ELSEVIER

Parallel Computing 24 (1998) 797–822

PARALLEL COMPUTING

Scalable parallel formulations of the Barnes–Hut method for *n*-body simulations

Ananth Grama [a,*], Vipin Kumar [b,1], Ahmed Sameh [a,2]

[a] *Computer Science Department, Purdue University, West Lafayette, IN 47907, USA*
[b] *Department of Computer Science, University of Minnesota, Minneapolis, MN 55455, USA*

Received 11 December 1996; revised 24 September 1997

Abstract

The problem of simulating the motion of a set of bodies arises in a variety of domains such as astrophysics, molecular dynamics, fluid dynamics, and high energy physics. The all-to-all nature of interaction between various bodies renders this problem extremely computation-intensive. Techniques based on hierarchical approximations have effectively reduced the complexity of this problem. Coupled with parallel processing, these techniques hold the promise of large scale *n*-body simulations. In this paper, we present a spectrum of parallel formulations that are suited for different particle distributions. We first present a parallel formulation that uses a static partitioning of the domain and assignment of subdomains to processors. We demonstrate that this scheme delivers acceptable load balance, and coupled with two collective communication operations, it yields good performance. We present a second parallel formulation that combines static decomposition of the domain with an assignment of subdomains to processors based on Morton ordering. This alleviates the load imbalance inherent in the first scheme. We generalize these schemes to dynamic domain decomposition coupled with a subtree assignment that tries to optimize locality of processor subdomains. Unlike existing schemes that are based on shipping data to processors needing them, our schemes are based on shipping computation to processors where data reside. We present an experimental evaluation of our schemes on a 256 processor nCUBE2 and a 256 processor CM5. The evaluation is based on an astrophysical simulation of a variety of Gaussian and Plummer distributions of varying irregularity. We study the impact of a variety of parameters such as the impact of multipole degree and the α-criterion on accuracy and parallel performance. We demonstrate that our parallel formulations yield excellent performance and scale up to a large number of processors, making it possible to run realistic simulations with millions of particles. Furthermore, we show that as the accuracy of simulations is increased by

* Corresponding author. E-mail: ayg@cs.purdue.edu
[1] E-mail: kumar@cs.umn.edu
[2] E-mail: sameh@cs.purdue.edu

PII: S0167-8191(98)00011-8

increasing multipole degree, our formulations yield improved efficiencies.

Keywords: *n*-Body simulation; Partitioning; Load balance; Hierarchical tree code; Communication; Experimental results

1. Introduction

The simulation of the motion of a set of bodies is an important problem. It arises in a variety of domains such as astrophysics, molecular dynamics, fluid dynamics, and high energy physics. The simulation proceeds in time-steps—each time-step requiring an all-to-all force interaction. The complexity of an n body system is therefore $O(n^2)$ per time-step. Using hierarchical approximations, it is possible to reduce this complexity to $O(n\log\ n)$ [1–4].

Hierarchical approximations work on the principle that a cluster of particles sufficiently far from the observation point can be approximated by a single entity. The domain of simulation is hierarchically decomposed into a spatial tree-data structure. Each node in the tree contains a series representation of the effect of the particles contained in the subtree rooted at the node. Once the tree has been computed, the force on each particle can be computed as follows: a multipole acceptance criterion (MAC) is applied to the root of the tree to determine if an interaction can be computed; if not, the node is expanded and the process is repeated for each of the four children. The multipole acceptance criterion for the Barnes–Hut method [1] computes the ratio of the dimension of the box to the distance of the point from the center of mass of the box. If this ratio is less than some constant, α, an interaction can be computed. Other hierarchical methods proposed by Appel [5,6] and Greengard and Rokhlin [2,7,8] use alternate MACs and allow node–node interactions in addition to particle–node interactions.

Parallel formulations of the Barnes–Hut method involve partitioning the domain (or the tree) among various processors with the combined objectives of optimizing communication and balancing load. If particle densities are uniform across the domain, these objectives are easily met [9–12,8]. For irregular distributions, these objectives are hard to achieve because of the highly unstructured nature of both computation and communication. Singh et al. [13] and Warren and Salmon [14,15] present schemes for irregular distributions that try to meet these objectives. The first scheme was presented for shared-address space machines and the latter for message-passing machines. The Costzones scheme of Singh et al. [13] uses a global image of the tree for constructing a Peano–Hilbert ordering. The message-passing scheme of Warren and Salmon [14,15] constructs a Morton (or Z) curve by sorting a list of Morton keys. These spatial orderings are used to partition the domain across processors.

In addition to minimizing communication and balancing loads, treecodes for highly unstructured particle distributions present other significant problems that must be explicitly addressed. Processors traversing trees may need to address arbitrary nodes in other processors' subtrees. This must be done fast (i.e., without having to trace a chain of pointers from the root of the tree). The formulation of Singh et al. relies on cache

hardware to locate these nodes. The formulation of Salmon and Warren relies on a hash function based on Morton keys that map nodes of the tree into a memory. However, when the tree gets highly unstructured, hashing leads to considerable collision and chaining overheads if the chains are not ordered appropriately. This overhead can be minimized by sorting nodes in the chain on the frequency of node usage.

In this paper, we present schemes that address the problems of communication minimization, locating arbitrary nodes in the tree and incorporating effective load-balancing schemes. We first present a parallel formulation that uses a static partitioning of the domain and assignment of subdomains to processors. We demonstrate that this scheme delivers acceptable load balance, and coupled with two collective communication operations, it yields good performance. We present a second parallel formulation which combines static decomposition of the domain with a dynamic assignment of subdomains to processors based on Morton ordering. This alleviates the load imbalance inherent in the first scheme. We generalize these schemes to dynamic domain decomposition coupled with a subtree assignment that tries to optimize locality of processor subdomains. Unlike existing schemes that are based on shipping data to processors needing them, our schemes are based on shipping computation to processors where data reside. This alleviates the addressing problem for nodes and makes it unnecessary to use hashing functions for unbalanced tree structures. Furthermore, as accuracy of simulations is increased by increasing multipole degree, our parallel formulations yield higher efficiencies. We present detailed experimental results on up to 256 processors of a nCUBE2 and CM5 addressing various aspects of our schemes. We also study the impact of various simulation parameters on the accuracy and parallel performance of our formulations.

The rest of the paper is organized as follows: Section 2 motivates the problem and briefly outlines fast serial algorithms; Section 3 discusses new parallel formulations of Barnes–Hut method; Section 4 addresses performance characteristics of proposed schemes; Section 5 presents experimental results; and Section 6 draws conclusions and discusses the applicability of our techniques to applications in boundary element methods. Parts of this paper appeared in Refs. [16,17].

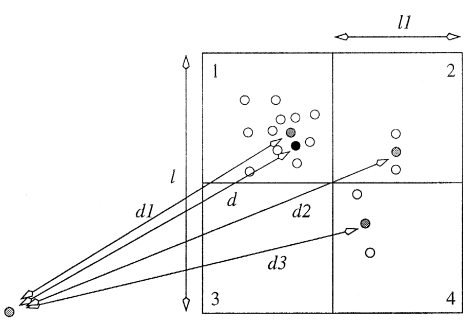
2. Applications and fast algorithms for n-body methods

A variety of physical simulations are based on *n*-body methods. Perhaps the most obvious one is the simulation of motion of heavenly bodies under gravitational forces. This gravitational force between two bodies in space separated by a distance, r, varies as $1/r^2$. For a system with two bodies, it is possible to determine the trajectories of the bodies analytically. However, for more than three bodies, one must discretize the system over time intervals and compute the forces between bodies at each snapshot.

More complicated force models arise in the solution of boundary element problems. In these problems, the boundary elements correspond to particles and the force model is defined by the Green's function of the integral equation. For applications in electromagnetic scattering, this Green's function of the Field Integral Equation (FIE) is e^{ikr}/r.

The all-to-all nature of forces in particle systems implies that an accurate formulation of the n-body problem has a $\Theta(n^2)$ complexity for an n particle system. This complexity can be reduced by exploiting the decaying nature of the interaction between bodies. For example, galaxies distant from earth can be viewed as point masses placed at their centers-of-mass. Many fast algorithms use this principle to accelerate n-body simulations. The Barnes–Hut method is one of the most popular methods due to its simplicity. It works in two phases: the tree construction phase and the force computation phase. In the tree construction phase, a spatial tree representation of the domain is derived. At each step in this phase, if the domain contains more than one particle, it is recursively divided into four equal parts (eight parts in three dimensions). This process continues until each part has a single element in it. The resulting tree is an unstructured quad-tree (oct-tree in three dimensions). This tree is now traversed in post-order. Each internal node in the tree computes and stores an approximate representation of the particles contained in that sub-tree. For astrophysical simulations, this can often be approximated by the center of mass of the particles contained in the tree. Once the tree has been computed, the force on each particle can be computed as follows: the multipole acceptance criterion is applied to the root of the tree to determine if an interaction can be computed; if not, the node is expanded and the process is repeated for each of the four (or eight) children. The multipole acceptance criterion for the Barnes–Hut method computes the ratio of the dimension of the box to the distance of the point from the center of mass of the box. If this ratio is less than some constant α, an interaction can be computed. This is illustrated in Fig. 1.

For a balanced tree, each of the n particles needs $O(\log\ n)$ interactions. This results in a total computational complexity of $O(n\log\ n)$. However, the tree size can be made arbitrarily large by bringing a pair of particles successively closer. The corresponding tree needs a large number of boxes to resolve the pair into separate boxes. Due to this, the worst case complexity of this technique is unbounded [3,4]. However, using box-collapsing techniques (the box is first collapsed to the smallest box that contains all




```
if (l/d < a)
    compute direct force interaction
    with the center of mass of domain.
else
    if (l1/d1 < a)
        compute direct force computation
        with center of mass of subdomain 1
    else
        expand subdomain 1 further

Apply similar criteria to domains 2, 3, and 4
```

Fig. 1. Illustration of the serial Barnes–Hut method.

the particles in the subdomain), this complexity can be reduced to $O(n \log n)$ [4]. The average case error of Barnes–Hut is good, but in the worst case, it is unbounded. Warren and Salmon [14] present a variant of the Barnes–Hut method that has a good worst-case error bound. There are some recent results demonstrating that it is beneficial to work with binary trees as opposed to higher-order trees [18]. Binary trees with controlled split allow better aspect ratios for partitions while reducing the number of nodes in the tree.

The fast multipole method (FMM) of Greengard and Rokhlin [2] is another hierarchical technique for computing n-body interactions. Unlike the Barnes–Hut method, FMM computes potentials instead of forces. These potentials may be electrostatic, gravitational or others depending on the application. It is easy to see that force is equal to the gradient of potential, and therefore can be easily computed from the latter. Furthermore, since potential is a scalar field, it simplifies many computations. FMM computes the potential due to a cluster of particles at the center of well-separated clusters. This can then be disseminated to individual particle positions to determine required potentials. FMM, therefore, uses cluster–cluster interactions in addition to particle–cluster interactions. The computational complexity of FMM was originally shown to be $O(n)$. However, due to degeneracy of trees, this can be unbounded. Once again, using box-collapsing techniques, it is possible bound this complexity. Unlike Barnes–Hut method, FMM has proven the worst-case error bounds [7]. Parallel formulations of FMM and the Barnes–Hut method are similar. In this paper, we focus only on the Barnes–Hut method but the techniques can be extended to FMM.

3. Parallel formulations of treecodes

A single time-step of the n-body simulation is a sequence of the following phases: tree construction, tree traversal (or force computation), and particle advance. Each of these phases must be performed in parallel. This is because the tree cannot be stored at a single processor due to memory limitations; and each of these may become serial bottlenecks unless parallelized. Before we discuss individual stages in detail, let us establish a framework for the parallel treecode. The overall schematic of the treecode is illustrated in Fig. 4. Processors cooperate to construct the spatial tree data structures and compute the center-of-mass information (or multipole series in FMM). At the end of this stage, each processor has a partial image of the entire tree. In our parallel formulations, we allow a certain degree of replication of the tree across processors. This is because the top nodes in the tree are repeatedly accessed. Therefore, it is logical to make these nodes available to all the processors prior to the force computation phase. The force computation phase proceeds as follows: each processor computes the force (or potential) on particles assigned to it. Particles may require interactions with nodes in the tree that are not locally available at the processor. These interactions result in interprocessor communication. Once all interactions have been computed, particles can be advanced based on the total force on particles. Let us now look at each phase of the parallel treecode in greater detail. To facilitate illustration, we use two-dimensional (2-D) instances of n-body simulations; however, all methods in this section generalize to three dimensions.

3.1. Tree construction

Let us assume that there is an initial distribution of particles to processors such that all the particles corresponding to a subtree of the hierarchical domain decomposition are assigned to a single processor. This is not a restrictive simplification; merely one that facilitates understanding the scheme. As an illustration, consider a subdomain corresponding to a node in the tree that contains particles assigned to processors, P_0 and P_1. It is possible to recursively subdivide this subdomain further until we can separate particles assigned to P_0 and P_1 into separate subdomains. The subdomains containing particles for P_0 are assigned to processor P_0, those containing particles for P_1 to processor P_1, and the empty subdomains to either of the processors. Therefore, by assigning more than one subdomain to a processor, we can always make sure that each

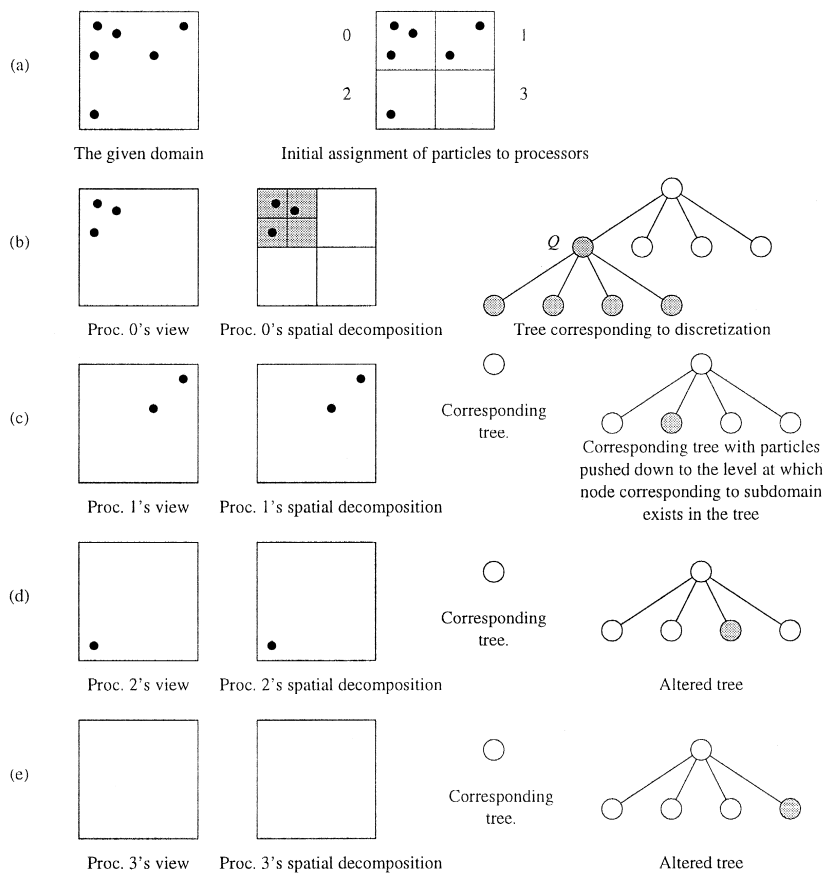


Fig. 2. Distributed tree construction. (a) Given domain and assignment of particles to processors. The value of *s* is picked to be 2. (b) Processor 0's domain. The tree is accurate at and under the node, *Q*. (c,d and e) Processor 1, 2, and 3's domains—each requiring tree adjustment to push the tree to the level of the node corresponding to the subdomain (the shaded nodes).

processor's exclusive partition corresponds to some node in the hierarchical tree data structure. Starting from such a distribution, each processor can independently construct their trees. This is done by injecting particles into the domain. Every time the domain contains more than s particles, it is split into eight octs, and each of the s particles are assigned to one of the new octs. We now try to re-inject the particle into the domain. This is done until the particle is assigned to a node.

After this phase, individual trees at processors may not be an accurate reflection of the entire tree since they do not account for particles at other processors. In general, one of the two cases occurs.

- One is when the subdomain has s particles or more. In this case, the representation is accurate at levels at or below the node in the tree that corresponds to the subdomain. The levels above this are not accurate since they do not account for nodes at other processors. This is illustrated in Fig. 2b. Here, the value of s is two. All nodes at and below the shaded node, Q, are accurately represented at processor 0.
- The other is when the subdomain contains less than s particles. This is a more difficult case in which the tree node corresponding to the subdomain does not exist in the tree. In this situation, none of the nodes in the tree are accurately represented by the local tree. To remedy this, we artificially force the particles down to the level at which the tree node corresponding to the subtree actually exists in the local tree. Note that this might increase the number of nodes in the tree slightly; but this increase is negligible. This is illustrated in Fig. 2c.

The processors must now communicate to construct the top part of the tree. This can be done in two ways—through broadcast-based and non-replicated tree constructions.

3.1.1. Broadcast-based construction

The shaded nodes in the tree represent the processor domains at the coarsest level. These nodes are referred to as *branch nodes*. Since a processor will generally get more than one subtree for load balance, it will have more than one branch node. These branch nodes can be communicated to all the processors using a single all-to-all broadcast operation. These branch nodes can now be inserted into the tree at each processor. Each processor now reconstructs the top parts of the tree independently. This results in some redundant computation but causes relatively small overhead.

3.1.2. Non-replicated tree construction

When the partitioning of trees among processors is static, it is possible to eliminate the overhead of redundant computation. If each processor knows where the parent of a specific branch node resides, the branch nodes are communicated to this processor which then computes the parent node. In the example illustrated in Fig. 2, it is easy to designate that processor 0 will hold the parent of the four branch nodes. Processors 1, 2, and 3 send their branch nodes to processor 0 which then computes the root node of the global tree-data structure. As was previously mentioned, the top levels of the tree are repeatedly accessed in the force computation phase. Therefore, this tree construction technique must be augmented with an all-to-all broadcast which makes the top-level nodes available to all the processors.

At the end of the tree construction phase, each processor has an accurate representation of the top few levels of the global tree and of everything lying beneath its branch nodes. It also maintains data structures indicating processors containing non-local branch nodes.

3.2. Force computation

Once the tree has been constructed, it must be traversed for each particle. Consider the scenario illustrated in Fig. 3. The figure illustrates a quad-tree for a 2-D simulation. In three dimensions, this will be an oct-tree. It has four branch nodes (the shaded nodes *A*, *B*, *C*, and *D*). In the force computation phase, for each particle, *i*, we start at the root of the tree and apply the multipole acceptance criterion. If the criterion is satisfied, an

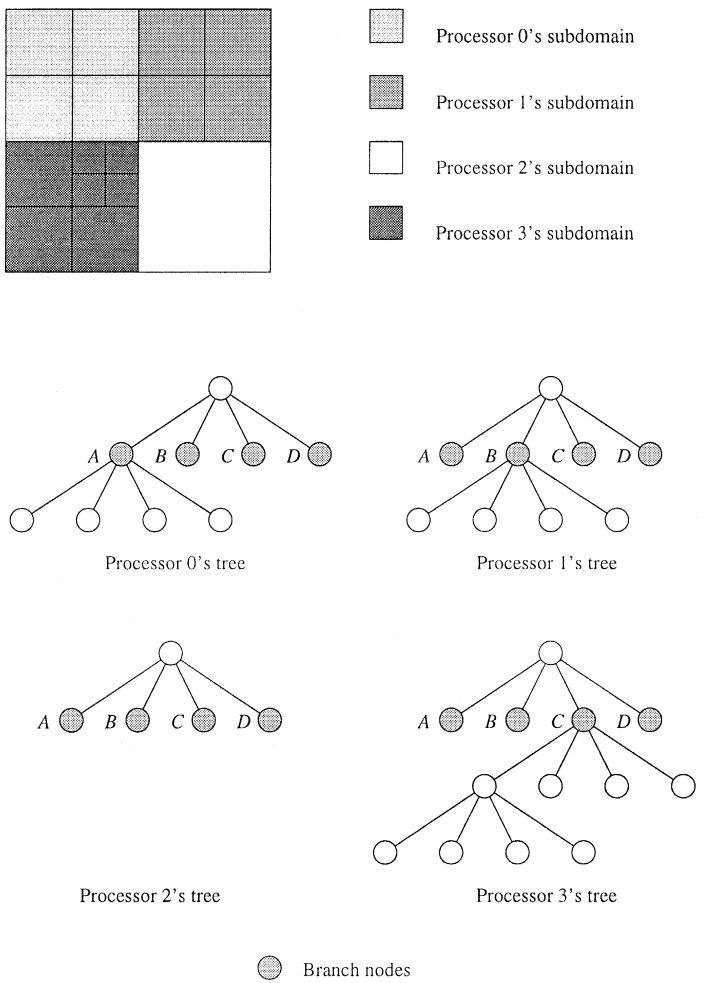


Fig. 3. Sample binary tree from the tree construction phase. The shaded nodes are the branch nodes.

interaction is computed; or else, its children must be explored. For example, In Fig. 3, the top level node is expanded to nodes *A*, *B*, *C*, and *D*. These nodes are locally available at processor 0. However, if for instance, the multipole acceptance criterion was not satisfied for node *B*, there must be communication between processors 0 and 1 to account for these nodes. There are two possible ways this communication can be accomplished: the four children of node *B* are fetched to processor 0 and the processor then applies the multiple acceptance criterion to each of these and possibly requests for more nodes. This is referred to as the *data-shipping paradigm* and is consistent with the *owner-computes* rule. Previously existing parallel formulations are based on the data-shipping paradigm. Alternately, it is possible for processor 0 to communicate the coordinates of point, *i*, to processor 1. Processor 1 then computes the contribution of the entire subtree rooted at node *B* on particle, *i*. It then sends the computed potential back to processor 0. Processor 0 adds this potential to the accrued potential for point, *i*. This communication paradigm is referred to as *function shipping* since the computation (or function) is actually being shipped to the processor that holds the data. If the simulation code was implemented in threads with a thread corresponding to each particle, this would correspond to migrating the thread as opposed to fetching the data. All the schemes presented in this paper are function-shipping schemes.

The communication in function shipping schemes involves particle coordinates. Since this corresponds to only three floating point numbers per particle, it is desirable to communicate many particle coordinates together to amortize start-up latency. For each of the assigned particles, a processor starts at the root until it reaches a non-local node (this can only be children of branch nodes). Branch nodes in the tree keep track of the processor holding their children. Using this, it is possible to determine the id of the processor to which the particle coordinates must be communicated. Furthermore, since a processor may have multiple branch nodes, it is also necessary to specify the remote branch node that a particle needs to communicate with. This is done using a unique key that is computed for each branch node. All of this information, the particle coordinates and the key, are placed in a bin meant for the remote processor. Note that the only non-local nodes that a processor can request interactions with are the branch nodes. This implies that at the receiving end, it is possible to locate this branch node quickly using the key. These keys are maintained in a hashed list of pointers that point to the actual branch nodes. Since the number of branch nodes is relatively small, searching this list does not involve significant collisions and chaining overheads.

Once a particle has been placed in a bin, the processor assumes that the corresponding interactions will be handled and proceeds to other nodes. When a bin corresponding to a processor fills up, it is communicated to the remote processor. In our implementations, we typically collect 100 particles before communicating them. This number is a function of the start-up latency and cache characteristics of individual processing elements. It is selected so that the interprocessor communication latency and memory latency at remote processor can be amortized over several particles. Processors must periodically process remote work requests. This involves reading bins from remote processors, extracting particles, computing their interactions and returning the potentials at each particle. This is necessary because the bins must use a fixed amount of memory. In our implementations, we do now allow two bins to be outstanding between the same

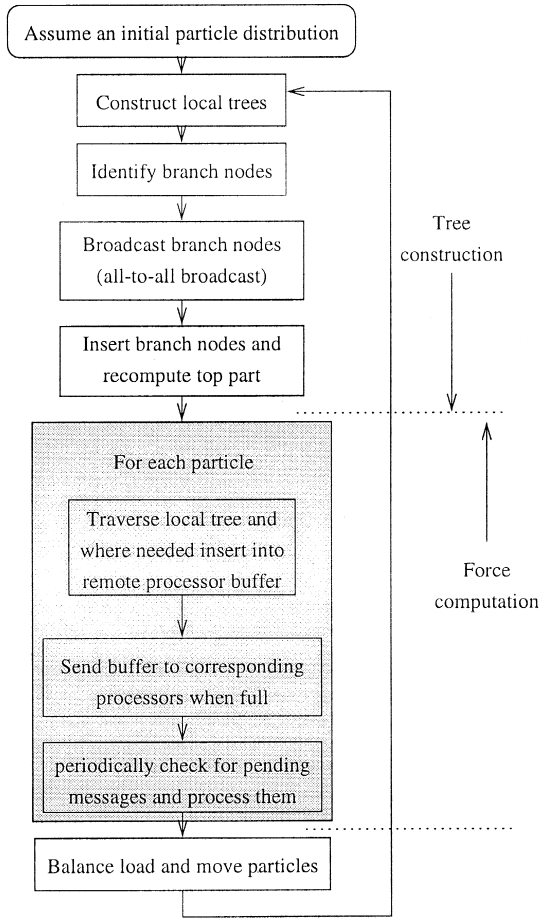


Fig. 4. A schematic of the parallel treecode formulation.

source–destination pair. Consider two processors, i and j. Assume that processor j has an outstanding bin from processor i. Now, if a second bin destined for processor j fills up at processor i before the first one has been processed, then processor i must stop processing local nodes and process outstanding nodes received from other processors.

The tree construction and force computation phases are illustrated in Fig. 4.

3.3. Load-balancing techniques

In our discussions thus far, we have assumed that the computational load is somehow distributed equally among the processors. Indeed, balancing load is critical to the performance of treecodes. In many applications such as protein synthesis, particle densities are largely uniform across the domain. (Proteins are modeled in water molecules and consequently, the variability in particle densities is less than 15–20%.) In

many other applications such as astrophysical simulations, high energy physics, and material simulation, density variations across domains maybe several orders of magnitude. Due to highly irregular particle distributions, the tree may be very imbalanced. A naive partitioning of this tree among processors may lead to significant communication and load imbalances. In this section, we will discuss load-balancing techniques that can be used with a variety of particle distributions.

Before we discuss specific load-balancing techniques, it is necessary to specify a unit of load. It is difficult to estimate the load associated with a part of the tree. However, it is reasonable to expect that the state of the system does not change drastically between two time-steps. If it does, the selected time-step is too large and as such, the simulation will be inaccurate. The number of force computations associated with a part of the tree in one time-step can be used to balance load in the next time-step. Conventional load-balancing techniques for data-shipping schemes keep track of the number of force computations associated with particles. The particles are reassigned after the time-step so that the load is balanced. For function-shipping schemes, this strategy will not work since the load is associated with the tree nodes and not the particles (because the computation is performed at processors holding the tree node). For this reason, each node in the tree keeps track of the number of particles it interacts with. This information can be used to balance the load across processors.

3.3.1. Static partitioning, static assignment (SPSA)

The simplest scheme for balancing the computational load relies on randomization. It partitions the simulated domain into r subdomains, where r is greater than the number of processors, p. By assigning r/p subdomains to each processor, it is possible to ensure a measure of load balance, provided r is sufficiently large. The parallel formulation based on static partitioning and assignment is illustrated in Fig. 5.

Since the domain of simulation is partitioned into $r(>p)$ subdomains, each processor is assigned $k=r/p$ subdomains. The value of k depends on the nature of the particle distribution. If the distribution is very irregular, k must be large so that the uneven parts of the domain can be partitioned and assigned to different processors; otherwise, the resulting load at processors would be unbalanced. On the other hand, if the domain is not highly irregular, a smaller value of k would suffice. By selecting k appropriately, the dense parts of the domain (parts that have high concentration of work) are distributed among various processors and the resulting partition and mapping are adequately load-balanced. For a two-dimensional simulation running on a d dimensional hypercube, subdomain (i,j) is assigned to processor (gray $(i,d/2)$, gray $(j,d/2)$). Here, gray (p,q) represents the pth entry in the gray-code table formed from q bits. A similar mapping that assigns neighboring subdomains to neighboring processors can be devised for a mesh, and also for three dimensional (3-D) domains. The partitioning and mapping are illustrated in Fig. 5a and b. This assignment is also referred to as *modular assignment*, and is similar to modular scatter decomposition analyzed by Nicol and Saltz [19].

This mapping simplifies many things in the parallel formulation since each processor can determine the location of all the nodes in the tree. This implies that we can use non-replicated tree construction. Furthermore, top-level nodes in the tree do not need to

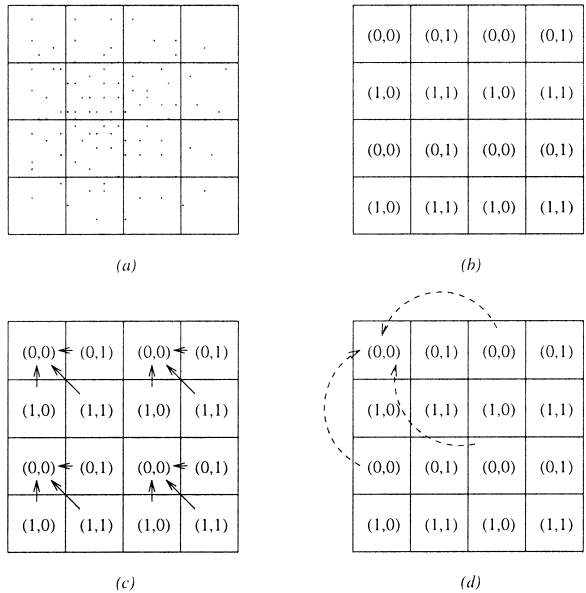


Fig. 5. (a) Partitioning the domain into $r = 16$ parts. (b) Mapping the subdomains to processors. (c and d) Communication for merging subtrees to form global tree. Note that no communication is required in (d).

store the processor ids corresponding to their children. This results in some saving in memory.

3.3.2. Static partitioning, dynamic assignment (SPDA)

The SPSA scheme relies on randomization for load balance. It is possible to combine the static partitioning of the domain into clusters with other techniques for balancing load in such a way that the resulting technique is better load-balanced and maintains the communication characteristics of this scheme. One way of achieving this is to use Morton ordering (or Peano–Hilbert ordering) for assigning clusters to processors.

In this new formulation, a Morton ordering is constructed by using the cluster coordinates (instead of particle coordinates as was the case in the scheme presented by Warren and Salmon [14]). This is illustrated in Fig. 6a. The bits of the row and column are interleaved and the boxes are labelled by the Morton number. This ordering can be computed in advance and stored in a sorted list. After an iteration, a processor computes the load in each of its clusters. This load is entered into the sorted list. Note that it is not necessary to sort the list after each iteration. Once sorted, the ordering of the clusters does not change.

After each iteration, the load of each cluster at a processor is added and a global sum is computed. The global sum is divided by the number of processors to derive the desired load at each processor during the next iteration. Each processor compares the load in the current iteration with the desired load in the next iteration. If the current load is less than the desired load, then clusters are imported from the next processor in the

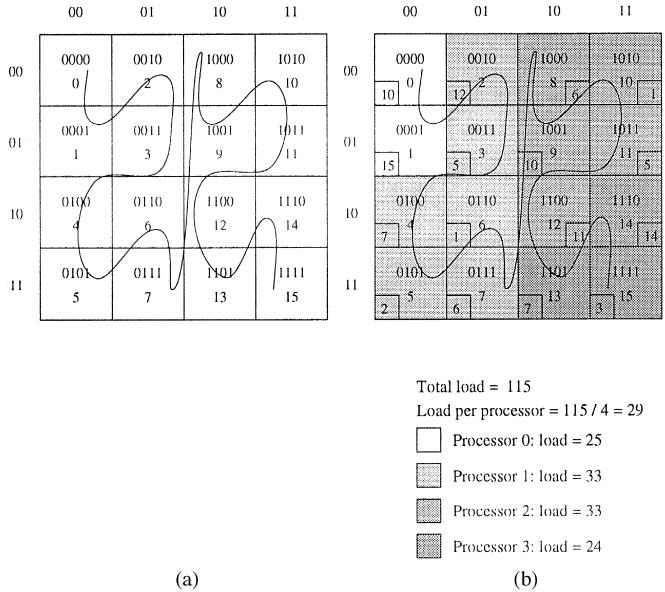


Fig. 6. (a) Morton ordering of a domain decomposed into 16 clusters. (b) An illustration of cluster mapping using the improved scheme. The numbers in the inset represent loads in each cluster. The total load is determined by adding these loads. Each processor is assigned approximately equal load in accordance with its Morton ordering.

Morton ordering. If this load is more, excess load is exported from the end of the list of clusters to the next processor. After the new cluster assignments have been computed, each processor broadcasts the starting point of its clusters in the Morton ordering. Cluster data can now be moved between processors as required by the reassignment. This data movement will not be significant since cluster loads are not expected to change drastically after each iteration.

3.3.3. Dynamic partitioning, dynamic assignment (DPDA)

When the distribution becomes very unstructured, the number of clusters, r, becomes very large. This results in excessive communication overhead. In this section, we generalize the improved cluster mapping to handle arbitrarily unstructured domains. We do this by allowing clusters of varying sizes, as opposed to fixed-size clusters in the previous case.

The new technique is essentially an efficient implementation of the Costzones scheme on message-passing computers. Each node in the tree contains a variable that stores the number of particles it interacted with. After the force computation phase, this variable is summed up along the tree. The value of load at each node now stores the number of interactions with all nodes rooted at the subtree. The loads at branch nodes are broadcast to all the processors using a single all-to-all broadcast. The non-local branch node loads are inserted into the local tree and the top level loads are recomputed. The root node at each processor now contains the total number of interactions, W, in the system. This

load must be partitioned equally among the processors; i.e., each processor must receive W/p load after load balancing. The corresponding load boundaries are 0, W/p, $2W/p$, $\ldots$, $(p-1)W/p$. The load-balancing problem now becomes one of locating these points in the tree. Each processor traverses its locally available tree in an in-order fashion and locates all load boundaries in its own domain. All particles lying in the tree between load boundaries iW/p and $(i+1)W/p$ are collected in a bin for processor, i. After each processor goes through its local tree, the points are communicated to the designated processors using a single all-to-all personalized communication [20]. The DPDA balancing technique is illustrated in Fig. 7.

In this load-balancing technique, the tree is traversed in a left to right manner. If the children of a node are ordered arbitrarily, the resulting partitions are not guaranteed to be physically contiguous. Singh et. al. [13] show that by ordering the children of a node in a specific manner while constructing the tree, it is possible to ensure that the partitions assigned to processors are contiguous in space.

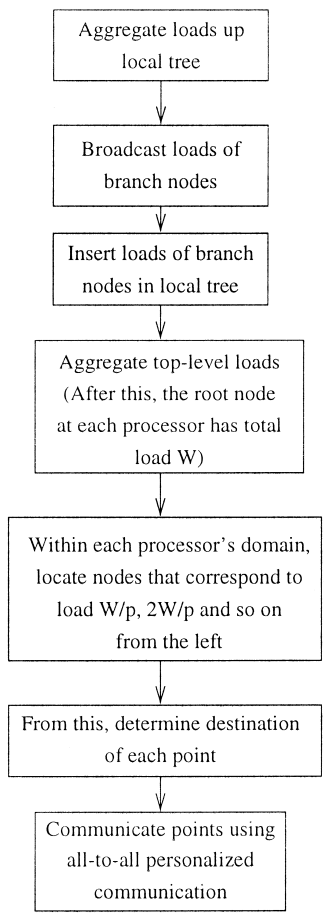


Fig. 7. A schematic of the DPDA load-balancing strategy.

4. Performance of parallel formulations

In this section, we will present analytical results for many of the parameters of our parallel formulations. We will also compare the performance of function shipping and data shipping schemes.

4.1. Number of subdomains for static decomposition

We had mentioned that the number of subdomains in static decomposition must be large enough to balance the load among processors. Furthermore, when the number becomes very large, the communication and memory overheads cause the efficiency to drop. It is, therefore, critical to identify appropriate number of subdomains. In this section, we present an analytical model for an upper bound on the number of clusters.

Kruskal and Weiss [21] analyze the problem of allocating independent subtasks to processors. Although this work cannot be directly applied to our parallel formulation, we will show how this can be used to gain an estimate of the overhead incurred in balancing load for our parallel formulation. Kruskal and Weiss derive the expected time of completion of r tasks on p processors under the assumption that the runtimes of all subtasks are independent, identically distributed random variables. The results cover a class of distribution functions that have increasing failure rates such as the normal distribution. They show that allocating an equal number of subtasks to each processor all at once yields good efficiency. We shall see that this result is of particular significance while analyzing our parallel formulation.

For the *n*-body simulation problem using the Barnes–Hut method, decomposing the domain does not yield independent subtasks. There will be a correlation between the workloads associated with each cluster. This correlation will, for most problems, be a function of the distance between the two clusters. The total processor workload is a sum of the cluster workloads over all the clusters that are assigned to a processor. In our parallel formulation, the correlation between clusters is used to balance load by assigning adjacent clusters (which have strongly correlated loads) to different processors. However, if we modify the assignment so as to assign clusters randomly to processors, we can approximate the workload on a processor by an independent random variable. Clearly, a modular assignment of clusters to processors will perform better than a random assignment, since the former uses the correlation among subtasks and the latter neglects it. Kruskal's analysis can be applied to random assignment of clusters to processors, and the expected timings can be used as an upper bound on the expected time for modular assignment of clusters to processors.

Kruskal's results state that if there are r independent subtasks with mean μ_r and standard deviation, σ, and if we assign them so that every processor gets r/p tasks at one time, the expected time of completion is given by:

$$T_p \sim \frac{r}{p}\mu_r + \sigma\sqrt{2\frac{r}{p}\log p}\,,$$

where r is large compared to $p\log\ p$. In this expression, the first term represents the actual work performed on p processors, and the second term represents overhead due to

load imbalance. It is clear that as we increase r, the first term grows linearly, whereas the second term grows as $\sqrt{r}$. Thus, on increasing r, essential computation grows faster than the overhead and consequently, the efficiency of the system increases. Conversely, if p increases, then the first term decreases faster than the second and efficiency decreases.

We can use this result to bound the performance of our parallel formulation. Assume that the total amount of work in the system (in terms of interactions) is W. The domain is divided into r equal clusters. Let $\mu_r t_c$ be the mean computation associated with each cluster and σ be its standard deviation. Thus, $r\mu_r = W$. Applying Kruskal and Weiss' result, we get:

$$T_p = \frac{t_c r \mu_r}{p} + \sigma\sqrt{2\frac{r}{p}\log p}\,. \tag{1}$$

This term does not include the communication overhead incurred by the algorithm. The first term denotes the essential computation and the second term represents the load imbalance term. For the load imbalance to grow slower than essential computation, the second term must grow slower than the first one. This yields $r \geq p\log\ p$. An important consequence of this result is that we can balance load among processors by allocating $\Theta(\log\ p)$ clusters to each processor. While this is a largely theoretical result, it is useful since we can estimate the rate of increase of number of clusters with processors if we have balanced load at a smaller number of processors.

4.2. Function shipping vs. data shipping

Let us now look at the relative merits and demerits of function and data shipping.

4.2.1. Communication volume

There is a symmetry between the communication induced by the two communication paradigms. However, in function shipping, the unit of communication is a single point's position which corresponds to three floating point numbers. In a data-shipping scheme, the entire multipole series corresponding to the cluster must be communicated in addition to the origin of the multipole series. For monopoles, the size of the series is a small constant, but for multipoles, it is a linear function of degree, k, in two dimensions and k^2 in three dimensions. For example, a multipole series corresponding to a 6 degree multipole expansion consists of 36 complex numbers or 72 floating point numbers. Clearly, in this case, data-shipping schemes require significantly higher communication than function shipping.

4.2.2. Impact of degree of multipole series on parallel efficiency

The accuracy of the simulation can be increased by increasing the multipole degree. If we increase the degree of the multipole series, the computation increases as $O(k^2)$. For function-shipping schemes, the communication overhead remains the same; hence, the parallel efficiency of the function-shipping scheme increases because the ratio of computation to communication becomes smaller. On the other hand, for data-shipping schemes, the communication volume also increases as $O(k^2)$; hence, parallel efficiency does not improve as fast.

4.2.3. The addressing problem

The data-shipping scheme may access arbitrary nodes in remote trees. It is, thus, necessary to provide fast addressing mechanisms for these nodes. This is typically accomplished using hash functions. The hash table is generally very dense since memory is a significant constraint on these simulations. Hashing highly non-uniform structures to constrained memory leads to significant collisions and chaining overhead. Chained lists must be sorted on node usage to minimize this overhead. In a function-shipping scheme, the only remote nodes that can be addressed by a processor are the branch nodes. Since the number of branch nodes is relatively small (of the order of hundreds or less), it is possible to provide very fast access to these nodes without significant memory wastage. We implement two schemes for locating branch nodes. Both schemes compute a unique key for each branch node. The first scheme maintains a hash table of these keys along with pointers to the branch nodes themselves. The second scheme maintains a sorted table of keys. Branch nodes are located using a binary search of this sorted table. In our experiments, we did not see a significant difference in the performance of these two schemes. This is because for each branch node location, we perform a significant amount of computation (interaction with the entire subtree rooted at that branch node). Therefore, slight differences in performance are largely masked by this computation. Note that in data-shipping schemes, a processor must perform one access to the hash table for each remote access. This makes the performance of data-shipping schemes more sensitive to the quality of hash table for storing nodes.

4.2.4. Working set considerations

One of the important parameters is the size of the working set. Consider a scenario when the locally essential subtree (the part of the tree required by a processor's particles to compute their interactions) does not fit in the local memory. In this case, for a data-shipping scheme, remote nodes must be fetched and all interactions corresponding to a particular node must be performed at once. Otherwise, it is likely that this node will have to be removed from the tree and then fetched again at a later time. This can drastically increase the communication overhead. Function-shipping schemes, on the other hand, control the working set size explicitly. By controlling the sizes of the message bins, we can make sure that the working set size never exceeds available memory.

5. Experimental results

In this section, we present an experimental evaluation of the parallel formulations in the context of an astrophysical simulation. The simulation code is written for an nCUBE2 and a CM5 and evaluated for up to 256 processors. The code is tested for Plummer and Gaussian distributions of varying irregularities. The number of particles in these instances ranges from a few thousands to over a million particles. Fig. 8 illustrates a sample plummer galaxy of 5000 particles. The input to the simulation code consists of particle masses, initial positions and velocity vectors. The code computes the positions and velocities at each subsequent time-step.

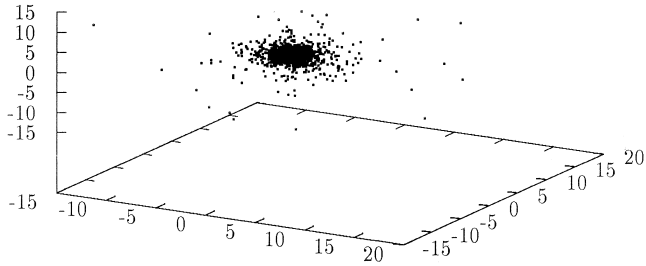


Fig. 8. Sample plummer distribution of 5000 particles (the simulation instances used in experiments are much larger).

The objectives of this experimental study are as follows:

- To study the performance and scalability of the parallel formulations proposed.
- To determine the computation time associated with each phase of the hierarchical method (tree construction, tree traversal) with a view to identifying potential bottlenecks of our formulations.
- To study the impact of various parameters (the α criterion) and the degree of the polynomial on the performance and the accuracy of the simulation.

5.1. Computing gravitational fields: Using monopoles

The first set of results are for computing vectors corresponding to the total force on a particle. This force is computed by aggregating distant sets of particles into point masses at their center of mass (monopoles). The simulation runs presented here are single iteration runs. We allow the simulation to run a few time-steps before timing an iteration. This is because during the initial iterations, there is a significant exchange of particles between processors. After a few iterations, the processor subdomains change gradually between iterations, since the load does not change significantly between iterations. The single iteration time includes the time for a single load balance cycle.

We first present runtimes of a number of simulation instances on an nCUBE2 parallel computer for up to 256 processors. The problems contain approximately 160 K, 326 K, 657 K, and 1.2 million particles. The problem instances are named as *g_n* and *p_n*. Here, *g* and *p* refer to Gaussian and Plummer distributions respectively, and *n* is the number of particles in the simulation. For instance, *g_*1 192 768 contains 1 192 768 or approximately 1.2 million particles. Some of the distributions contain multiple distributions (*g_*1 192 768 contains two Gaussian distributions). The largest problem requires approximately 1.3×10^8 force computations. A problem of this size is considered relatively small for *n*-body simulations. By demonstrating good performance even for such small problems, we effectively demonstrate the quality of these parallel formulations. On bigger problems, computation per processor will be an even bigger fraction of the total time taken, and thus, better efficiencies will be obtained. The size of the problems attempted on the nCUBE is limited by the fact that the available nCUBE2 computer only had 4 megabytes of memory at each processor. In fact, for most of the problems, the limited amount of memory makes it impossible to run these problems on a

single processor. Therefore, speed-up and efficiency results are computed by extrapolating force computation rates on a single processor.

5.1.1. Comparison of SPSA and SPDA schemes

Table 1 presents runtimes of various problems on different numbers of processors. The following conclusions can be drawn from this table: the parallel runtime of both formulations reduces consistently with increasing number of processors. For example, on increasing the number of processors from 64 to 256 (a factor of 4) for problem $g\_1\,192\,768$, the time reduces by a factor of 3.63 for the SPDA scheme and by a factor of 3.6 for the SPSA scheme. This shows that for larger problems, both schemes are scalable up to at least 256 processors. For smaller problems, the time reduces by a somewhat smaller factor. The SPSA scheme has higher runtimes because of load imbalances.

Table 2 demonstrates the effect of increasing the number of clusters on the performance of the two schemes. As we increase the number of clusters, the load balance should improve for both schemes. However, for the SPSA scheme, when the number of clusters is increased beyond a certain point, the gains in load balance can be offset by the increased communication overheads incurred during the tree construction and remote force computation phases. This is because as the clusters become small, more and more particles would require communication with neighboring clusters (that are assigned to neighboring processors). In the SPDA formulation, communication overhead does not increase with the number of clusters. However, the cost of repartitioning the clusters goes up. From Table 2, we can see that in most cases, the runtime decreases with increasing number of clusters (we could not increase the number of clusters for these problems beyond 64×64 due to memory limitations). However, for 16 processors, for the SPSA scheme, the performance degrades as we increase the number of clusters from 32×32 to 64×64.

Let us now take a closer look at the time taken by various phases of the algorithm. A breakup of the time taken by various phases of the two schemes for problems $g\_1\,192\,768$ and $g\_326\,214$ for $p = 256$ is given in Table 3. The time for local tree construction is very small for both schemes. The time taken by the SPDA scheme is slightly more than the SPSA scheme. This can be explained by the fact that in trying to

Table 1
Runtimes (in seconds) of the SPSA and SPDA schemes for various problems using monopoles

No. of processors → problem	Scheme	16	64	256
$g\_160\,535$ ($\alpha = 0.67$) $F = 2.2 \times 10^7$	SPSA	179.74	65.53	25.08
	SPDA	132.37	51.02	17.13
$g\_326\,214$ ($\alpha = 1.0$) $F = 2.1 \times 10^7$	SPSA	167.449	62.79	22.57
	SPDA	133.75	45.42	15.63
$g\_657\,499$ ($\alpha = 1.0$) $F = 4.6 \times 10^7$	SPSA		114.75	31.06
	SPDA		91.02	24.27
$g\_1\,192\,768$ ($\alpha = 1.0$) $F = 1.2 \times 10^8$	SPSA		197.51	54.86
	SPDA		163.96	45.17

F denotes the number of force computations.

Table 2
Runtimes (in seconds) for different numbers of clusters for the two parallel formulations

p	Problem	Scheme	Number of clusters		
			16×16	32×32	64×64
16	$g\_28131\ (\alpha=0.67)$	SPSA	27.16	28.12	28.22
		SPDA	23.97	21.38	20.75
	$g\_160535\ (\alpha=0.67)$	SPSA		179.74	180.6
		SPDA		139.27	132.37
	$g\_326214\ (\alpha=1.0)$	SPSA		167.45	173.615
		SPDA		133.75	134.403
64	$g\_160535\ (\alpha=0.67)$	SPSA	69.56	70.35	65.53
		SPDA	69.83	69.52	51.02
	$g\_326214\ (\alpha=1.0)$	SPSA	67.43	65.88	62.79
		SPDA	61.48	50.49	45.42
	$g\_657499\ (\alpha=1.0)$	SPSA		127.61	114.75
		SPDA		109.39	91.02
256	$g\_326214\ (\alpha=1.0)$	SPSA		28.26	22.57
		SPDA		18.19	15.63
	$g\_657499\ (\alpha=1.0)$	SPSA		40.61	31.06
		SPDA		31.48	24.27

balance load, the SPDA scheme may have vastly different number of particles at individual processors. The tree-merging cost is higher for the SPDA scheme because unlike the SPSA scheme in which all processors contained equal number of clusters, the number of clusters here may be very different. Furthermore, the systematic communication structure of the SPSA scheme does not exist for the SPDA scheme in the tree-merge phase. The time for all-to-all broadcast is comparable for the two schemes. The SPDA formulation spends less time in the force computation and tree-traversal phase because of better load balance. Finally, the SPSA scheme spends no time in balancing load since

Table 3
Time taken (in seconds) by various phases of the parallel formulations for the SPSA and SPDA schemes for problems $g\_1192768$ and g_326214 for $p=256$

Problem → phase	$g\_1192768$		g_326214	
	SPSA	SPDA	SPSA	SPDA
Local tree construction	0.004	0.0065	0.0018	0.0023
Tree merging	0.061	0.79	0.022	0.24
All-to-all broadcast	0.40	0.39	0.30	0.28
Force computation and tree traversal	53.62	42.46	21.94	14.30
Load balancing	0	0.86[b]	0	0.61[b]
Total[a]	54.86	45.17	22.57	15.63

[a] The total times are not exactly equal to the sum of individual phases. This is because of overheads incurred in other small segments of the code that set up required data structures.
[b] The load-balancing cost does not include the time for sorting the Morton numbers corresponding to the cluster numbers. This time is not included since the cost is amortized over a large number of iterations.

load balance is implicit. We can see that the SPDA scheme incurs a small overhead in balancing load.

Now, we study the impact of particle distribution on the performance of the SPDA scheme. We report results of four simulations on datasets *s_1g_a*, *s_1g_b*, *s_10g_a*, and *s_10g_b*. Each of these simulations contains 25 130 particles. Dataset *s_1g_a* has a single Gaussian distribution centered randomly in a $100 \times 100 \times 100$ simulation domain. The variance of the distribution is such that most particles lie within a $2 \times 2 \times 2$ subdomain. Dataset *s_1g_b* is a single Gaussian distribution of 25 130 particles with a lower variance. All particles are distributed in a $4 \times 4 \times 4$ subdomain. Datasets *s_10g_a* and *s_10g_b* have 10 Gaussians (each with 2513 particles) centered randomly in the domain. Dataset *s_10g_a* has a high variance with all particles belonging to a certain Gaussian distributed in a $2 \times 2 \times 2$ subdomain. Dataset *s_10g_b* has a lower variance, with all particles belonging to a certain Gaussian distributed in a $4 \times 4 \times 4$ subdomain. Table 4 presents the speed-ups obtained for these datasets on 4, 16, and 64 processors with different numbers of clusters. The speed-up results can be explained on the basis of available concurrency and load balance. Partitioning the dataset *s_1g_a* into 128×128 clusters yields very few clusters that have any particles in them. Therefore, the available concurrency is limited. Consequently, the speed-up saturates at a small number of processors ($p = 4$). Increasing the number of clusters increases the degree of concurrency and therefore, the saturation point is pushed back ($p = 16$). Decreasing the variance has the effect of providing better concurrency for the same number of clusters. Therefore, simulations on dataset *s_1g_b* yield better speed-up. Partitioning dataset *s_10g_a* into 128×128 clusters yields a larger number of clusters with particles compared to the datasets with a single Gaussian. Increasing the number of areas of particle concentrations, in general, yields higher concurrency and better load balance. Reducing the variance of distributions in dataset *s_10g_a*, as is done in dataset *s_10g_b*, yields a more regular distribution and consequently, better performance. For problems with very irregular particle concentrations, it becomes necessary to increase the number of clusters to very large values before adequate load balance can be achieved.

Table 4
Speed-up results for four problems with varying degrees of irregularities

Problem	Number of clusters	Speed-up		
		$p = 4$	$p = 16$	$p = 64$
s_1g_a ($F = 6.8 \times 10^6$)	128×128	3.1	3.07	2.98
	256×256	3.5	8.2	7.9
s_1g_b ($F = 4.9 \times 10^6$)	128×128	3.68	11.46	11.23
	256×256	3.79	12.38	20.10
s_10g_a ($F = 5.1 \times 10^6$)	128×128	3.73	12.51	28.16
	256×256	3.78	13.81	39.40
s_10g_b (4.1×10^6)	128×128	3.81	13.81	38.46
	256×256	3.80	13.83	44.18

F represents the total number of force computations. The α parameter for these simulations is set to 0.67.

5.2. Computing gravitational potentials: using multipole series

We present results for the DPDA load-balancing scheme in the context of gravitational potential instead of force. The potential is a scalar quantity and can be conveniently expressed as a series using Legendre's polynomials [7]. Vector forces can be computed using potentials by differentiating them. In this section, we will study the parallel performance of the DPDA scheme and the impact of the degree of the polynomial, the α parameter, and the overall error in simulation.

5.2.1. Parallel efficiency and overheads

We present the parallel runtime and efficiency of four different problem instances. These instances range from approximately 63 K to 354 K particles. It is impossible to run these instances on a single processor because of their computational and memory requirements. Therefore, we use the force evaluation rates of the serial and parallel versions to compute parallel efficiency. In our code, each particle–cluster interaction requires $13 + k^2 * 16$ floating point instructions, where k is the degree of polynomial used. The MAC routine requires 14 floating point instructions. The square root instruction is assumed to be a single floating point instruction. Table 5 presents the runtimes, efficiencies, and computation rates for four problems on the CM5. The value of the α parameter in each of these cases is 0.67, and the degree of the multipole expansion is 4. The efficiencies were computed by determining the sequential time for each MAC and force computation. The sequential times for the larger problem instances were projected using these values and the efficiencies computed.

A number of observations can be made from Table 5. Our parallel formulation yields excellent performance for a wide range of processors. The speed-up scales almost linearly unto 256 processors even for small problems ($g\_160\,535$). For the range of problem sizes where hierarchical methods are really useful ($g\_326\,214$ and $p\_353\,992$), the formulation yields a relative speed-up of over 3.3 on going from 64 to 256 processors. Clearly, for these problem instances, the overhead due to communication and load imbalance is low.

5.2.2. Impact of polynomial degree on performance and accuracy

We run two sets of problems with different degree polynomials for $p = 64$ and $p = 256$. Fixing the α parameter at 0.67, we note the parallel runtime and compute

Table 5
Runtimes (in seconds), efficiency, and computation rates of the CM5 for different problems for $p = 64$ and 256

Problem	$p = 64$		$p = 256$	
	Runtime	Efficiency	Runtime	Efficiency
$p\_63\,192$	21.93	0.76	8.86	0.47
$g\_160\,535$	42.35	0.84	13.34	0.67
$g\_326\,214$	88.19	0.88	26.61	0.73
$p\_353\,992$	93.74	0.89	28.29	0.74

In each of the instances, $\alpha = 0.67$, and degree = 4.

Table 6
Runtimes (in seconds), efficiency, and fractional percentage errors for different degree polynomials

Problem	Degree = 3			Degree = 4			Degree = 5		
	Time	Efficiency	Error	Time	Efficiency	Error	Time	Efficiency	Error
p = 64, α = 0.67									
*p*_63192	13.94	0.71	4.62	21.93	0.76	2.10	31.93	0.80	0.93
*g*_160535	27.90	0.76	4.90	42.35	0.84	2.43	63.31	0.86	1.21
*g*_326214	54.71	0.84	4.56	88.19	0.88	2.91	133.83	0.89	1.08
p = 256, α = 0.67									
*p*_353992	18.48	0.67	6.12	28.29	0.74	3.06	41.57	0.77	1.63

corresponding efficiencies and fractional percentage error. The fractional error is defined as follows: if $\boldsymbol{x}_k$ is the potential vector returned by the k-degree polynomial approximation and $\boldsymbol{x}$ is the accurate potential vector, then the fractional error is defined as

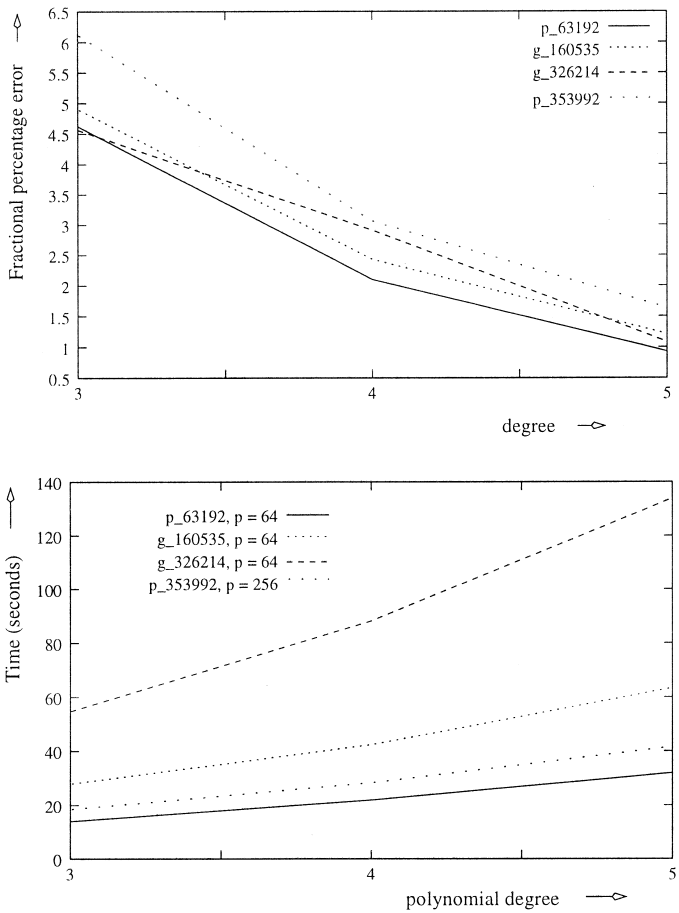


Fig. 9. Variation of fractional percentage error and parallel runtime with increase in degree of polynomial.

$\|\boldsymbol{x} - \boldsymbol{x}_k\|/\|\boldsymbol{x}\|$. When expressed as a percentage, we refer to this as the fractional percentage error of the treecode. Table 6 shows the parallel runtime, efficiency, and fractional percentage errors of several instances.

Fig. 9 presents the corresponding graphs depicting the drop in fractional percentage error and increase in parallel runtime with increase in the degree of the polynomial. This overall runtime increases as $\Theta(k^2)$ with degree of polynomial k. It is, therefore, important to determine the right error tolerance for the treecode to minimize runtime for the required tolerance. Another observation that can be made from Table 6 is that the efficiency of the parallel formulation increases with the degree of polynomial. This is consistent with our expectations since the communication overhead remains constant, but computation increases with the degree of the polynomial. This is unlike data-shipping parallel formulations in which both communication and computation increase as $\Theta(k^2)$. This is one of the big advantages of function-shipping treecodes.

5.2.3. Impact of α-parameter on performance and accuracy

We now study the impact of changing the α parameter. We run the same problem instances. In this case, the degree of the polynomial is fixed at 4 and the value of α is varied between 0.67 and 1.0. Once again, the parallel efficiency and the fractional percentage error are computed. The results are presented in Table 7.

Table 7 reveals some interesting trends. First, as expected, the runtime goes down and the fractional error increases as the value of α is increased. This is consistent with our expectations. However, the efficiency of the parallel formulations often increases with increasing α. At the first glance, this seems anomalous since we expect efficiencies to go down with decreasing problem size. However, a more careful examination reveals that as α increases, more and more interactions are accounted as near-field (localized) interactions. This results in a reduction in overall communication, and correspondingly results in an increase in efficiency. In cases where this reduced communication is not offset by reduced computation at processor, we observe higher overall efficiency.

Tables 6 and 7 illustrate some interesting trends. The gains from increasing the degree of the multiple expansion diminish but the corresponding runtime increases as $\Theta(k^2)$. Therefore, it is desirable to identify the desired error level and the corresponding degree. Higher degree polynomials are more effective in reducing overall error than

Table 7
Runtimes (in seconds), efficiency, and fractional percentage errors for different values of α

Problem	$\alpha = 0.67$			$\alpha = 0.80$			$\alpha = 1.0$		
	Time	Efficiency	Error	Time	Efficiency	Error	Time	Efficiency	Error
p = 64, degree = 4									
*p*_63192	21.93	0.76	2.10	17.43	0.75	3.11	14.92	0.72	4.91
*g*_160535	42.35	0.84	2.43	34.71	0.85	3.54	23.55	0.82	5.44
*g*_326214	88.19	0.88	2.91	64.04	0.89	3.89	45.60	0.85	5.81
p = 256, degree = 4									
*p*_353992	28.29	0.74	3.06	22.65	0.73	4.16	17.91	0.61	6.93

smaller values of α (for a given operation count). This is favorable to the parallel formulation presented in two ways:

- Lower values of α result in lower communication overhead.
- The performance of the parallel formulation improves with increasing degree of polynomial. This is not the case for data-shipping schemes.

6. Conclusions

In this paper, we presented three parallel formulations of hierarchical treecodes. These parallel formulations can be used to compute vector forces as well as scalar potentials. The first two parallel formulations are based on a static partitioning of the domain, and are particularly suitable for moderately irregular distributions. For arbitrarily unstructured domains, we present a third scheme which is a generalization of the first two schemes. All of these schemes are based on the function-shipping paradigm, as opposed to the previously used data-shipping paradigm. We show that function-shipping schemes require less communication than data-shipping schemes, particularly when particle–cluster interactions need to be computed using higher degree multipoles.

Our results on up to 256 processors of an nCUBE2 and CM5 show that even for small problem instances, it is possible to achieve high parallel efficiency. We also investigated the impact of various program parameters such as degree of multipole expansion and the α-criterion of the Barnes–Hut method on parallel efficiency and simulation error. The state of the art in microprocessor and network technology has driven us past the nCUBE2 and CM5. The relative computation to communication speeds are more favorable in many current machines (such as the Cray T3E) than in the nCUBE2 and CM5. This indicates that our formulations will yield even better performance on these machines.

Acknowledgements

This work is sponsored by the by Army Research Office contract DA/DAAH04-95-1-0538 and by Army High Performance Computing Research Center under the auspices of the Department of the Army, Army Research Laboratory cooperative agreement no. DAAH04-95-2-0003/contract no. DAAH04-95-C-0008, the content of which does not necessarily reflect the position or the policy of the government, and no official endorsement should be inferred. This work is also sponsored in part by MSI. Access to computing facilities was provided by Cray Research and by the Pittsburgh Supercomputing Center. Related papers are available via WWW at URL: http://www.cs.umn.edu/users/kumar/papers.html. The authors wish to acknowledge anonymous reviewers whose comments served to improve the quality of this manuscript.

References

[1] J. Barnes, P. Hut, A hierarchical $o(n\log n)$ force calculation algorithm, Nature 324 (1986) .
[2] L. Greengard, V. Rokhlin, A fast algorithm for particle simulations, J. Comp. Phys. 73 (1987) 325–348.
[3] S. Aluru, Greengard's n-body algorithm is not $O(n)$, SIAM J. Sci. Comput. 17 (3) (1996) 773–776.

[4] P.B. Callahan, S.R. Kosaraju, A decomposition of multi-dimensional point-sets with applications to *k*-nearest-neighbors and *n*-body potential fields, Proc. 24th Annual ACM Symp. on Theory of Computing, May 1992, pp. 546–556.
[5] A.W. Appel, An efficient program for many-body simulation, SIAM J. Comput. 6 (1985) .
[6] K. Esselink, The order of Appel's algorithm, Inf. Processing Lett. 41 (1992) 141–147.
[7] L. Greengard, The Rapid Evaluation of Potential Fields in Particle Systems, MIT Press, 1987.
[8] L. Greengard, W. Gropp, A parallel version of the fast multipole method, Parallel Processing for Scientific Computing, 1987, pp. 213–222.
[9] J.A. Board, J.W. Causey, J.F. Leathrum, A. Windemuth, K. Schulten, Accelerated molecular dynamics with the fast multipole algorithm, Chem. Phys. Lett. 198 (1992) 89.
[10] F. Zhao, S.L. Johnsson, The parallel multipole method on the connection machine, SIAM J. Sci. Stat. Comp. 12 (1991) 1420–1437.
[11] J.F. Leathrum, J.A. Board, Mapping the adaptive fast multipole algorithm into mind systems, in: P. Mehrotra, J. Saltz (Eds.), Unstructured Scientific Computation on Scalable Multiprocessors, MIT Press, Cambridge, MA, 1992.
[12] K.E. Schmidt, M.A. Lee, Implementing the fast multipole method in three dimensions, J. Stat. Phys. 63 (1991) 1120.
[13] J. Singh, C. Holt, T. Totsuka, A. Gupta, J. Hennessy, Load balancing and data locality in hierarchical *n*-body methods, J. Parallel Distributed Comput., 1994.
[14] M. Warren, J. Salmon, A parallel hashed oct tree *n*-body algorithm, Proceedings of Supercomputing Conference, 1993.
[15] M. Warren, J. Salmon, Astrophysical *n*-body simulations using hierarchical tree data structures, Proceedings of Supercomputing Conference, 1992.
[16] A. Grama, V. Kumar, A. Sameh, Scalable parallel formulations of the Barnes–Hut method for *n*-body simulations, Supercomputing '94 Proc., 1994.
[17] A. Grama, V. Kumar, A. Sameh, Parallel matrix–vector product using hierarchical methods, in: Proc. Supercomputing '95, San Diego, CA, 1995.
[18] R.J. Anderson, Computer science problems in astrophysical simulation, Silver Jubilee Workshop on Computing and Intelligent Systems, Indian Institute of Science, Tata McGraw-Hill Publishing, New Delhi, 1993.
[19] D.M. Nicol, J.H. Saltz, An analysis of scatter decomposition, IEEE Trans. Comp. 39 (1990) 1337–1345.
[20] V. Kumar, A. Grama, A. Gupta, G. Karypis, Introduction to Parallel Computing: Algorithm Design and Analysis, Benjamin Cummings/Addison Wesley (ISBN 0-8053-3170-0), Redwood City, 1994.
[21] C.P. Kruskal, A. Weiss, Allocating independent subtasks on parallel processors, IEEE Trans. Software Eng. SE-11 (10) (1985) 1001–1016.