N-body Simulation I: Fast Algorithms for Potential Field Evaluation and Trummer's Problem*

John H. Reif†
Department of Computer Science
Duke University
Box 90129
Durham, NC 27708–0129

reif@cs.duke.edu

Stephen R. Tate‡
Department of Computer Science
University of North Texas
P.O. Box 13886
Denton, TX 76203–6886

srt@cs.unt.edu

Abstract

In this paper, we describe a new approximation algorithm for the n-body problem. The algorithm is a non-trivial modification of the fast multipole method that works in both two and three dimensions. Due to the equivalence between the two-dimensional n-body problem and Trummer's problem, our algorithm also gives the fastest known approximation algorithm for Trummer's problem.

Let A be the sum of the absolute values of the particle charges in the n-body problem under consideration (or the sum of the masses if the simulation is gravitational). To approximate the particle potentials with error bound ϵ, we let $p = \lceil \log(A/\epsilon) \rceil$ and give complexity bounds in terms of p. Note that, under reasonable assumptions on the particle charges, if we desire the output to be accurate to b bits, then $p = \Theta(b)$. In two dimensions, our algorithm runs in time $O(n \log^2 p)$, which is a substantial improvement over the previous best algorithm which requires $\Theta(np \log p)$ time. We also apply our new algorithm to the three dimensional problem and get a new algorithm that has time complexity $O(np^2)$, an improvement over the best previously-known three-dimensional algorithm which requires $\Theta(np^2 \log p)$ time. Our algorithms do not make any assumptions about the input distribution, and are true worst-case bounds.

Keywords: N-body problem, numerical approximation, algebraic algorithms, parallel algorithms

AMS subject classifications: 68Q25, 65F30, 70F10

Abbreviated title: N-body Simulation I: Potential Field Evaluation

---

*A preliminary version of these results was published in *FOCS 92* [32].

†Supported by NSF Grant NSF-IRI-91-00681, Rome Labs Contracts F30602-94-C-0037, ARPA/SISTO contracts N00014-91-J-1985, and N00014-92-C-0182 under subcontract KI-92-01-0182.

‡Supported in part by NSF Grant CCR-9409945.

1 Introduction

This paper presents new algorithms for the n-body problem, one of the most prevalent and important problems amenable to computer solution today. The algorithms that we present give approximate (to any precision desired) solutions to the n-body problem, and have asymptotic complexity measures better than any previous algorithm. In this introduction, we first give a definition of the n-body problem, then discuss some related problems and previous work, and finally summarize our results.

In the n-body problem, you are given n charged particles at positions $\mathbf{p}_1, \cdots, \mathbf{p}_n$, with charges $q_1, \cdots, q_n$, respectively[1]. These particles have pairwise force interactions, which are determined by equations that depend on the dimensionality of the space. In particular, in three dimensions the force obeys the widely known inverse-square force law, and in two dimensions particle i induces a force on particle j given by

$$E_{\mathbf{p}_i}(\mathbf{p}_j) = k q_i q_j \frac{\mathbf{p}_i - \mathbf{p}_j}{||\mathbf{p}_i - \mathbf{p}_j||^2}, \tag{1}$$

where k is a constant that determines the units of measurement of the force. For the remainder of this paper, we will use $k = 1$ — if the forces are desired in some other units, then the final results may simply be scaled at the end of the computation. The goal of the n-body problem is to determine the force exerted on each particle by all other particles. Directly using equation (1) to calculate these forces would take $\Theta(n^2)$ time, but since the input consists of $O(n)$ values, it is sensible to try to find better algorithms. This is the problem that we examine in the current paper — in a companion paper, we examine the related problem of simulating the motion of particles over time, giving both lower and upper bounds on the complexity required to do so [37].

1.1 Trummer's problem

Trummer's problem is the problem of multiplying a particular structured matrix T times an arbitrary vector $\vec{y}$. The matrix T is completely specified by n complex values $c_1, \cdots, c_n$ and the formula

$$T_{i,j} = \begin{cases} \frac{1}{c_i - c_j} & \text{if } i \neq j, \\ 0 & \text{if } i = j. \end{cases}$$

Since the matrix can be specified with n values, it is reasonable to try to design $o(n^2)$ algorithms for this problem. While this problem, originated by Manfred Trummer and publicized by Gene Golub as a "challenge" in *SIGACT News*, is mentioned in neither AHU nor the Knuth volumes, we will argue that Trummer's problem is of fundamental interest, with a wide range of applications.

[1] In this paper, boldface variables represent vectors and standard math italic variables represent scalars.

The relationship between Trummer's problem and the n-body problem in two dimensions is clear, once some basic facts from complex analysis are recalled. The following discussion is a standard encoding of two dimensional n-body problems in the complex plane, which can also be found, for example, in [12]. In particular, the force equation (1) is induced by a potential field ϕ, which at point $\mathbf{p}_m$ is

$$\phi(\mathbf{p}_m) = \sum_{\substack{i=1 \\ i \neq m}}^{n} -q_i \ln \|\mathbf{p}_i - \mathbf{p}_m\|. \tag{2}$$

When the $\mathbf{p}_j$'s are two dimensional, this equation is harmonic except at the $\mathbf{p}_j$ positions, so if each position $\mathbf{p}_j = (x_j, y_j)$ is represented by a point in the complex plane as $z_j = x_j + iy_j$, then there exists an analytic function over the complex numbers such that the real part of the function at $z = x + iy$ is exactly the value of ϕ at position (x, y). In particular, in this case we have (with slight abuse of notation by using the same symbol ϕ to now denote a function over the complex numbers)

$$\phi(z_m) = \sum_{\substack{i=1 \\ i \neq m}}^{n} q_i \mathrm{Re}\left[-\ln(z_i - z_m)\right].$$

Furthermore, since the force exerted on a particle is the derivative of the potential field, and the function is differentiable (so we can apply the Cauchy-Riemann equations), it is not hard to see that the force exerted on particle j by all other particles is

$$E(\mathbf{p}_j) = q_j \cdot \left(\mathrm{Re}\left[\phi'(z_m)\right], -\mathrm{Im}\left[\phi'(z_m)\right]\right). \tag{3}$$

In other words, all we need to do is calculate $\phi'(z_m)$ at each of the points determined by the particle positions. This value is given by

$$\phi'(z_m) = \sum_{\substack{i=1 \\ i \neq m}}^{n} \frac{q_i}{z_i - z_m},$$

which is now obviously just an instance of Trummer's problem. We note here that by replacing the particle charges by particle masses, we can compute gravitational forces in this same manner. The preceding discussion shows that Trummer's problem and the two-dimensional n-body problem are equivalent, and so our improved algorithms for two-dimensional n-body problems also give improved algorithms for Trummer's problem.

1.1.1 Evaluation of the Riemann Zeta function

A key application of Trummer's problem is in the numerical evaluation of the Riemann zeta function. Improved algorithms for evaluating the zeta function allow the very important Riemann hypothesis to be verified for large sets of zeros. Odlyzko and Schönhage have shown that multiple evaluations of the zeta function can be done very efficiently, if some time is allowed for preprocessing [30]. The dominating computation in the preprocessing is the evaluation of the function

$$\sum_{k=1}^{n} \frac{a_k}{z - b_k}, \tag{4}$$

for values $a_1, a_2, \cdots, a_n$, and $b_1, b_2, \cdots, b_n$, at z equal to the n nth roots of unity. The algorithm of [30] takes time $O(np^c)$ for p bit accuracy, for some moderate constant c, which is a considerable improvement (see [29]) over the best previously known n^2 naive algorithm for this problem for the ranges of n of interest (Odlyzko's computations ranged over 1,000,000). The summation shown in (4) can easily be reduced to an instance of Trummer's problem of size $2n$, which can then be evaluated by efficient algorithms for Trummer's problem. Furthermore, the algorithm in this paper can easily be adapted to directly evaluate (4) in time $O(n \log^2 p)$ *without* the factor of 2 overhead required in the explicit reduction to Trummer's problem.

1.1.2 Other Applications of Trummer's Problem

The further applications of Trummer's problem are quite extensive. One is the solution of 2D PDEs by Potential Methods, as pioneered by Rokhlin. An early paper of Rokhlin [39] showed that the solution of Laplace's equation in 2D with irregular boundary conditions (a difficult, fundamental problem in numerical analysis) reduced to Trummer's problem.

Another major application is in fluid mechanics; particularly the vortex method of Anderson and Chorin [2, 8], which is used in 2D turbulent flow simulations and plasma simulations. In particular Greengard [11] showed that an important case of 2D fluid flow through a channel reduces to a Poisson problem and hence to variants of Trummer's problem.

1.2 Motivation and History

The goal of molecular dynamics studies is to simulate a set of n charged particles, where the particles interact under a self-induced electrostatic or gravitational potential field. Generally, these simulations are done by time stepping (see, for example, [3, 15, 17, 19, 28, 37] for details). These simulations are one of the heaviest users of supercomputer cycles, and are widely used by astronomers, chemists, and biochemists, and to a lesser degree physicists[2]. For example, a study by the Microelectronics Center of North Carolina recently showed that over 30 percent of all compute time on their CRAY-YMP was used for n-body simulation by molecular chemists. Most of the widely available commercial molecular simulation packages used by chemists for molecular dynamics have the n-body potential problem as the predominant subroutine.

[2]Some physicists prefer other simulation methods based on energy minimization.

Rather than attempting to compute the exact potential induced by a set of particles, people have considered various approximation algorithms, which can produce output with accuracy up to (or exceeding) the machine precision. All current approximation algorithms require that the simulation space be partitioned using a space decomposition tree. An algorithm that uses an already constructed decomposition tree is called a *static* algorithm, and an algorithm that must build the tree itself (or modify a previous tree) is called an *adaptive* algorithm.

1.3 Previous Work and Our Algorithm

Due to the equivalence between the two-dimensional n-body problem and Trummer's problem, previous work comes from both these areas. Two years after Golub published Trummer's problem as a challenge, a solution was published by Gerasoulis [9], who gave an $O(n \log^2 n)$ time algebraic algorithm that solved Trummer's problem. Unfortunately, Gerasoulis' algorithm turned out to be numerically unstable for large n. Looking at the n-body problem at about the same time, Greengard and Rokhlin gave provably good approximation algorithms (known as the fast multipole algorithm) for the n-body problem in both two and three dimensions [12, 13, 14]. In order to understand the complexity of their algorithm, we define $A = \sum_{i=1}^{n} |q_i|$ to be the sum of the absolute particle charges, and if we desire the output to meet error bound ϵ we define $p = \lceil \log(A/\epsilon) \rceil$. We refer to a problem with these input/output constraints as an "accuracy p" version of the n-body problem. In most previous work, it was assumed that there was some number of bits b that bounded both input and output precision, and that furthermore you could bound A by 2^b. Under these reasonable assumptions, $p \leq \lceil \log(2^b/2^{-b}) \rceil = \Theta(b)$, and so the concepts of "accuracy p" and "output accurate to b bits" were used interchangeably. In this paper, we refer to "accuracy p", but keep this concept distinct from the pure number of bits of accuracy of the output, since they are really two separate things. Throughout this paper we assume that $p \leq n$. In the unlikely event that $p > n$, simply using the exact algorithm of Gerasoulis [9] gives asymptotic complexity at least as good as the approximation algorithms of this paper.

For the two-dimensional problem with accuracy p, Greengard and Rokhlin's initial algorithm had running time $O(np^2)$, which was later improved to $O(np \log p)$ time [14]. The fast multipole algorithm attracted a lot of attention, and numerous studies and implementations have been made (see, for example, [5, 20, 23, 25, 24, 40]). The running time's dependence on p can be considerable, since n-body simulations for molecular dynamics typically require values of p of at least 16 (and often much more for high accuracy planetary simulations). One additional restriction of Greengard and Rokhlin's algorithm is that it requires certain assumptions about uniformity of the input distribution in order to meet these complexity bounds.

An early version of the work presented here improved the running time of the multipole algorithm to $O(n \log^2 p)$ by using methods similar to those presented in this paper, but using a much more complicated spatial decomposition [36]. A summary of these results was subsequently published [32], but since that time the algorithms (as presented here) have completely changed.

In one further important piece of previous work, Callahan and Kosaraju examined the problem of removing input assumptions from the multipole algorithm, and devised a framework for spatial decomposition that proved to be useful not only for the n-body problem, but also for many other interesting problems from computational geometry [6]. These decomposition methods did not improve the running time of the multipole algorithm, but did remove the assumptions about the input distribution that were required by Greengard and Rokhlin's algorithms. Our improved multipole algorithms now incorporate many of the ideas developed by Callahan and Kosaraju, which have simplified the algorithm and improved the presentation substantially over the previous version. In two dimensions, our multipole algorithms match our previous bound of $O(n \log^2 p)$ time, and require no assumptions about the input distribution (hence they are true worst-case time complexity bounds).

1.4 Improved Results for n-body potential Calculations

In practice, many divide-and-conquer algorithms are implemented so that a more naive algorithm, with possibly worse asymptotic running time but small constants, is used when the problem is reduced to a small enough size. This technique is common in implementing often-used algorithms, such as sorting, arithmetic on large numbers, etc. In some cases, this technique can be used to give algorithms with improved asymptotic running times (see for example [33, Problem 330]). In this paper, we apply this technique to the fast multipole algorithm, which involves some complex interaction between the various parts of the algorithm, and involves developing a new decomposition algorithm, as well as new algorithms for the "small size" subproblems that result. Several of these problems reduce to operations on structured matrices, and others can be reduced to the previous algebraic algorithm for Trummer's problem developed by Gerasoulis [9].

Our main positive result is an improved static algorithm that has running time $O(n \log^2 p)$ in two dimensions. Thus, when compared to the previous best algorithm, the time is improved by a factor of $\Omega(\frac{p}{\log p})$. Our static algorithm makes use of an improved notion of space decomposition, in which the decomposition tree has particles associated with internal nodes as well as leaves. In addition to improving the running time of n-body potential calculation, our static algorithm achieves this running time *without any assumptions on the input distribution.* Our techniques also extend to three dimensional problems, giving a new algorithm with a time bound of $O(np^2)$. We also parallelize both of these algorithms (using a variant of parallel tree contraction), giving a two-dimensional algorithm that runs in parallel time $O(\log n \log^2 p \log^* p)$ with $\frac{n}{\log n \log^* p}$ processors, and a three-dimensional algorithm that runs in parallel time $O(\log n \log p)$ with $O(np^2/(\log n \log p))$ processors.

Furthermore, we make our algorithms adaptive by giving an efficient algorithm for building the space decomposition tree. Our tree-building algorithm takes $O(n \log n)$ time in both two and three dimensions without any assumptions about the form or distribution of input particles, which was shown to be optimal for algebraic computation by Callahan and Kosaraju [6]. In addition, if input constraints are allowed, we give improved decomposition algorithms. In particular, if the particle coordinates are given using $O(\log n)$ bits, then we can perform the spatial decomposition in $O(n \log\log n)$ time; furthermore, if the particles are uniformly distributed then we can compute the spatial decomposition in $O(n)$ expected time. It should be noted that the space decomposition algorithm of Carrier, Greengard, and Rokhlin [7] actually takes $\Theta(n \log n)$ time, despite their claimed linear time bounds. They state the running time of their algorithm as $O(np)$, but have an implicit assumption that $p = \Theta(\log n)$. Using these same assumptions, our fully dynamic algorithm takes only $O(n \log\log n + n \log^2 p) = O(n \log^2 p)$ time, whereas the complete algorithm of Greengard and Rokhlin required $\Theta(np \log p)$ time.

2 Building the Decomposition Tree

In this section, we consider the problem of creating a decomposition tree for a set of particles, hierarchically subdividing the space containing the particles until each subdivision has a small number of particles. Our decomposition is based on the method due to Callahan and Kosaraju [6], with modifications to support our efficient potential field evaluation.

Any hierarchical space decomposition can be naturally represented as a tree, where each node represents a region of space (which we will refer to as a "box") and the children of a box represent the subdivisions of that region in the space decomposition. The leaves of such a decomposition tree represent boxes that are not subdivided.

The Callahan and Kosaraju decomposition [6] takes a space with n designated points (which we will refer to as particles), and produces a decomposition tree in which each leaf corresponds to a region containing exactly one particle. Of course, such spatial decompositions are not difficult to construct. The true contribution of Callahan and Kosaraju is that their decomposition has a linear size well-separated realization (this will be defined later, and is an essential property for multipole algorithms), and doesn't depend on any assumptions regarding the distribution of the input points or the precision of their representation, removing restrictions that existed for all previous multipole algorithms [12, 10].

Our decomposition will have the above properties (linear size well-separated realization and distribution independence), but will contain more than a single particle in each leaf of the decomposition tree. In particular, we use s (which may be a function of n) to represent our "goal size" for the number of particles in each leaf, and will guarantee that the average number of particles in each leaf is $\Theta(s)$. Since each internal node will have two children, this implies that the size of our decomposition tree is $\Theta(n/s)$. If the decomposition tree is viewed as a divide-and-conquer algorithm, this corresponds to stopping the divide-and-conquer at problems of size s (on average), and switching to an alternate algorithm. In later sections we will see how this is advantageous.

Since the Callahan and Kosaraju decomposition is a top-down decomposition, the first approach that suggests itself is to simply stop subdividing when you obtain a region with $\leq s$ particles. Unfortunately, a subdivision of a region with m particles can have anywhere from 1 to $m - 1$ particles, so this method can produce decomposition trees with $\Omega(n)$ nodes. We will next define terminology required in order to discuss decomposition trees, review the algorithm of Callahan and Kosaraju, describe our modifications, and finally prove some important properties concerning our decomposition.

Let R denote a rectangular region of d-dimensional space, and let $l_i(R)$ denote the length of this region along dimension i. We use $\max(R)$ to denote the dimension number of the longest side[3] of R, and use $l_{\max}(R)$ as short-hand for $l_{\max(R)}(R)$. For any region of space R, let $P(R)$ denote the set of particles in that region, and for any set of particles P, we use $R(P)$ to denote the smallest rectilinear region enclosing all the particles of P. We will often need to refer to the region $R(P(R))$, the minimal region containing all the particles covered by R, so give it the shorthand notation $\hat{R}$; notice that $\hat{R}$ is not necessarily the same as R, but will certainly be a subset of R.

The Callahan and Kosaraju decomposition works as follows: Given a region R containing more than one particle, the algorithm decomposes this region by splitting $\hat{R}$ in half along dimension $\max(\hat{R})$. This is repeated for each subregion until each region contains only a single particle. A *fair split* of any region R is one in which the distance from the splitting hyperplane to either of the parallel sides of R is at least $l_{\max}(\hat{R})/3$; since this distance is at least $l_{\max}(\hat{R})/2$ when the splits are chosen as described above, every subdivision in the Callahan/Kosaraju decomposition is a fair split. A decomposition tree in which all internal nodes represent fair splits is called a *fair split tree.*

Unfortunately, the notion of a fair split tree is too strict for us, since it can produce many regions containing only one particle; therefore, we introduce the following definition.

Definition 2.1 Given a decomposition tree, consider a region R which contains more than s particles and is split into two sub-regions R_1 and R_2 along dimension k. This split is an *internal fair split* if both of the following conditions are met:

[3] In cases of a tie, we can disambiguate the term "longest side" to refer to minimum dimension number among the maximum-length sides.

- The dimension split is least as large as the maximum side of $\hat{R}$. In other words, $l_k(R) \geq l_{\max}(\hat{R})$.
- For $i = 1, 2$ either $l_k(R_i) \geq l_{\max}(\hat{R})/2$, or R_{3-i} (the other side) contains $\leq s$ particles and hence is a leaf in the decomposition tree.

Notice that nodes that have two internal nodes as children must be fairly split, but nodes in which one child is a leaf may have that leaf region larger than allowed by a true fair split. An *internal fair split tree* is a tree in which all splits are internal fair splits.

The algorithm for creating an internal fair split tree is the same as the algorithm of Callahan and Kosaraju, except for an additional test during a split to determine if either subregion of an even geometric split has fewer than $\lfloor s/2 \rfloor$ particles. If so, then the splitting hyperplane is moved so that the light side has exactly $\lfloor s/2 \rfloor$ particles. Since that region will not be subdivided any further, it is a leaf and so the split is not required to be geometrically balanced.

Note that in doing the splits this way, we have insured that every leaf has at least $\lfloor s/2 \rfloor$ particles, and so there are $O(n/s)$ nodes in the decomposition tree. The complexity of this algorithm is summarized in the following theorem, and the reader may refer to Callahan and Kosaraju's paper [6] for more details about the algorithm.

Theorem 2.1 Given a set of n particles, in time $O(n \log(n/s))$ we can compute an internal fair split tree in which all leaves have at least $\lfloor s/2 \rfloor$ particles.

The next two sections consider special cases of the spatial decomposition problem, where either particle positions are specified with limited precision or particles are uniformly distributed. In Section 2.3 the important property of internal fair split trees that is shared by all three cases is examined.

2.1 Special Case: Limited Precision Input

In this section, we consider constructing a spatial decomposition if the input particle positions are given in fixed point and with $O(\log n)$ bits of accuracy. This was, in fact, one of the assumptions of Greengard and Rokhlin's algorithm, as they assumed that their quad-tree based decomposition would never have more than $O(\log n)$ levels.

This special case has a non-trivial solution which may be applied to other geometric problems such as closest pair problems, and the authors have described the general limited precision spatial decomposition in a separate paper [38]. The decomposition algorithm from that paper is summarized in the following Lemma (based on the Theorem 2.1 and Lemma 3.1 from [38]).

Lemma 2.1 Given n particles in d dimensions (d a constant), where the coordinates of each particle are given using $c \log n$ bits, we can construct a fair split tree for these particles in $O(n \log\log n)$ time.

The tree constructed by the algorithm in the cited paper is a simple decomposition tree: each leaf has only one particle, meaning that the tree has size $\Theta(n)$. In the remainder of this section we show how a simple post-processing step can convert that tree into the required $O(n/s)$-node internal fair split tree. While not every leaf will have at least $\lfloor s/2 \rfloor$ particles as in the last section, the *average* number of particles in each leaf will still be $\Theta(s)$, which is sufficient for guaranteeing that the decomposition tree has $O(n/s)$ nodes.

The first step after constructing a fair split tree is to do a tree traversal and collapse all maximal subtrees that contain at most s particles into a single node. We next modify internal nodes slightly so that in addition to the regions represented by its children, it can also contain "supplemental particles" that represent particles outside of its child regions. No node will ever contain more than s supplemental particles. While an efficient algorithm could actually bypass the supplemental particles, it is convenient to think of building a tree containing nodes with such supplemental particles and then transforming the tree back into a normal decomposition tree, which can be done in at most four subdivisions, as shown in figure 1. It is important to note that since the splits of the original tree *always* exactly divide a dimension in half, no child region can cross the half-way point in any dimension. This means that we can always insert these four additional splits in such a way that the side containing only supplemental particles is the larger side of the split, which is necessary for the split to be an internal fair split.

Since each node with supplemental particles is expanded into a constant number of regular decomposition nodes (with no supplemental particles), if the decomposition tree with supplemental particles has $O(n/s)$ nodes then the final decomposition tree will also have $O(n/s)$ nodes.

The transformation of the original fair split tree into a decomposition tree with supplemental particles and $O(n/s)$ nodes is based on a post-order traversal of the tree. When a node v is visited in this postorder traversal, if the node has one child that is an internal node and one that is a leaf, and if the number of supplemental particles in the internal node plus the number of particles in the leaf is at most s, then we can combine v with its two children to create a new internal node with the same children as the internal child of v and an expanded region that adds the particles from the leaf child to the supplemental particles covered by the node.

When this process completes it is obvious that no internal node contains more than s supplemental particles, but how many nodes can there be? The following lemma addresses exactly this question.

Lemma 2.2 The decomposition tree with supplemental particles constructed as just described has $O(n/s)$ nodes.

Proof: Consider any leaf in the tree. If the sibling of this node were another leaf, then the sum of

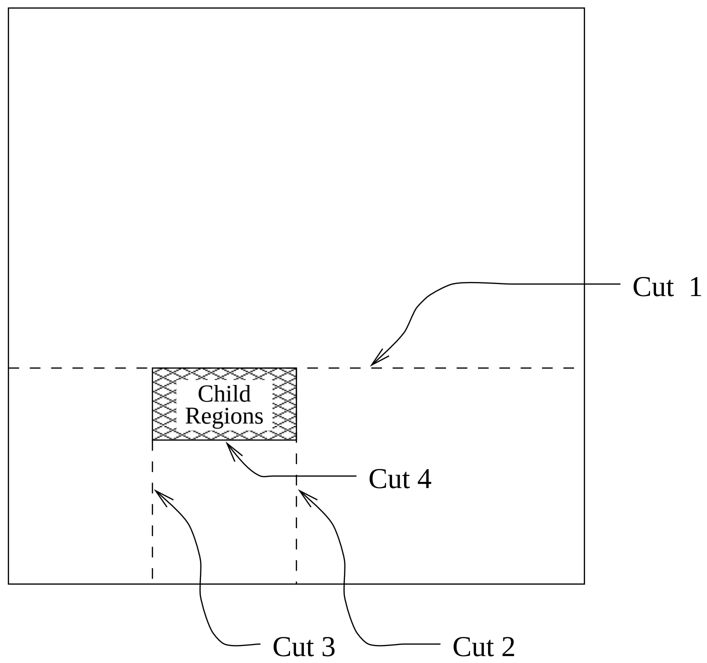

Figure 1: Turning a node with a "supplemental region" into a decomposition into five regions.

the number of particles in both leaves must be more than s; otherwise, the first step would have collapsed these two nodes and their parent into a single leaf node. On the other hand, if the sibling of the leaf under consideration is an internal node, then the sum of the number of particles in the leaf plus the number of supplemental particles in its sibling must be greater than s; otherwise, they would have been merged by the last step. In other words, even though every leaf may not contain s or even $s/2$ particles, when combined with either the particles or the supplemental particles in its sibling the total number of particles is over s. This means that there can be at most n/s such sibling pairs, and so the number of leaves in the tree is at most $2n/s$. Since each internal node has two children, the total number of nodes is $O(n/s)$. ∎

Recalling that this decomposition tree can be easily transformed into a standard decomposition tree with no supplemental particles, and that is in fact an internal fair split tree, this section can be summed up in the following lemma.

Lemma 2.3 Given n particles in d dimensions (d a constant), where the coordinates of each particle are expressed using $c \log n$ bits, we can construct an $O(n/s)$-node internal fair split tree for these particles in $O(n \log \log n)$ time.

2.2 Special Case: Uniform Particle Distribution

If the particles are uniformly distributed, we can do even better in the expected case by producing a decomposition in $O(n)$ expected time. The idea is straight-forward, and only described briefly here. First, divide the entire space into n equal size boxes (by dividing a side of length ℓ into equal size intervals of length $\ell/n^{1/d}$). Next, scan through all the input points putting each one into the appropriate bucket; this is done in constant time per particle, or $O(n)$ time total. After this step the expected number of particles in any bucket is constant, and we can run the decomposition algorithm from the beginning of this section on any bucket that has more than one particle. While the original decomposition algorithm took $O(n \log n)$ time, since we are using it on buckets that contain a constant expected number of particles, the expected time of running this algorithm for each bucket is constant. When this is completed we have a full internal fair split tree of size $O(n)$ that was constructed in $O(n)$ expected time. Given this, we perform the raking operation of the previous section in order to produce an $O(n/s)$ node decomposition tree that we can use in the rest of our algorithm. This is summarized below.

Lemma 2.4 Given n particles that are uniformly distributed in a d-cube, we can compute an $O(n/s)$-node internal fair split tree in $O(n)$ expected time.

2.3 Internal Fair Split Tree Properties

In this section, we prove an important property for internal fair split trees such as those described in the preceding three sections (for the general problem, for inputs with limited precision, and for uniformly distributed particles, respectively).

From this point on, we will use the notation DT to refer to such a decomposition tree (that is, an internal fair split tree with at most $O(n/s)$ nodes). We will refer to the nodes of DT and the regions that they represent interchangeably, and we use standard tree terminology — for example, if region R is decomposed into regions R_1 and R_2, then we say that R is the parent of R_1 and R_2, and use notation $\mathsf{p}(R_1)$ to denote the "parent of R_1" (so $\mathsf{p}(R_1) = R$ in this example). Furthermore, we say that any region of space R covers a set of nodes S if no leaves of DT other than descendants of elements of S intersect with R.

The next lemma describes one of the most important properties of internal fair split trees for our multipole algorithms.

Lemma 2.5 For any node R in an internal fair split tree DT, we can place a d-cube with side-length $\frac{1}{2} l_{\max}(\widehat{\mathsf{p}(R)})$ such that it covers R and at most d leaves in the decomposition tree.

Proof: Let R be an arbitrary node of DT and let $D = \{i | l_i(R) < \frac{1}{2} l_{\max}(\widehat{\mathsf{p}(R)})\}$. If D is empty, then we can select an appropriately sized d-cube such that it covers only R, the lemma is easily seen to be true; therefore, assume for the rest of this proof that D is non-empty.

Let i be any dimension in the set D, and look above R in the decomposition tree to find the last split in dimension i (if region R has never been split in dimension i, then there is only empty space adjoining region R in dimension i — in this case, the proof goes through with trivial modifications that are omitted here for simplicity). Let Q represent the node in DT immediately before this split. Since we split dimension i of Q, and since Q is an ancestor of $\mathsf{p}(R)$, we can derive

$$l_i(Q) \geq l_{\max}(\hat{Q}) \geq l_{\max(\widehat{\mathsf{p}(R)})}(\hat{Q}) \geq l_{\max}(\widehat{\mathsf{p}(R)}). \tag{5}$$

Since Q is a proper ancestor of R, exactly one of its children must be an ancestor of R. We call this child of Q by the name R_i, and we will call the other child of Q (which is not an ancestor of R) by the name Q_i. Since dimension i is in the set D, and Q represents the last time this region was subdivided along dimension i, inequality (5) implies that $l_i(R_i) = l_i(R) < \frac{1}{2} l_i(\hat{Q})$, so the split of region Q is not geometrically even. Since this is an internal fair split, Q_i must be a leaf of DT. Furthermore, since the split of region Q is the most recent split in dimension i, region Q_i must be adjacent to region R, and $l_i(R \cup Q_i) = l_i(Q) \geq l_{\max}(\widehat{\mathsf{p}(R)})$.

Since region R can be expanded in each short dimension by adjoining a leaf node Q_i, with the resulting area having length at least $\frac{1}{2} l_{\max}(\widehat{\mathsf{p}(R)})$ across R in all dimensions, clearly, we can place a d-cube with side-length $\frac{1}{2} l_{\max}(\widehat{\mathsf{p}(R)})$ such that it covers

$$P \cup \bigcup_{i \in D} Q_i.$$

Since each Q_i is a leaf, and $|D| \leq d$, the lemma follows. ∎

3 Creating a Realization

The spatial decomposition and the multipole algorithm are connected by a *realization*, which is essentially a set of pairs of nodes from the decomposition tree. For the reader unfamiliar with multipole algorithms, bear with this section and the motivation behind the various definitions and proofs will be clear in the next section, when our modified multipole algorithm is discussed.

As mentioned above, realizations deal with pairs of nodes from the decomposition tree, DT, so we first introduce some terminology and notation for dealing with such pairs. We assume that there is some arbitrary linear ordering on the nodes of DT (this could be an ordering by storage address, creation time, or any other method that gives a consistent ordering). If node P comes before node Q in this ordering, we use the notation $P \prec Q$.

We refer to a pair of nodes by the notation (P, Q), which is an *ordered pair* in which the ordering is selected to be consistent with the following conditions:

- If P and Q are both internal nodes, then either $l_{\max}(\hat{P}) > l_{\max}(\hat{Q})$, or $l_{\max}(\hat{P}) = l_{\max}(\hat{Q})$ and $P \prec Q$.
- If exactly one of P and Q is an internal node, it must be the first node in the pair (i.e., P).
- If both P and Q are leaves, then $P \prec Q$.

For any two nodes P and Q, the function $\mathsf{Pair}(P, Q)$ gives the properly ordered pair (either (P, Q) or (Q, P)) for these nodes.

The most important property of a pair of nodes for multipole algorithms is the notion of *well-separatedness*. In particular, for some constant $\alpha > 1$ called the *separation constant*, an ordered pair (P, Q) of nodes is called α-well-separated (or just plain "well-separated" when α is either understood or unimportant) if the appropriate condition from the following list is met:

- If P and Q are both internal nodes, then both $\hat{P}$ and $\hat{Q}$ can be enclosed in spheres of radius r which are separated by distance at least αr.
- If P is internal and Q is a leaf, then $\hat{P}$ can be enclosed in a sphere S of radius r, where $d(S, \hat{Q}) \geq \alpha r$.
- If P and Q are distinct leaves, then they are always well-separated.

The following notions are based on concepts from Callahan and Kosaraju [6], but have been extended to work with our more complex spatial decomposition. Let A and B be two nodes of DT. The *interaction product*, denoted $A \otimes B$, is the set of pairs of particles

$$A \otimes B = \{\{p, q\} |\ p \in P(A),\ \ q \in P(B),\ \text{ and } p \neq q\}.$$

In particular, if R is the root of DT then $R \otimes R$ is the set of all pairs of distinct particles.

The set $\{\{A_1, B_1\}, \{A_2, B_2\}, \cdots, \{A_k, B_k\}\}$ is said to be a *realization* of $A \otimes B$ if

- A_i and B_i are nodes of DT for all $i = 1, 2, \cdots, k$.
- $A_i \subseteq A$ and $B_i \subseteq B$ for all $i = 1, 2, \cdots, k$.
- For each $i = 1, 2, \cdots, k$, either $A_i \cap B_i = \emptyset$ or $A_i = B_i$ is a leaf of DT.
- $(A_i \otimes B_i) \cap (A_j \otimes B_j) = \emptyset$ for all i, j such that $1 \leq i < j \leq k$.
- $A \otimes B = \cup_{i=i}^{k} A_i \otimes B_i$.

We say that such a realization is an *α-well-separated realization* (or just a well-separated realization) if it satisfies the additional property

- If $A_i \neq B_i$, then $\mathsf{Pair}(A_i, B_i)$ is α-well-separated for all $i = 1, 2, \cdots, k$.

Notice that any pair of particles $\{p, q\} \in A \otimes B$ occurs in exactly one of the interaction products $A_i \otimes B_i$ of the realization. We will see in the next section that this corresponds to the potential generated by particle p being counted exactly once in the potential at point q, which is exactly what we need for a valid solution to the n-body problem.

To create a well-separated realization, we use the easily verified fact that if P_1 and P_2 are children of P in DT, then the interaction product satisfies the property

$$P \otimes P = (P_1 \otimes P_2) \cup (P_1 \otimes P_1) \cup (P_2 \otimes P_2),$$

where the three interaction products in this union are disjoint. Repeating this property from the root R of DT, we see that

$$R \otimes R = \bigcup_{\substack{\{P,Q\} \text{ siblings} \\ \text{in } DT}} (P \otimes Q) \quad \cup \bigcup_{\substack{L \text{ a leaf} \\ \text{in } DT}} (L \otimes L).$$

Since all of these interaction products are disjoint, realizations can be combined in exactly this same manner. In other words, if the set of all siblings in DT is $\{\{P_1, Q_1\}, \{P_2, Q_2\}, \cdots, \{P_k, Q_k\}\}$, and if R_i is a well-separated realization of $P_i \otimes Q_i$ for each $i = 1, 2, \cdots, k$, then the set

$$\bigcup_{i=1,\cdots,k} R_i \quad \cup \bigcup_{\substack{L \text{ a leaf} \\ \text{in } DT}} \{\{L, L\}\}$$

is a well-separated realization of $R \otimes R$.

Making a well-separated realization for $P \otimes Q$ is fairly easy when P and Q are disjoint (as are siblings in DT). The algorithm for doing this is called MakeSibTree and is shown in Figure 2. This algorithm actually takes a pair (P, Q) (recall that this is an ordered pair) where P and Q are disjoint, and returns a tree whose nodes are labeled with pairs of DT nodes and whose leaf labels are exactly the pairs of the desired well-separated realization of $P \otimes Q$. We call this tree the *realization tree* of (P, Q), and the collection of all such trees for all sibling pairs of DT is called the *realization forest* and denoted RF. The leaves of all trees in RF, combined with the set containing each leaf node of DT paired with itself, gives a well-separated realization of $R \otimes R$ which can be used by our multipole algorithm.

The algorithm obviously terminates, since leaves of DT are eventually reached, and any pair of two leaves is well-separated. Furthermore, since only α-well-separated pairs are not processed further, the leaves of RF clearly give a well-separated realization.

```
MakeSibTree((P, Q))
   if (P, Q) is α-well-separated then
      Make a single node tree T with root labeled with (P, Q);
   else
      let P_1 and P_2 denote the two children of P in DT;
      T_1 = MakeSibTree(Pair(P_1, Q));
      T_2 = MakeSibTree(Pair(P_2, Q));
      Make new tree T with root labeled with (P, Q), left subtree T_1 and right subtree T_2;
   endif
   return T;
end
```

Figure 2: Function MakeSibTree, which returns a labeled binary tree.

In the remainder of this section, we will show that the total size of RF is linear in the size of DT. Using our decomposition tree from the preceding section, we have a realization forest (and hence a realization) of size $O(n/s)$.

Since we are dealing with many trees, we will sometimes prefix common graph terminology with DT or RF to denote which tree(s) we are talking about. For example, "A is a DT-ancestor of B" means that A and B are nodes in DT, and A is an ancestor of B in DT.

Lemma 3.1 If (P, Q) labels a node in RF, then P is neither an DT-ancestor nor a DT-descendant of Q. Furthermore, the root of the RF-tree containing (P, Q) has label Pair(LeftChild(L), RightChild(L)), where L is the least common ancestor of P and Q in DT.

Proof: Consider any node in RF, with label (P, Q). This node exists in some tree of RF whose root is labeled with a pair of siblings, say (A, B). Due to the operations of MakeSibTree, either P is a DT-descendant of A and Q is a DT-descendant of B or vice-versa. Since A and B are siblings, they are disjoint regions; therefore, P and Q are also disjoint regions, and neither can be a DT-ancestor of the other, proving the first claim of the lemma. It also easily follows from this observation that the parent of siblings A and B must be the least common ancestor of P and Q (in DT), proving the second claim. ∎

Lemma 3.2 If (P, Q) is a node in RF, then for all A that are proper ancestors of Q, (P, A) cannot label any node in RF.

Note: Remember that (P, A) is an *ordered* pair. There may very well be a pair (A, P) in RF where

A is a proper ancestor of Q.

Proof: For the sake of contradiction, assume that (P, A) is in RF for some A that is a proper ancestor of Q. By Lemma 3.1 we know that A cannot be an ancestor of P, and that (P, A) must be in the same RF-tree as (P, Q) (since $LCA(P, A) = LCA(P, Q)$).

Consider how procedure MAKESIBTREE computes a realization for pair (P, A). We know that A is an internal node of DT, and due to the ordering we therefore know that P is an internal node with $l_{\max}(\hat{P}) \geq l_{\max}(\hat{A})$. Thus P has two DT-children, say P' and P'', and MAKESIBTREE creates RF-subtrees for $\mathsf{Pair}(P', A)$ and $\mathsf{Pair}(P'', A)$ — since P has already been split, it cannot be paired with Q when MAKESIBTREE eventually subdivides A to form a pair containing Q, so it is impossible for (P, Q) to be in RF. This contradicts a basic premise of the lemma, and the lemma is proved by contradiction. ▮

Lemma 3.3 The total number of nodes in RF is $O(|DT|)$.

Proof: We will bound the number of *internal* nodes in RF, since the number of leaves is at most twice the number of internal nodes. For each $P \in DT$ define the set

$$S_P = \{Q \mid (P, Q) \text{ is an internal node in } RF\}.$$

We will show that S_P has constant size for all $P \in DT$.

First, since (P, Q) is an internal node of RF for each $Q \in S_P$, (P, Q) is not an α-well-separated pair. If Q is an internal DT-node, then $l_{\max}(\hat{P}) \geq l_{\max}(\hat{Q})$, and the smallest d-ball that can enclose both $\hat{P}$ and $\hat{Q}$ (separately) has radius $(\sqrt{d}/2) l_{\max}(\hat{P})$. If Q is a leaf, it may be any size, but $\hat{P}$ can still be enclosed in such a d-ball. Since (P, Q) is not α-well-separated, the distance between the two d-balls in the first case, and between the d-ball enclosing $\hat{P}$ to region $\hat{Q}$ in the second case, must be less than $\alpha(\sqrt{d}/2) l_{\max}(\hat{P})$. Taking into account the size of the spheres, the resulting necessary (although not sufficient) condition for (P, Q) to be not α-well-separated is

$$d(\hat{P}, \hat{Q}) < \left[\alpha(\sqrt{d}/2) + \sqrt{d}\right] l_{\max}(\hat{P}) = (\alpha/2 + 1)\sqrt{d}\; l_{\max}(\hat{P}).$$

Put another way, any $Q \in S_P$ must overlap a d-cube with side length $\left[(\alpha + 2)\sqrt{d} + 1\right] l_{\max}(\hat{P})$.

Next, for any $Q \in S_P$, consider two cases:

Case 1. $\mathsf{p}(Q)$ is an ancestor of P.

In this case, obviously $l_{\max}(\widehat{\mathsf{p}(Q)}) \geq l_{\max}(\hat{P})$.

Case 2. $\mathsf{p}(Q)$ is not an ancestor of P.

In this case, there must be some internal node of RF in which $\mathsf{p}(Q)$ is paired with some node

A, where A is an ancestor (not necessarily proper) of P, and such that MAKESIBTREE splits $\mathsf{p}(Q)$ and makes two subtrees, one of which is rooted at $\mathsf{Pair}(Q, A)$. Since $\mathsf{p}(Q)$ was split and A is an ancestor of P, we know that $l_{\max}(\widehat{\mathsf{p}(Q)}) \geq l_{\max}(\hat{A}) \geq l_{\max}(\hat{P})$.

Notice that in both cases, $l_{\max}(\widehat{\mathsf{p}(Q)}) \geq l_{\max}(\hat{P})$.

By Lemma 2.5, there is a d-cube of size $(1/2)l_{\max}(\widehat{\mathsf{p}(Q)}) \geq (1/2)l_{\max}(\hat{P})$ that covers Q and at most d others that can be paired with P in RF (since at most one ancestor of each of the d leaves mentioned in Lemma 2.5 can be paired with P). Furthermore, Lemma 3.2 shows that all $Q \in S_P$ are disjoint, and a packing argument (given explicitly by Callahan and Kosaraju [6, Lemma 4.1]) shows that there are at most $(3\left[(\alpha+2)\sqrt{d}+1\right]+2)^d$ such non-overlapping d-cubes not well-separated from P. Therefore, there are at most

$$(d+1)\left(3\left[(\alpha+2)\sqrt{d}+1\right]+2\right)^d$$

Q in S_P, which is constant for constant d.

Finally, since the size of S_P is constant for all P, the number of internal nodes of RF must be linear in the size of DT. ▮

While the realization forest was useful for describing the algorithm and proving a bound on the size of the constructed realization, it is not necessary to actually construct the forest. In the following sections we don't refer to RF at all, as we only need the realization that this process produces. To summarize this section, we give the following theorem.

Theorem 3.1 Given an internal fair split tree DT with $|DT|$ nodes and a separation constant $\alpha > 1$, we can compute an α-well-separated decomposition of size $O(|DT|)$ in time $O(|DT|)$.

4 The Multipole Algorithm

The key observation that drives the multipole algorithm is that the potential field generated by a set of charged particles is analytic in any region not containing those particles, and may be approximated by a finite prefix of some power series. For the power series to be accurate, we must evaluate it only in areas that are not "close to" a singularity (i.e., a particle generating the potential field), which is the motivation behind the notion of well-separatedness as developed in the preceding section. The following paragraphs describe the multipole algorithm at a very high level, which is presented initially as a sequential algorithm, with the parallel version presented at the end of this section. In the following two sections we will give specific details and series' for 2-dimensional and 3-dimensional problems.

A typical situation is shown in Figure 3, where the power series is of the traditional kind (such as a Taylor series) that converges within a d-ball. We will refer to this type of power series as a "local

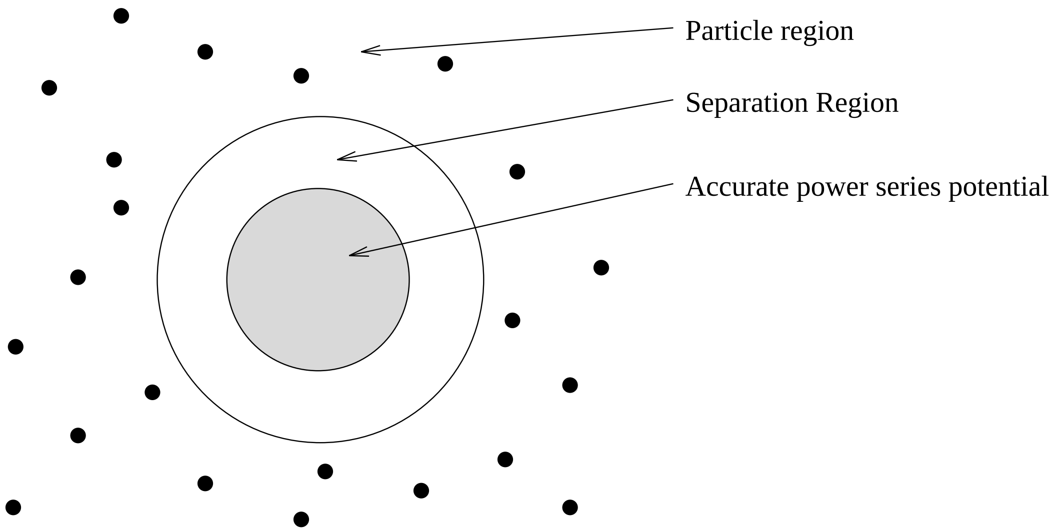

Figure 3: Power series representation of potential field.

expansion", since it converges in a local region; this distinguishes it from multipole expansions, which we will define later. In this case, we are interested in the potential field generated by the particles scattered around the outside of the circles. The local expansion will converge at all points within the larger circle, and will converge quickly within the smaller circle. By "converge quickly" we mean that the potential field is well-approximated by a finite prefix of the power series. Our goal is to compute, for each node $P \in DT$, an approximation $\Psi_P(\mathbf{x})$ of the potential field within P that is generated by all points in DT-nodes that are well-separated from P. Then, by evaluating this power series and adding in the potential from regions that are not well-separated, we have completed the n-body computation. Note that the local expansion is actually a power series in $(\mathbf{x} - \mathbf{x}_0)$, where $\mathbf{x}_0$ is the center of our desired region of convergence. When actual series' are discussed in the following sections, we will also discuss translating these series' to a common center so that they can be combined.

Unfortunately, finding all these $\Psi_P(\mathbf{x})$ series' takes too long without the following observation, which allows hierarchical composition of potential fields in order to reduce the time required. We will need a different kind of power series expansion called a multipole expansion, which is what gives this class of fast n-body algorithms its name, and was the key insight of Greengard and Rokhlin that spawned the active research into multipole algorithms. A multipole expansion is easily understood in two dimensions: As explained in the introduction, in two dimensions we can treat point coordinates as complex numbers. In this notation, the local expansion is a Taylor series in the complex variable z, and the multipole expansion will be a power series in $1/z$. It should be

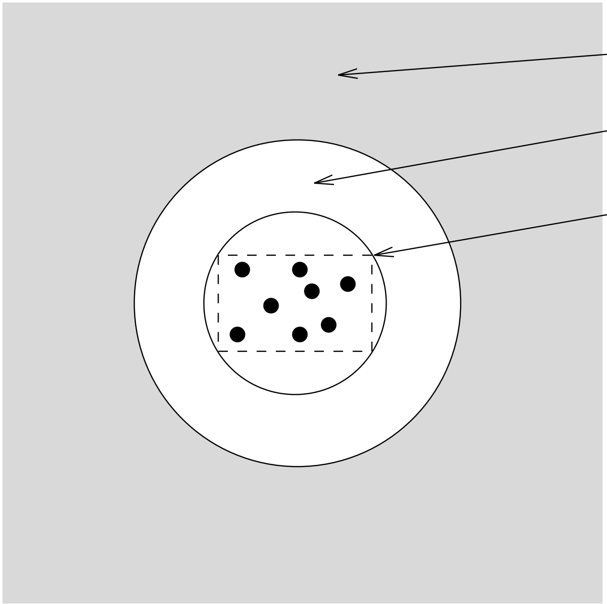

Figure 4: Multipole representation of potential field.

clear that such a series will converge in a region *external* to some ball, and is, in a way, a perfect complement to the more common Taylor series.

Figure 4 illustrates a multipole expansion. The infinite multipole series represents the potential field due to the particles within the inner circle, and converges at all points outside this smaller circle. However, we can't deal with infinite series', so as before, we use a finite prefix of the multipole series. Just as in the preceding series, this finite series will be accurate within a region separated from any singularities, which is shown as the shaded region outside the larger disk in Figure 4.

For each node $P \in DT$, we will compute the multipole expansion representing the potential due to all particles covered by P, and we will refer to this expansion as $\Phi_P(\mathbf{x})$. Multipole expansions for disjoint sets of particles can be combined to produce a new multipole expansion for the potential generated by the union of these sets of particles, so we generate all the $\Phi_P(\mathbf{x})$ expansions by explicitly generating expansions at all DT-leaves, and then sweeping up the tree from the leaves combining multipole expansions to compute the multipole expansions for the internal nodes of DT.

Finally, in order to compute all the $\Psi_P(\mathbf{x})$ series', we will be combining the potential fields that are approximated by multipole expansions. However, in order to do this, we need to be able to convert a multipole expansion into a local expansion, as illustrated in Figure 5. So to compute a series $\Psi_P(\mathbf{x})$, we combine the local expansion of P's parent ($\Psi_{\mathsf{p}(P)}(\mathbf{x})$) with the converted multipole expansions for all DT-nodes that are well-separated from P but weren't well-separated from $\mathsf{p}(P)$

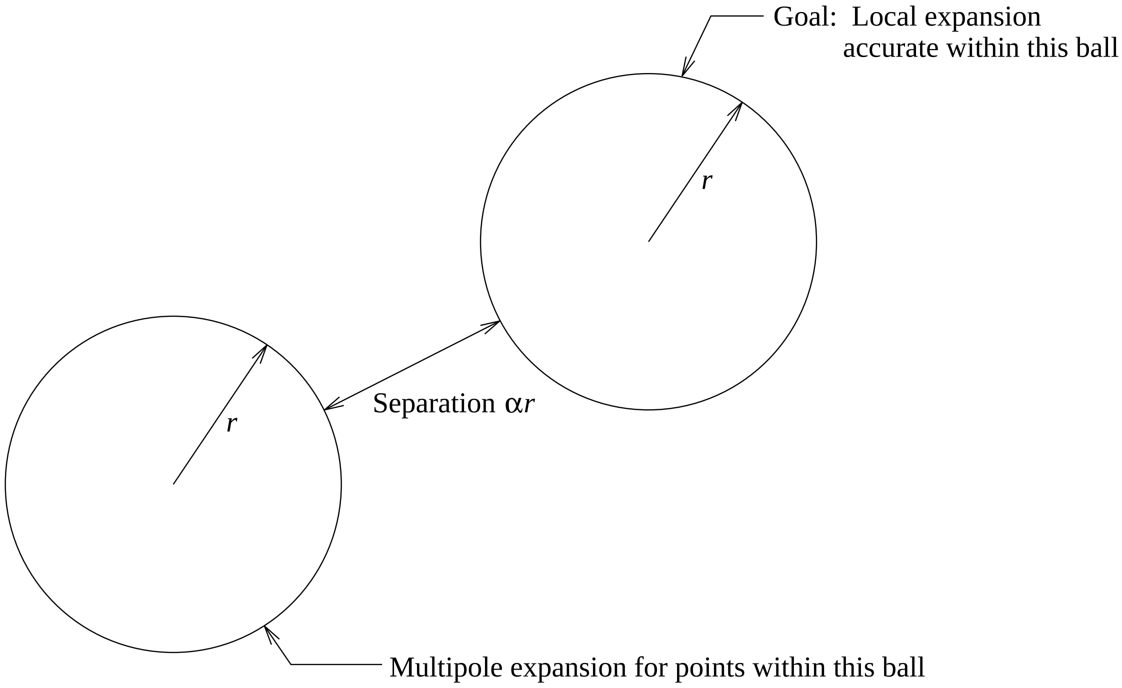

Figure 5: Conversion of a multipole representation to a power series expansion.

(these are exactly the nodes paired with P in our constructed realization). This latter value is called the "local potential" for a node P, and is denoted $\Lambda_P(\mathbf{x})$. Using the $\Phi_P(\mathbf{x})$ values computed by the preceding sweep up DT, and the computed local potentials, we can make a single sweep *down* DT to compute all the $\Psi_P(\mathbf{x})$ series'.

With a final step that computes the potential of all non-well-separated regions for each DT-leaf, we have completed the approximation of the potential on every particle of the system. The full algorithm is given below, where we postpone details of actual series' used in two and three dimensions until the following sections.

Assume we have a decomposition tree DT and a realization R. In the following description, $\mathsf{convert}(P, \Phi_Q(\mathbf{x}))$ refers to the conversion of multipole expansion $\Phi_Q(\mathbf{x})$ to a power series that is accurate within P (as illustrated in Figure 5). Furthermore, $\mathsf{makeps}(P, Q)$ refers to the procedure of constructing a power series that is accurate within P directly from the particles in region Q.

Step 1. For each DT-leaf P, compute the multipole expansion $\Phi_P(\mathbf{x})$.

Step 2. For each internal node $P \in DT$, merge the multipole expansions of its children to create the multipole expansion $\Phi_P(\mathbf{x})$.

Step 3. For each *internal* $P \in DT$, partition the nodes it's paired with in R as follows:

$$A_I(P) = \{Q \mid \mathsf{Pair}(P,Q) \in R \text{ and } Q \text{ is an internal node}\}$$

$$A_L(P) = \{Q \mid \mathsf{Pair}(P,Q) \in R \text{ and } Q \text{ is a } DT\text{-leaf}\}$$

Now for each of these $P \in DT$ compute

$$\Lambda_P(x) = \sum_{Q \in A_I(P)} \mathsf{convert}(P, \Phi_Q(\mathbf{x})) + \sum_{Q \in A_L(P)} \mathsf{makeps}(P,Q).$$

Step 4. Starting at the root of DT, define power series expansions for each $P \in DT$ as $\Psi_P(\mathbf{x}) = \Psi_{\mathsf{p}(P)}(\mathbf{x}) + \Lambda_P(\mathbf{x})$.

Step 5. For each *leaf* $P \in DT$, partition the nodes it's paired with in R as follows:

$$B_I(P) = \{Q \mid (Q,P) \in R \text{ and } Q \text{ is an internal node}\}$$

$$B_L(P) = \{Q \mid \mathsf{Pair}(P,Q) \in R \text{ and } Q \text{ is a } DT\text{-leaf}\}$$

Compute the potential at each point $p \in P$ by adding together the potentials from (a) evaluating $\Psi_P(\mathbf{x})$, (b) evaluating $\Phi_Q(\mathbf{x})$ for each $Q \in B_I(P)$, and (c) directly evaluating the potential due to particles in regions of $B_L(P)$.

Notice that all power series conversions, constructions, and evaluations occur for regions that are in the realization R, and hence are well-separated.

There are eight "basic operations" used in our multipole algorithm, which we identify below and assign notation for the time complexity of each of these basic operations. In the following table, the p refers to the precision of the multipole/power series expansions (which is related to both the input and output precision as described in the introduction), and s refers to the maximum number of particles in any leaf region of DT. The final entry, $T_{all}(p,s)$ is not a basic operation at all, but will be an important value in describing the time complexity of the overall multipole algorithm.

$T_{mc}(p,s)$	Time to create a multipole expansion.
$T_{mm}(p)$	Time to merge two multipole expansions.
$T_{m-p}(p)$	Time to convert a multipole expansion to a local (power-series) expansion.
$T_{pc}(p,s)$	Time to create a power-series expansion.
$T_{pm}(p)$	Time to merge two power-series expansions.
$T_{pe}(p,s)$	Time to evaluate power-series at s points.
$T_{me}(p,s)$	Time to evaluate multipole expansion at s points.
$T_d(s)$	Time to directly evaluate potential for s particle at s other points.
$T_{all}(p,s)$	The sum of the complexities of all eight basic operations.

Using the notation from the table above, we can state the running time of our multipole algorithm as follows.

Theorem 4.1 The multipole algorithm of this section runs in time $O(|DT| \cdot T_{all}(p, s))$.

Proof: The following is a step-by-step analysis of the running time:

Step 1. $O(|DT| \cdot T_{mc}(p, s))$.

Step 2. $O(|DT| \cdot T_{mm}(p, s))$.

Step 3. $O(|R|(T_{m-p}(p) + T_{pc}(p, s) + T_{pm}(p, s))) = O(|DT|(T_{m-p}(p) + T_{pc}(p, s) + T_{pm}(p, s)))$.

Step 4. $O(|DT| \cdot T_{pm}(p, s))$.

Step 5. $O(|DT| \cdot T_{pe}(p, s) + |R|(T_{me}(p, s) + T_d(s)) = O(|DT|(T_{pe}(p, s) + T_{me}(p, s) + T_d(s)))$.

Combining all this, we get a total time complexity of

$$O(|DT|(T_{mc}(p, s) + T_{mm}(p, s) + T_{m-p}(p) + T_{pc}(p, s) + T_{pm}(p, s) + T_{pe}(p, s) + T_{me}(p, s) + T_d(s)))$$

$$= O(|DT| \cdot T_{all}(p, s)). \qquad \blacksquare$$

4.1 Parallel Algorithm

Our parallel algorithms use the Exclusive Read/Exclusive Write (EREW) Parallel Random Access Machine (PRAM), which is a machine model in which multiple synchronized processors share a common memory but at no time step can multiple processors read or write the same memory location. The multiple access restrictions make this machine model one of the more restrictive used for parallel computation, and algorithms (such as the one in this paper) designed for an EREW PRAM can be easily used with less restrictive models. For more information on models of parallel computation, see [18] or [35].

Consider the problem of evaluating expression trees, that is, trees with the leaves labeled with numerical values and internal nodes labeled with arithmetic operations. For example, $x + y$ is represented by a tree with two leaves labeled x and y and a single internal node (the root of the tree) labeled with the "plus" operator. Any expression that can be written out as a parenthesized expression of basic arithmetic operations can be converted into an expression tree with a very simple algorithm. If each internal node gets a value which corresponds to the expression described by the subtree rooted at that node, then a natural problem is to compute the values at all internal nodes in the tree. Sequentially, this is easily performed by doing a postorder traversal of the tree, and evaluating internal nodes when they are visited; however, in parallel this problem poses a challenge if the tree is not balanced.

The widely known tree contraction algorithm was designed to solve precisely this problem. While the original work of Miller and Reif [26] produced a randomized algorithm for tree contraction, later work (such as that by Kosaraju and Delcher [21]) gave purely deterministic algorithms for tree contraction. It is these later algorithms that we use in this paper, so our algorithms are deterministic and the time bounds are true worst-case bounds. Furthermore, tree contraction does not have to work with numerical values and arithmetic operations, but rather can be defined to work with an arbitrary semigroup and semigroup operation (or with a ring or a field). For an n node tree, if it takes T_{op} time and P_{op} processors to perform the operation at a single node using the EREW model, then the entire tree contraction procedure requires $O(T_{\mathsf{op}} \log n)$ time and $O(P_{\mathsf{op}} n / \log n)$ processors. The multipole expansions together with the operation of propagating these expansions up the tree create a problem of this form (with a little bit of work describing the precise semigroup, as described below), and we will use tree contraction in order to do this first phase of our n-body parallel algorithm. In order to perform tree contraction, we must define values that the algorithm works with, and binary operators on these values.

In the case of multipole expansions, we are working with expansions that represent the potential due to a set of particles within some rectilinear region of space. The values used by the tree contraction algorithm consist of two items: a description of a rectilinear region (box) and a multipole expansion centered at the center of the box. The box must contain all the particles that contribute to the multipole expansion, but may also contain particles that do *not* contribute to the multipole expansion. The value at a leaf of DT consists of the box defined by that leaf and the multipole expansion for all the particles at that leaf. All of these values are independent, so can be computed in parallel (this is step 1 of the algorithm).

To define the operator $\circ$, consider two such values A and B. Then $A \circ B$ is a value of the same type, so has a box and a multipole expansion. The box is the smallest box large enough to enclose both the box of A and the box of B; to compute the associated multipole expansion, translate the multipole expansions for A and B to a common center at the center of the box just described for $A \circ B$, and add them together. This is just the multipole merge operation described for the sequential algorithm. Note that since this operation is defined for any two values A and B, and the box associated with $A \circ B$ is the smallest box containing both A and B, the box associated with $A \circ B$ may contain particles that do not contribute to the multipole expansion (i.e., were not particles associated with A or B), as mentioned above. This will not cause any problems.

Notice that with these definitions, if we process the tree bottom-up, starting at the leaves, then the result is precisely that described above for the sequential algorithm. However, when using tree contraction, we are not guaranteed that operations are performed strictly bottom-up (since this could only guarantee $O(\log n)$ time for trees with depth $O(\log n)$). Tree contraction requires that the operation be associative, which of course depends on the precise form of the expansions (since we are dealing with approximations). The standard expansions for both two and three dimensional n-body problems do indeed have associative merging operations, so work fine in this parallel algorithm. Therefore, we can run the tree contraction algorithm using these operators. If the expansions can be combined using $P(p)$ processors and $T(p)$ time, then the total parallel complexity for this part (step 2 of the algorithm description) is $O(P(p)|DT|/\log|DT|)$ processors and $O(T(p)\log|DT|)$ time.

Step 3 of the multipole algorithm parallelizes very easily. First all pairs (P, Q) in the realization that take part in a summation of step 3 can be easily identified, and appropriate action (either directly creating a local expansion or converting a multipole expansion to a local expansion) can be performed for each such pair in parallel. Combining these for each internal $P \in DT$ is simply a summation, which is easily parallelized.

Step 4 involves propagating local expansions down the tree, starting at the root. The desired result of this step is for each node $P \in DT$ to have a value that corresponds to the sum of all $\Lambda_Q(x)$ expansions on nodes Q that are ancestors of P. While this seems backwards from the normal tree contraction algorithm (in which values flow from the leaves toward the root), the tree contraction algorithm can still be used. Reid-Miller, Miller, and Modugno demonstrate this technique on a similar problem: given a tree in which each node contains an integer label, for each node compute the maximum value held in any of its ancestors [34]. Given box/local-expansion pairs as values, our algorithm may use the same process for step 4, giving a parallel complexity of $O(T(p)\log|DT|)$ time and $O(P(p)|DT|/\log|DT|)$ processors, where $T(p)$ and $P(p)$ denote the time and number of processors, respectively, of the merge operation on local expansions.

Step 5 consists of independent operations for each leaf in DT, and is similar to the structure of step 3, described above. Step 5 can be parallelized in an identical way to step 3.

The preceding text explains how the multipole algorithm can be parallelized, but we need to analyze its parallel complexity. Just as we used the notation $T_{all}(p, s)$ to denote, for the sequential algorithm, the total complexity of all individual operations listed for the sequential algorithm, we will assume that each individual operation can be performed in parallel using parallel time $PT(p, s)$ with $PP(p, s)$ processors. The result is then given in the following theorem, whose proof is straight-forward and omitted.

Theorem 4.2 The parallel multipole algorithm described in this section runs in $O(PT(p, s)\log|DT|)$ time using $O(PP(p, s)|DT|/\log|DT|)$ processors.

5 Application to 2-Dimensional Problems

Our most dramatic improvement over previous algorithms is in the 2-dimensional n-body problem (and thus also for Trummer's problem). As described in the introduction, in two dimensions we can treat point coordinates as complex numbers, where the real part corresponds to the x coordinate, and the imaginary part corresponds to the y coordinate.

The form of the multipole expansions used for the 2-dimensional n-body problem was first given by Greengard and Rokhlin [14, Theorem 2.1], and we repeat it here for easy reference (the notation has been changed to correspond to the notation of the current paper).

Theorem 5.1 Let $\alpha > 1$ be the separation constant. Suppose that s charges of strengths $q_1, q_2, \cdots, q_s$ are located at points $z_1, z_2, \cdots, z_s$, respectively, with $|z_i| < r$ for all $1 \leq i \leq s$. Then for any $z \in \mathbf{C}$ with $|z| > \alpha r$, the potential $\Phi(z)$ induced by these points is given by

$$\Phi(z) = Q \ln(z) + \sum_{k=1}^{\infty} \frac{a_k}{z^k}, \tag{6}$$

where

$$Q = \sum_{i=1}^{s} q_i \quad \text{and} \quad a_k = \sum_{i=1}^{s} \frac{-q_i z_i^k}{k}.$$

Furthermore, by using only the first $p \geq 1$ terms of (6),

$$\left| \Phi(z) - Q \ln(z) - \sum_{k=1}^{p} \frac{a_k}{z^k} \right| \leq \left(\frac{A}{\alpha - 1} \right) \left(\frac{1}{\alpha} \right)^p,$$

where $A = \sum_{i=1}^{s} |q_i|$.

This theorem shows two main things: First, if $p = \lceil \log(A/\epsilon) \rceil$ terms of the expansion are used with a separation constant $\alpha = 2$, then the error is bounded by $A \cdot 2^{-p} \leq A \cdot 2^{\log(\epsilon/A)} = \epsilon$. Second, the coefficients of the power series expansion have a very structured form, which can be used to very efficiently calculate $\Phi_P(z)$ for any P which is a leaf of DT. Of course, most sets of particles will not be centered at the origin as required by this theorem; however, this was merely a notational convenience, and linear translations are easily accomplished, and furthermore fast algorithms for translating existing multipole expansions are given by Greengard and Rokhlin [14].

Theorem 5.2 Let P be a leaf of DT. If $s \geq p$, then the coefficients of the p term expansion of $\Phi_P(z)$ from Theorem 5.1 can be computed in $O(s \log^2 p)$ time. The coefficients may also be computed on a EREW PRAM in $O(\log^2 p \log^* p + \log(s/p))$ time using $O(s/\log^* p)$ processors.

Proof: Notice that we can compute the vector $(-a_1, -2a_2, ..., -ka_k, ..., -pa_p)^T$ by multiplying the row vector $q = (q_1, q_2, ..., q_s)$ of charges times the $s \times p$ matrix $M = (m_{ij})$ defined by $m_{ij} = z_i^j$. Each block of p consecutive rows of M forms a $p \times p$ Vandermonde matrix, so multiplying block-by-block and adding the results we can compute qM by performing $\lceil s/p \rceil$ row-vector times Vandermonde-matrix products. Due to the structure of the Vandermonde matrix, Pan has shown that each of these products can be computed in $O(p \log^2 p)$ time [31], giving a total complexity of $O(s \log^2 p)$.

Simple parallelization of the vector-times-Vandermonde algorithm given by Pan (who only describes the sequential version) [31], along with the observation that all these products can be done in parallel and then efficiently added together, gives a total parallel complexity of $O(s/\log^* p)$ processors and $O(\log^2 p \log^* p + \log(s/p))$ time. ∎

We also need to be able to construct the local expansion for a set of particles, as illustrated in Figure 3. The following theorem complements Theorem 5.1.

Theorem 5.3 Let $\alpha > 1$ be the separation constant. Suppose that s charges of strengths $q_1, q_2, \cdots, q_s$ are located at points $z_1, z_2, \cdots, z_s$, respectively, with $|z_i| \geq \alpha r$ for all $1 \leq i \leq s$. Then for any $z \in \mathbf{C}$ with $|z| < r$, the potential is given by

$$\Gamma(z) = b_0 + \sum_{k=1}^{\infty} b_k z^k, \tag{7}$$

where

$$b_0 = \sum_{i=1}^{s} q_i \ln(-z_i) \quad \text{and} \quad b_k = -\frac{1}{k} \sum_{i=1}^{s} \frac{q_i}{z_i^k}.$$

Furthermore, by using only the first $p \geq 1$ terms of (7),

$$\left| \Gamma(z) - b_0 - \sum_{k=1}^{p} b_k z^k \right| \leq \left(\frac{A}{\alpha - 1} \right) \left(\frac{1}{\alpha} \right)^p,$$

where $A = \sum_{i=1}^{s} |q_i|$.

Notice that the coefficients (the b_i's) of the local expansion are of a very similar form to those of the multipole expansion, the only difference being that we use z_i rather than $1/z_i$ for the local expansion. Therefore, the computational structure is again that of a row vector times a Vandermonde matrix, and an almost identical algorithm to that for multipole expansion construction (Theorem 5.2) gives the following theorem.

Theorem 5.4 Let P be any node of DT, and let Q be a leaf of DT such that Pair(P, Q) is well-separated. If $s \geq p$, then the coefficients of the p term local expansion representing the potential in region P due to the particles in Q (the makeps operation from Step 3 of the multipole algorithm description) can be computed by a sequential algorithm in $O(s \log^2 p)$ time, and by a parallel algorithm in time $O(\log^2 p \log^* p + \log(s/p))$ using $O(s/\log^* p)$ processors.

Greengard and Rokhlin have given efficient methods for merging multipole expansions and converting multipole expansions to power series expansions [14]. Their algorithms are based on convolutions that can be performed using the Fast Fourier Transform (FFT), and have complexity $O(p \log p)$. In parallelizing these operations we could use the fastest $O(\log p)$ time algorithms for FFT, but this would require $O(p)$ processors, which is more than the other steps of the algorithm. We can afford to slow down this algorithm to use only $O(p/\log p)$ processors, making the parallel time $O(\log^2 p)$. For the remaining operation on the different series', translation of the local expansion, Greengard and Rokhlin used convolution methods; we note that since a truncated power series is simply a polynomial, we can use a standardized polynomial representation regardless of the center point of the power series' region of convergence, and translation of these polynomials therefore costs nothing — thus the power series merge operation involves simply adding coefficient of the corresponding power series', and so the complexity is linear in the size of the series.

Evaluating either a truncated multipole expansion or a truncated Taylor series is simply a multipoint polynomial evaluation, so using standard methods (originally from [27, 22], but see [1] or [4] for a textbook description) we can evaluate a p term polynomial at $s \geq p$ points in time $O(s \log^2 p)$. The corresponding parallel algorithm runs in time $O(\log^2 p \log^* p)$ with $O(s/\log^* p)$ processors.

The last operation listed in the table, direct evaluation of the potential, is directly solved by Gerasoulis [9] for sequential models of computation. We note that his algorithm requires only a constant number of multipoint polynomial evaluations and polynomial interpolations, so can be easily and efficiently parallelized.

The following table summarizes the time required for each operation in the two-dimensional multipole algorithm.

Operation	Sequential Time	Parallel Processors	Parallel Time	Reference
$T_{mc}(p,s)$	$O(s \log^2 p)$	$O(s/log^* p)$	$O(\log^2 p \log^* p + \log(s/p))$	Theorem 5.2
$T_{mm}(p)$	$O(p \log p)$	$O(p/\log p)$	$O(\log^2 p)$	[14]
$T_{m-p}(p)$	$O(p \log p)$	$O(p/\log p)$	$O(\log^2 p)$	[14]
$T_{pc}(p,s)$	$O(s \log^2 p)$	$O(s/log^* p)$	$O(\log^2 p \log^* p + \log(s/p))$	Theorem 5.4
$T_{pm}(p)$	$O(p)$	$O(p)$	$O(1)$	See notes above
$T_{pe}(p,s)$	$O(s \log^2 p)$	$O(s/\log^* p)$	$O(\log^2 p \log^* p)$	Multi-point evaluation
$T_{me}(p,s)$	$O(s \log^2 p)$	$O(s/\log^* p)$	$O(\log^2 p \log^* p)$	Multi-point evaluation
$T_d(s)$	$O(s \log^2 s)$	$O(s/\log^* s)$	$O(\log^2 s \log^* s)$	[9]

By setting $s = p$, the sequential complexity of each of these operations is $O(s \log^2 s) = O(s \log^2 p)$, and the parallel algorithm uses $O(s/\log^* p)$ processors and $O(\log^2 p \log^* p)$ time. Plugging in to Theorem 4.1 we get the following, which is the main result of this section.

Theorem 5.5 Combining the operations of this section with the general multipole algorithm of the previous section gives a two-dimensional n-body algorithm with sequential time complexity $O(n\log^2 p)$. The parallel algorithm uses $O(n/(\log n \log^* p))$ processors and $O(\log n \log^2 p \log^* p)$ time.

6 Application to 3-Dimensional Problems

Application of our techniques in three dimensions is based more heavily on previously studied techniques, and the speedup over previous algorithms is not as dramatic as in the two-dimensional case. In fact, the power series (both multipole and local expansions) are exactly of the form used by Greengard and Rokhlin, and we combine power series manipulations from two of their papers with our improved decomposition techniques; therefore, we simply state the complexity of the various operations here with references to the appropriate papers. For general discussion of the expansions used in three dimensions, see Greengard's dissertation [10, Chapter 3]. The parallelization of these operations is straight-forward, once it is noted that the series creation and translation operators are dominated by a computation of p terms of p different linear recurrences, which can be done in $O(\log p)$ time with $O(p^2/\log p)$ processors by standard techniques [16].

Operation	Sequential Time	Parallel Processors	Parallel Time	Reference
$T_{mc}(p,s)$	$O(sp^2)$	$O(sp^2/\log p)$	$O(\log p)$	[10, Chapter 3]
$T_{mm}(p)$	$O(p^2\log p)$	$O(p^2)$	$O(\log p)$	[14]
$T_{m-p}(p)$	$O(p^2\log p)$	$O(p^2)$	$O(\log p)$	[14]
$T_{pc}(p,s)$	$O(sp^2)$	$O(sp^2/\log p)$	$O(\log p)$	[10, Chapter 3]
$T_{pm}(p)$	$O(p^2\log p)$	$O(p^2)$	$O(\log p)$	[14]
$T_{pe}(p,s)$	$O(sp^2)$	$O(sp^2/\log p)$	$O(\log p)$	[10, Chapter 3]
$T_{me}(p,s)$	$O(sp^2)$	$O(sp^2/\log p)$	$O(\log p)$	[10, Chapter 3]
$T_d(s)$	$O(s^2)$	$O(s^2/\log s)$	$O(\log s)$	All-pairs computation

As in the two-dimensional case, we set $s = p$ to get our main result, stated in the following theorem, which is an improvement over the previous best algorithm which required $\Theta(np^2\log p)$ time [14].

Theorem 6.1 Combining the operations of this section with the general multipole algorithm of Section 4 gives a three-dimensional n-body algorithm with sequential time complexity $O(np^2)$. The parallel version of this algorithm runs in $O(\log n \log p)$ time using $O(np^2/(\log n \log p))$ processors.

One interesting thing to note about this theorem is that, when compared with Theorem 5.5, it seems like the three dimensional algorithm can be parallelized better than the two dimensional algorithm ($O(\log n \log p)$ time versus $O(\log n \log^2 p \log *p)$ time. However, this is not strictly true

— we have a much more work-efficient algorithm for the two-dimensional case, and as such it is harder to parallelize. If we relax the work requirement from $O(n \log^2 p)$ work to $O(np)$ work, we can construct a two-dimensional algorithm that runs in $O(\log n \log p)$ time with $O(np/(\log n \log p))$ processors (we simply use the naive $O(sp)$ work algorithm for multipoint evaluation rather than the $O(s \log^2 p)$ time algorithm).

7 Conclusion

In this paper, we have presented improved algorithms for the static n-body problem in both two and three dimensions, and parallelizations of these algorithms. In addition, we have given spatial decomposition algorithms that can be used to support our static algorithms.

The spatial decomposition algorithm to use depends on the assumptions that can be made about the input, and the results are summarized in the following table.

Assumptions	Time
No assumptions	$O(n \log n)$
$O(\log n)$ bit inputs	$O(n \log \log n)$
Uniformly distributed particles	$O(n)$ expected time

For the static n-body problem, the previous best algorithms required time $\Theta(np \log p)$ in two dimensions, and $\Theta(np^2 \log p)$ in three dimensions. Our results from this paper are summarized in the following table.

Problem	Sequential Complexity	Parallel	
		Processors	Time
Two dimensions	$O(n \log^2 p)$	$O(n/(\log n \log^* p))$	$O(\log n \log^2 p \log^* p)$
Three dimensions	$O(np^2)$	$O(np^2/(\log n \log p))$	$O(\log n \log p)$

As no lower bounds are known, each of these complexities is subject to improvement, although it seems unlikely that the two dimensional complexity can be reduced much. However, practical improvements may still be made by considering how these n-body potential field problems are used inside of larger systems. In particular, the algorithms of this paper would typically be used inside a system that simulates charged (or gravitational) particles over time, and we examine this problem from a complexity and error-analysis approach in a companion paper [37].

References

[1] Aho, A., Hopcroft, J., and Ullman, J. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.

[2] Anderson, C. R. A method of local corrections for computing the velocity field due to a distribution of vortex blobs. *Journal of Computational Physics 62* (1986), 111–123.

[3] Beveridge, D., and Jorgensen, W. L., Eds. *Computer Simulation of Chemical and Biochemical Systems*, vol. 482 of *Ann. NY Acad. Sci.* Proceedings of a 1986 Conference, 1986.

[4] Bini, D., and Pan, V. *Polynomial and Matrix Computations, Volume 1: Fundamental Algorithms.* Birkhäuser, Boston, 1994.

[5] Board, Jr., J. A., Batchelor, R. R., and Leathrum, Jr., J. F. High performance implementations of the fast multipole algorithm. In *Symposium on Parallel and Vector Computation in Heat Transfer, Proc. 1990 AIAA/ASME Thermophysics and Heat Transfer Conference* (1990).

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

[7] Carrier, J., Greengard, L., and Rokhlin, V. A fast adaptive multipole algorithm for particle simulations. *SIAM Journal of Scientific and Statistical Computing 9*, 4 (1988), 669–686.

[8] Chorin, A. J. Numerical study of slightly viscous flow. *J. Fluid Mech. 57* (1973), 785–796.

[9] Gerasoulis, A. A fast algorithm for the multiplication of generalized Hilbert matrices with vectors. *Mathematics of Computation 50*, 181 (1988), 179–188.

[10] Greengard, L. *The Rapid Evaluation of Potential Fields in Particle Systems.* MIT Press, Cambridge, MA, 1988.

[11] Greengard, L. Potential flow in channels. *SIAM Journal of Scientific and Statistical Computing 11*, 4 (July 1990), 603–620.

[12] Greengard, L., and Rokhlin, V. A fast algorithm for particle simulations. *Journal of Computational Physics 73* (1987), 325–348.

[13] Greengard, L., and Rokhlin, V. Rapid evaluation of potential fields in three dimensions. Tech. Rep. Research Report YALEU/DCS/RR-515, Yale University, Department of Computer Science, 1987.

[14] Greengard, L., and Rokhlin, V. On the efficient implementation of the fast multipole algorithm. Tech. Rep. Technical Report RR-602, Yale University, Department of Computer Science, 1988.

[15] Gunsteren, W. F. v., and Weiner, P. K., Eds. *Computer Simulations of Biomolecular Systems.* ESCOM, Leiden, 1989.

[16] Heller, D. A survey of parallel algorithms in numerical linear algebra. *SIAM Review 20*, 4 (Oct. 1978), 740–776.

[17] Hockney, R. W., and Eastwood, J. W. *Computer Simulation Using Particles.* McGraw-Hill, New York, 1981.

[18] JáJá, J. *An Introduction to Parallel Algorithms.* Addison-Wesley Publishing Company, Inc., Reading, MA, 1992.

[19] Karpus, M., and Petsko, G. A. Molecular dynamics simulations in biology. *Nature 347* (1990), 631–639.

[20] Katzenelson, J. Computational structure of the N-body problem. *SIAM Journal of Scientific and Statistical Computing 10* (1989), 787–815.

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

[22] Kung, H. T. Fast evaluation and interpolation. Carnegie-Mellon University, Dept. of Computer Science, 1973.

[23] Leathrum, Jr., J. F., and Board, Jr., J. A. Parallelization of the fast multipole algorithm using the b012 transputer network. In *Transputing T91* (Washington, D.C., 1991).

[24] Leathrum, Jr., J. F., and Board, Jr., J. A. Mapping the adaptive multipole algorithm onto mind systems. In *NASA/ICASE Workshop: Unstructured Parallel Scientific Computing* (Cambridge, MA, 1992), MIT Press.

[25] Leathrum, Jr., J. F., and Board, Jr., J. A. The parallel fast multipole algorithm in three dimensions. Tech. rep., Duke University Department of Electrical Engineering Technical Report, 1992.

[26] Miller, G. L., and Reif, J. H. Parallel tree contraction part 1: Fundamentals. In *Randomness and Computation*, S. Micali, Ed., vol. 5. JAI Press, Greenwich, CT, 1989, pp. 47–72.

[27] Moenck, R., and Borodin, A. B. Fast modular transforms. *J. Comput. Syst. Sci. 8*, 3 (1974), 366–386.

[28] Nabors, K., and White, J. Fastcap: A multipole accelerated 3-D capacitance extraction program. Tech. rep., MIT Department of Electrical Engineering and Computer Science, 1991.

[29] Odlyzko, A. M. The 10^{20}-th zero of the Riemann zeta function and 70 million of its neighbors. Tech. rep., ATT Bell Laboratories, Murray Hill, NJ., 1991.

[30] Odlyzko, A. M., and Schönhage, A. Fast algorithms for multiple evaluations of the Riemann zeta function. *Transactions of the American Mathematical Society 309*, 2 (Oct. 1988).

[31] Pan, V. On computations with dense structured matrices. *Mathematics of Computation 55*, 191 (1990), 179–190.

[32] Pan, V. Y., Reif, J. H., and Tate, S. R. The power of combining the techniques of algebraic and numerical computing: Improved approximate multipoint polynomial evaluation and improved multipole algorithms. In *Proceedings of the 33rd Symposium on Foundations of Computer Science* (1992), pp. 703–713.

[33] Parberry, I. *Problems on Algorithms*. Prentice Hall, Englewood Cliffs, New Jersey, 1995.

[34] Reid-Miller, M., Miller, G. L., and Modugno, F. List ranking and parallel tree contraction. In *Synthesis of Parallel Algorithms*, J. H. Reif, Ed. Morgan Kaufmann Publishers, Inc., 1993, pp. 115–194.

[35] Reif, J. H., Ed. *Synthesis of Parallel Algorithms*. Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[36] Reif, J. H., and Tate, S. R. N-body simulation I: Potential field evaluation. Tech. rep., Duke University, Deptartment of Computer Science, 1992.

[37] Reif, J. H., and Tate, S. R. N-body simulation II: Simulation of moving particles, 1995. In preparation; preliminary version appeared as "The Complexity of N-body Simulation" in *ICALP '93*, pp. 162–176.

[38] Reif, J. H., and Tate, S. R. Fast spatial decomposition and closest pair computation for limited precision input. Tech. Rep. N-96-001, University of North Texas, Department of Computer Science, 1996.

[39] Rokhlin, V. Rapid solution of integral equations of classical potential theory. *Journal of Computational Physics 60* (1983), 187–207.

[40] Schmidt, K. E., and Lee, M. A. Implementing the fast multipole method in three dimensions. *J. Stat. Phys. 63* (1991), 1220.