_2$. In drawing a compressed octree, we make use of this ordering. Thus, ordering of the leaf nodes of the compressed octree results in the left to right ordering of the leaves according to our drawing protocol.

To construct the compressed octree, first sort the particles in parallel based on their positions and using the ordering described above. This orders the leaf

nodes of the compressed octree. The next step is to construct all the internal nodes. By constructing an internal node, we mean generating the small cell of the internal node. If we generate the least 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. The internal nodes are then mixed with the leaf nodes and they are sorted together in parallel, again using the ordering defined above. This results in the postorder traversal of the tree. Duplicates are brought together due to sorting, and can be easily eliminated. The array of nodes in this order is referred to as $T$, for the remainder of this paper.

It remains to assign pointers to complete the construction of the tree. For each node, we wish to determine its parent cell, search for and locate it and set up the pointers. The details of the process are as follows: For each node, we can determine its parent cell by finding the smallest cell containing the small cells at the node and the its right neighbor in $T$. Such a generated parent cell is called a *virtual parent cell*. We generate the virtual parent cells of all nodes in parallel. To this, we add the copies of cells corresponding to all internal nodes (actual parents), mix them together and sort them in parallel to create an array $T'$. In $T'$, the virtual parent cells and the corresponding actual parent cells come together. Each virtual parent cell brings with it information on the index of the child node that generated it. Similarly, each actual parent cell contains information on the index of the parent node. The child information from the virtual parent cell should be communicated to the actual parent. The parent information from the actual parent cell should be communicated to the child. Note that the total amount of incoming and outgoing information at each processor is bounded by $O\left(\frac{n}{p}\right)$. The communication is done by invoking the transportation primitive.

The parallel compressed octree construction algorithm uses 3 parallel sort operations and 1 transportation primitive. The running time of the algorithm is $O\left(\frac{n \log n}{p} + \tau p + \mu\left(\frac{n}{p} + p \log^2 p\right)\right)$ for $n > p^2 \log p$.

### 4.2 Multipole Expansion Calculation

The multipole expansion calculation is similar to summing up all the leaves in the subtree of each node where the leaves store numbers. This is known as upward tree accumulation. Parallel algorithms for this problem have been studied on the PRAM model (eg. [7, 9]). Our interest is to solve this problem efficiently, taking the distributed nature of memory into account.

A trivial parallelization of the sequential algorithm is as follows: Calculate the multipole expansions concurrently for the leaf nodes. Incorporate the multipole expansions at the leaf nodes to their parents in parallel and remove the leaf nodes. Repeat the process on the residual tree until it reduces to a single node. Because the height of the compressed octree can be $\Omega(n)$, the number of iterations is bounded only with $O(n)$.

In a tree where each internal node has at least two children, removing the leaf nodes in parallel removes at least half the nodes in the tree. Unfortunately, removal of the leaf nodes may create chains in the tree. If we can compress all the chains before starting the next iteration, at least half of the nodes in the residual tree are leaf nodes which leads to an algorithm that runs in $O(\log n)$ iterations.

When a node becomes a leaf node in the residual tree, its complete multipole expansion is known. We call a chain $v_1, v_2, \ldots, v_l$ a *leaf chain*, if $v_l$ is a leaf, and an *internal chain*, otherwise. For a leaf chain, the multipole expansion at $v_i$ is obtained by summing the stored multipole expansions in $v_i, v_{i+1}, \ldots v_l$. This is the familiar prefix sum problem and a leaf chain can be compressed in parallel using parallel prefix. This strategy does not work for an internal chain because $v_i$ needs the complete multipole expansion at $v_l$, which is not available for an internal chain. In Lemma 2, we prove that compressing only the leaf chains is sufficient to yield an $O(\log n)$ iteration algorithm.

**Lemma 2** *Compressing only leaf chains, multipole expansions can be calculated in $O(\log n)$ iterations.*

**Proof:** Consider a modified algorithm in which, at each iteration, not only the chains ending with leaf nodes but all the chains are compressed. Of course, this algorithm may not compute the correct multipole expansions for all the nodes. However, it guarantees that at least one half of the nodes are removed from the tree at each iteration. Thus, the number of iterations is bounded by $O(\log n)$.

To prove that $O(\log n)$ iterations are sufficient even when we compress only leaf chains, we show that if a node $v$ is removed as a leaf during the $k^{th}$ iteration by the modified algorithm, it is also removed by the original algorithm during the $k^{th}$ iteration as a leaf. The proof is by induction on the iteration number. Obviously, in the first iteration the nodes removed as leaves are the same. Assume this holds for $k-1$ iterations.

Let a node $v$ be removed as a leaf by the modified algorithm during step 1 of the $k^{th}$ iteration. Then, $v$ has never been on a chain and has at least two descendants $(u_1, ..., u_j,\ j > 1)$ removed as leaves during the $(k-1)^{th}$ iteration. Note that, there could be some

chains removed between $v$ and $u_i$ during the $(k-1)^{th}$ iteration. Clearly, $v$ could not be removed at a preceding iteration and $u_1, ..., u_j$ are removed as leaves at $(k-1)^{th}$ iteration by the original algorithm. For any $i$, consider the descendants of $v$ that are also ancestors of $u_i$ in the initial compressed octree. Each of these is removed by the modified algorithm before $k^{th}$ iteration as part of a chain. Even though none of these nodes is removed by the original algorithm before $(k-1)^{th}$ iteration, they all form a chain and are removed during phase 2 of iteration $k-1$. Thus, $v$ is a leaf at the beginning of the $k^{th}$ iteration in the original algorithm and is removed. This completes the proof. ■

As the execution of the algorithm progresses, some nodes are marked as deleted and they are never considered in subsequent iterations. Each iteration of the algorithm consists of two steps: In step 1, the multipole expansions at the leaf nodes are added to their parents and the leaf nodes are removed in parallel. The communication from leaves to the processors containing their parents can be effected using one transportation primitive. Then, the computation is performed by the processors containing their parents.

In step 2, each leaf chain is compressed using a parallel prefix operation. The entries in $T$ that are not deleted correspond to the postorder traversal of the residual tree. In this order, a chain of nodes occupies consecutive positions in $T$ (considering only entries that are not marked as deleted). We use a variant of segmented parallel prefix to collapse the chains together in one operation. We mark the beginning of each leaf chain in $T$ with a *start* marker. Intermediate nodes on any chain are not marked. The final node of any chain is marked with an *end* marker. Nodes that do not fall under any of the previous categories are also marked with an end marker. The identification of different categories is done as follows: If the next entry for a node in the residual tree has only one child, it must be the parent of the node. Consider a node $v$. If $v$ is a leaf and its next entry is a node with a single child, it gets a start marker. If $v$ is not a leaf and its next entry is a node with a single child, it is not marked. Otherwise, it gets an end marker. A segmented parallel prefix should be performed between matching start and end markers.

In each iteration of the algorithm, we perform $O\left(\frac{n}{p}\right)$ work on each processor for the purpose of identifying the leaves, setting up markers etc. We also use one transportation primitive and one segmented parallel prefix per iteration. Thus, the overall running time of the multipole expansion calculation is $O\left(\frac{n}{p}\log n + \tau p \log n + \mu\frac{n}{p}\log n\right)$ for $n > p\log p$.

### 4.3 Partial Local Expansion Calculation

The partial local expansion at a node is obtained by adding the multipole expansions of all nodes in its interaction set. Sequentially, the interaction sets are computed in a systematic manner during a top-down traversal. To compute interaction sets in parallel, we need a mechanism to independently generate the interaction sets. A further complication arises due to the fact that while the sum of the sizes of the interaction sets of all nodes in the tree is $O(n)$, we can only place a bound of $O(n)$ on the size of a single interaction set. Therefore, simply computing each interaction set independently is not efficient.

Instead of computing the interaction set of each node $v$, we can reverse the perspective and try to compute the set of nodes which have $v$ in their interaction sets. Then we may try to add the multipole expansion of $v$ to all these nodes. This strategy would be beneficial if the size of each such set is a constant. Unfortunately, this cannot be guaranteed and the size of such a set is bounded only by $O(n)$.

We will show that by employing a combination of the two strategies, we can ensure that the set sizes are constant. For a node $v$, each node in $I(v)$ is either in $P(parent(v))$ or a descendant of a node in $P(parent(v))$. We divide the interactions according to these two cases and define two additional sets, the *forward interaction set* $I_f(v)$ and the *backward interaction set* $I_b(v)$ as follows:

$$I_f(v) = \{w \mid \textit{well-sep}(v,w),\ w \in P(parent(v))\}$$

$$\begin{aligned} I_b(v) = \{&w \mid \textit{well-sep}(w,v), \neg\textit{well-sep}(w, parent(v)),\\ &\exists u \in P(parent(w)), v \text{ is a descendant of } u,\\ &length(S(w)) < length(S(parent(v)))\} \end{aligned}$$

Note that the interaction set for a node $v$ satisfies $I(v) = I_f(v) \cup \{w \mid v \in I_b(w)\}$. In the following, we seek to establish that $|I_f(v)| = O(1)$ and $|I_b(v)| = O(1)$ for any node $v$ and show how to compute these sets in parallel for all nodes in the tree.

For a cell $C$, $P^{=}(C)$ can be formed with only the knowledge of $C$. This is because $P^{=}(C)$ contains cells that are of the same size as $C$ but that are close enough that their multipole expansions do not converge in $C$. These cells are all contained in a constant number of layers around cell $C$. Furthermore, $|P^{=}(C)|$ is constant. The following two lemmas assert that for a node $v$, if we search the cells in $P^{=}(S(parent(v)))$ in the compressed octree we can find the nodes in $I_f(v)$ and if we search the immediate subcells of $P^{=}(S(parent(v)))$ we can find the nodes in $I_b(v)$.

**Lemma 3** *Let $v$ and $w$ be two nodes in a compressed octree. Then $w \in I_f(v)$ if and only if well-sep$(v,w)$, $\neg$well-sep$(parent(v),w)$, $\exists C$ such that $S(w) \subseteq C \subseteq L(w)$ and $C \in P^{=}(S(parent(v)))$.*

**Proof:** Let $w \in I_f(v)$. By definition, *well-sep*$(v,w)$ and $w \in P(parent(v))$. By the definition of proximity set, $\neg$*well-sep*$(parent(v),w)$ and $length(S(w)) \leq length(S(parent(v))) \leq length(L(w))$. Thus, $\exists C$ such that $length(S(parent(v))) = length(C)$ and $S(w) \subseteq C \subseteq L(w)$. Since $\neg$*well-sep*$(parent(v),w)$ and $S(w) \subseteq C$, $C$ and $S(parent(v))$ are not well-separated. Then, $C \in P^{=}(S(parent(v)))$.

Now, let *well-sep*$(v,w)$, $\neg$*well-sep*$(parent(v),w)$ and $C$ be such that $S(w) \subseteq C \subseteq L(w)$ and $C \in P^{=}(S(parent(v)))$. Then, $length(S(w)) \leq length(S(parent(v))) \leq length(L(w))$ and by definition, $w \in P(parent(v))$. Thus, by the definition of $I_f(v)$, $w \in I_f(v)$. ■

**Corollary 4** *$|I_f(v)|$ is bounded by a constant.*

**Proof:** Follows from Lemma 3 because the size of the proximity set of a cell is bounded by a constant. ■

**Lemma 5** *Let $v$ and $w$ be two nodes in a compressed octree. Then, $w \in I_b(v)$ if and only if well-sep$(w,v)$, $\neg$well-sep$(w,parent(v))$, $\exists C$ such that $S(w) \subseteq C \subseteq L(w)$ and $C$ is an immediate subcell of a cell in $P^{=}(S(parent(v)))$.*

**Proof:** Let $w \in I_b(v)$. Then, *well-sep*$(w,v)$ and $\neg$*well-sep*$(w,parent(v))$ by definition. Since $length(S(w)) < length(S(parent(v)))$, $\exists C'$ such that $S(w) \subset C'$ and $length(S(w)) < length(C') = length(S(parent(v)))$. Furthermore, $\exists u \in P(parent(w))$ such that $u$ is an ancestor of $v$ so, $length(S(parent(v))) \leq length(S(u)) \leq length(S(parent(w)))$. Therefore, an immediate subcell $C$ of $C'$ satisfies $S(w) \subseteq C \subseteq L(w)$. Because $\neg$*well-sep*$(w,parent(v))$, $C'$ and $S(parent(v))$ are not well-separated, $C' \in P^{=}(S(parent(v)))$.

On the other hand, let *well-sep*$(w,v)$ and $\neg$*well-sep*$(w,parent(v))$ and let $C$ be such that it is an immediate subcell of a cell $C' \in P^{=}(S(parent(v)))$ and $S(w) \subseteq C \subseteq L(w)$. Then, $length(S(w)) < length(S(parent(v)))$. Since $C \subseteq L(w)$, $C' \subseteq S(parent(w))$ and $\neg$*well-sep*$(parent(w),parent(v))$. Then, there is an ancestor $u$ of $v$ such that $length(S(u)) \leq length(S(parent(w))) \leq length(L(u))$. Note that $\neg$*well-sep*$(parent(w),u)$ because both $\neg$*well-sep*$(parent(w),parent(v))$ and $u$ is either $parent(v)$ or an ancestor of $parent(v)$. So, by definition, $u \in P(parent(w))$. It follows from the definition of $I_b(v)$ that $w \in I_b(v)$. ■

**Corollary 6** *$|I_b(v)|$ is bounded by a constant.*

**Proof:** Follows from Lemma 5 because the number of immediate subcells of a cell and the size of the proximity set of a cell are bounded by constants. ■

We use Lemma 3 to identify the forward interaction sets. For each node $v$, compute the cells in $P^{=}(S(parent(v)))$. For each cell $C \in P^{=}(S(parent(v)))$, search for the node whose small cell is contained in $C$ and whose large cell is at least as large as $C$. Retrieve the nodes and their multipole expansions. The multipole expansion of such a retrieved node $w$ is added provided $w$ is well-separated from $v$ but not well-separated from its parent.

The searches are performed in the following way: For each node $v$ in $T$, put $S(v)$ and the cells in $P^{=}(S(parent(v)))$ together into an array $T'$ and sort $T'$ in parallel. Along with each $S(v)$, carry its multipole expansion also. Sorting is done so that each cell having a multipole expansion comes before all the cells searching for it that are grouped together. Now, the multipole expansions can be broadcast using segmented parallel prefix to the cells searching for such information. The cells having the multipole expansions show the separation of the segments. Send the multipole expansion at each search cell to its originating node using the transportation primitive. Each entry of $T'$ generates only constant sized information. Similarly, each processor receives $O(1)$ amount of information for each node in array $T$. Thus, the total amount of outgoing and incoming communication at any processor is bounded by $O\left(\frac{n}{p}\right)$ and this leads to balanced communication. After the multipole expansions are received, they are incorporated if found appropriate according to Lemma 3.

An almost identical strategy works for the computation of the backward interactions sets. We highlight only the deviations: Use Lemma 5 and for each node $v$, generate the immediate subcells of $P^{=}(S(parent(v)))$. After sorting, the array $T'$ contains segments as before. Within each segment, the first entry is a cell from $T$ and rest of the entries are identical cells whose multipole expansions should be added to the first entry. This addition is accomplished in parallel for all segments by reduction using a segmented parallel prefix.

The partial local expansion calculation uses 2 parallel sort, 2 segmented parallel prefix and 2 transportation primitive operations. Thus, for $n > p^2 \log p$, the partial local expansion calculation requires $O\left(\frac{n \log n}{p} + \tau p + \mu\left(\frac{n}{p} + p \log^2 p\right)\right)$ time.

### 4.4 Local Expansion Calculation

To perform the top-down traversal required for local expansion calculation, we reverse the bottom-up traversal algorithm described in Section 4.2. During the bottom-up traversal, leaf chains are collapsed into single nodes. To reverse the procedure, single nodes must be expanded into leaf chains. It is difficult to predict when a node should be expanded and what should be the length of the chain if a node should indeed be expanded. We achieve this by collecting information during the execution of the bottom-up traversal so that we can exactly reverse it. This ensures that the run-time is identical to the bottom-up traversal procedure. We compute local expansions while activating the deleted nodes in the reverse order. We repeat two steps, which are actually the reverse of the bottom-up traversal.

The total run-time of our FMM algorithm is $O\left(\frac{n \log n}{p} + \tau p \log n + \mu\left(\frac{n \log n}{p} + p \log^2 p\right)\right)$. From an inspection of the run-times of the primitives described in Section 2, it is evident that the run-time is dominated by the sort operations for $n$ sufficiently larger than $p$. Thus, run-time of our parallel algorithm is bounded by the efficiency of parallel sorting which is an inevitable factor for any FMM algorithm as sorting can be reduced to compressed octree construction.

Several experimental studies have shown that the time spent in force calculation dominates the total running time, despite its lower order complexity ($O(n)$ vs. $O(n \log n)$ sequentially). Here, force calculation includes computing multipole expansions, adding multipole expansions to multipole expansions, shifting multipole expansions to local expansions, adding local expansions to local expansions and evaluating local expansions. It is important to design parallel algorithms that have optimal speedup for force calculation part of the algorithm. While the total running time of our parallel algorithm is optimal $O\left(\frac{n \log n}{p}\right)$, the same complexity applies to force calculation as well. We have designed a slightly more complex algorithm that does give optimal $O\left(\frac{n}{p}\right)$ running time for force calculation. It is not discussed here due to space limitations.

## 5 Conclusions and Open Problems

Our main contribution in this paper is the designing of the first parallel algorithm for computing the Fast Multipole Method (FMM) that is provably efficient, distribution-independent and uses statically determined mappings of computational and communication load to processors. We show that the problem can be solved in such a fashion on a realistic model of parallel computers, taking into detailed account the physically distributed nature of memory. We show that the FMM can be computed in parallel using such well-understood and simple primitives as parallel prefix, all-to-all communication and sorting.

Several interesting questions remain unanswered at this point: How does an implementation of the algorithm presented compare with the best existing implementations that employ dynamic load distribution? Given that this is the first algorithm to use a static decomposition to satisfactorily solve the problem, can we expect to find even simpler algorithms without the need for dynamic load distribution? In our tree construction algorithm, we assume that the logarithm and floor operations can be performed in constant time. Is it possible to achieve similar results using only the standard algebraic model of computation?

## References


[1] S. Aluru, Greengard's N-body algorithm is not order N. *SIAM Journal on Scientific Computing, 17* (1996) 773-776.

[2] S. Aluru, G.M. Prabhu, J. Gustafson and F. Sevilgen, Distribution-Independent Hierarchical Algorithm for the N-body Problem, *Journal of Supercomputing, 12* (1998) 303-323.

[3] M. Bern, D. Eppstein and S.H. Teng, Parallel Construction of Quadtrees and Quality Triangulations, *Workshop on Algorithms and Data Structures* (1993) 188-199.

[4] P.B. Callahan and S.R. Kosaraju, A decomposition of multidimensional point sets with applications to $k$-nearest neighbors and $n$-body potential fields, *J. of the ACM, 42(1)* (1995) 67-90.

[5] K. L. Clarkson, Fast Algorithms for the All Nearest Neighbors Problem, *Proc. Foundations of Computer Science* (1983) 226-232.

[6] L. Greengard, *The rapid evaluation of potential fields in particle systems*, MIT Press, Cambridge, MA, 1988.

[7] S.R. Kosaraju and A.L. Delcher, Optimal parallel evaluation of tree-structured computations by raking, *Proc. 3rd Aegean Workshop on Computing* (1988) 101-110.

[8] V. Kumar, A. Grama, A. Gupta and G. Karypis, *Introduction to Parallel Computing*, The Benjamin/Cummings Publishing Co., 1994.

[9] G.L. Miller and J.H. Reif, Parallel tree contraction and its application, *Proc. IEEE Symposium on the Foundation of Computer Science* (1985) 473-489.

[10] S. Ranka, R.V. Shankar and K.A. Alsabti, Many-to-many communication with bounded traffic, *Proc. Frontiers of Massively Parallel Computation* (1995), 20-27.

[11] V. Rokhlin, Rapid solution of integral equations of classical potential theory, *Journal of Computational Physics, 60* (1985) 187-207.

[12] S.H. Teng, Provably good partitioning and load balancing algorithms for parallel adaptive N-body simulation, *SIAM Journal on Scientific Computing, 19(3)* (1998) 635-565.