A Parallel Hashed Oct-Tree N-Body Algorithm[*]

Technical Paper Submitted to Proceedings of Supercomputing '93

Michael S. Warren[†]
Theoretical Astrophysics
Mail Stop B288
Los Alamos National Laboratory
Los Alamos, NM 87545

John K. Salmon
Physics Department
206-49
California Institute of Technology
Pasadena, CA 91125

March 31, 1993

Abstract

We report on an efficient adaptive N-body method which we have recently designed and implemented. The algorithm computes the forces on an arbitrary distribution of bodies in a time which scales as $N \log N$ with the particle number. The accuracy of the force calculations is analytically bounded, and can be adjusted via a user defined parameter between a few percent relative accuracy, down to machine arithmetic accuracy. Instead of using pointers to indicate the topology of the tree, we identify each possible cell with a key. *The mapping of keys into memory locations is achieved via a hash table. This allows the program to access data in an efficient manner across multiple processors. Performance of the parallel program is measured on the 512 processor Intel Touchstone Delta system. We also comment on a number of wide-ranging applications which can benefit from application of this type of algorithm.*

1 Introduction

N-body simulations have become a fundamental tool in the study of complex physical systems. Starting from a basic physical interaction (e.g., gravitational, Coulombic, Biot-Savart, van der Waals) one can follow the dynamical evolution of a system of N bodies, which represent the phase-space density distribution of the system. N-body simulations are essentially statistical in nature (unless the physical system can be directly modeled by N bodies, as is the case in some molecular dynamics simulations). More bodies implies a more accurate and complete sampling of the phase space, and hence more accurate or complete results. Unfortunately, the minimum accuracy required to model systems of interest often depends on having N be much larger than current computational resources allow.

Because interactions occur between each pair of particles in a N-body simulation, the computational work scales asymptotically as N^2. Much effort has been expended to reduce the computational complexity of such simulations, while retaining acceptable accuracy. One approach is to interpolate the field from a lattice with resolution h, where it can be computed in time $O(h^{-3})$ (using multigrid) or $O(h^{-3} \log h^{-3})$ (using Fourier transforms). The N-dependence of the time complexity then becomes $O(N)$. The drawback to this method is that dynamics on scales comparable to or smaller than h cannot be modeled. In three dimensions, this restricts the dynamic range in length to about one part in a hundred (or perhaps one part in a thousand on a parallel supercomputer), which is insufficient for many calculations.

Another approach is to divide the the interactions into "near" and "far" sets. The forces due to distant particles can then be updated less frequently, or their forces can be ignored completely if one decides the effects of distant particles are negligible. However, this risks significant errors which are hard to analyze. Also, any significant clustering in the system will reduce the efficiency of the method, since a large fraction of the particles end up being in a "near" set.

Over the past several years, a number of methods have been introduced which allow N-body simulations to be performed on arbitrary collections of bodies in time much less than $O(N^2)$, without imposition of a lattice. They all have in common the use of a truncated expansion (e.g., Taylor expansion, Legendre expansion, Poisson expansion) to *approximate* the contribution of many bodies with a single *interaction*. The resulting complexity is usually cited as $O(N)$ or $O(N \log N)$, but a careful analysis of what dependent variables should be held constant (e.g., constant per-

[*] LAUR 93-1224

[†] Department of Physics, University of California, Santa Barbara

timestep error, constant integrated error, constant memory, constant relative error with respect to discreteness noise) often leads to different conclusions about the scaling. In any event, the scaling is a tremendous improvement over $O(N^2)$ and the methods allow accurate computations with vastly larger N.

The basic idea of an N-body algorithm based on a truncated series approximation is to partition an arbitrary collection of bodies in such a manner that the series approximation can be applied to the pieces, while maintaining sufficient accuracy in the force (or other quantity of interest) on each particle. In general, the methods represent a system of N *bodies*[1] in a hierarchical manner by the use of a spatial *tree* data structure. Aggregations of bodies at various levels of detail form the internal nodes of the tree, and are called *cells*. Generally, the expansions have a limited domain of convergence, and even where the infinite expansion converges, the truncated expansion introduces errors of some magnitude. Making a good choice of which cells to interact with, and which to reject as being too inaccurate is critical to the success of these algorithms. The decision is controlled by a function which we shall call the multipole acceptance criterion (MAC). Some of the multipole methods which have been described in the literature are briefly reviewed in the next section.

2 Background

2.1 Multipole Methods

Appel was the first to introduce a multipole method [1]. Appel's method uses a binary tree data structure whose leaves are bodies, and internal nodes represent roughly spherical cells. Some care is taken to construct a "good" set of cells which minimize the higher order multipole moments of the cells. The MAC is based on the size of interacting cells. The method was originally thought to be $O(N \log N)$, but has more recently been shown to be $O(N)$ [2].

The Barnes-Hut (BH) algorithm [3] uses a regular, hierarchical cubical subdivision of space (an oct-tree in three dimensions). A two-dimensional illustration of such a tree (a quad-tree) is show in Fig. 1. Construction of BH trees is much faster than construction of Appel trees. In the BH algorithm, the MAC is controlled by a parameter θ, which requires that the cell size, s, divided by the distance from a particle to the cell center-of-mass be less than θ (which is usually in the range of 0.6–1.0). Cell-cell interactions are not computed, and the method scales as $N \log N$.

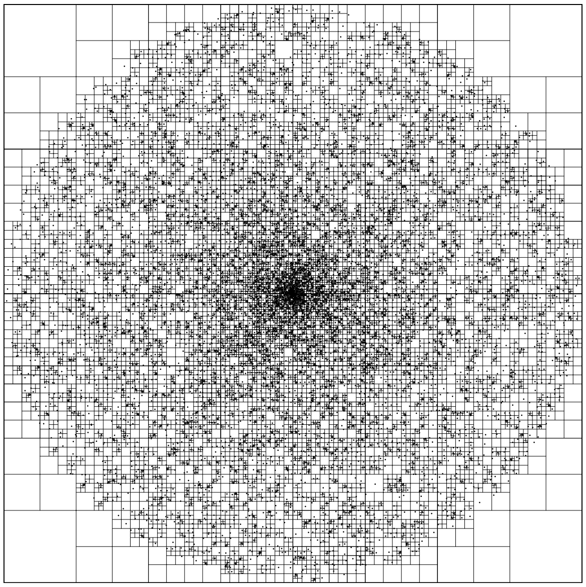

Figure 1: A representation of a regular tree structure in two dimensions (a quad-tree) which contains 10 thousand particles which are centrally clustered.

The fast multipole method (FMM) of Greengard & Rokhlin [4] has achieved the greatest popularity in the broader population of applied mathematicians and computational scientists. It uses high order multipole expansions and interacts fixed sets of cells which fulfill the criterion of being "well-separated." In two dimensions, when used on systems which are not excessively clustered, the FMM is very efficient. It has been implemented on parallel computers [5, 6]. The crossover point (the value of N at which the algorithm becomes faster than a direct N^2 method) with a stringent accuracy is as low as a few hundred particles. On the other hand, implementations of the FMM in three dimensions have not performed as well. Schmidt and Lee have implemented the algorithm in three dimensions, and find a crossover point of about 70 thousand particles [7]. The reason is that the work in the most computationally intensive step scales as p^2 in two dimensions, and p^4 in three dimensions. The major advantage of the FMM over the methods such as that of Barnes & Hut is that the FMM has a well defined worst case error bound. However, this deficiency has been remedied, as is shown in the following section.

2.2 Analytic Error Bounds

Recently, Salmon & Warren have analyzed the performance of the Barnes-Hut algorithm, and have shown that the worst case errors can be quite large (in fact, unbounded) for commonly used values of the opening criterion, θ [8]. Salmon

[1] We refer to both bodies and particles, which should both be understood to be general "atomic" objects which may refer to a mass element, charge, vortex element, panel, or other quantity subject to a multipole approximation.

& Warren have developed a different method for deciding which cells to interact with. By using moments of the mass or charge distribution within each cell, the method achieves far better worst case error behavior, and somewhat better mean error behavior, for the same amount of computational resources.

In addition, the analysis provides a strict error bound which can be applied to any fast multipole method. This error bound is superior to those used previously because it makes use of information about the bodies contained within a cell. This information takes the form of easily computed moments of the mass or charge distribution (strength) within the cell. Computing this information takes place in the tree construction stage, and takes very little time compared with the later phases of the algorithm. The exact form of the error bound is:

$$\Delta a_{(p)}(r) \leq \frac{1}{d^2}\frac{1}{(1-\frac{b_{max}}{d})^2} \left((p+2)\left(\frac{B_{(p+1)}}{d^{p+1}}\right) - (p+1)\left(\frac{B_{(p+2)}}{d^{p+2}}\right)\right). \quad (1)$$

The moments, $B_{(n)}$ are defined as:

$$B_{(n)} = \int_{\mathcal{V}} d^3x\, |\rho(x)|\, |\vec{x}-\vec{r}_0|^n = \sum_{\beta} |m_\beta|\, |\vec{x}_\beta - \vec{r}_0|^n . \quad (2)$$

The scalar $d = |\vec{r}-\vec{r}_0|$ is the distance from the particle position $\vec{r}$ to the center of the multipole expansion, p is the largest term in the multipole expansion, and b_{max} is the maximal distance of particles from the center of the cell. (see Fig. 2). This equation is essentially a precise statement of several common-sense ideas. Interactions are more accurate when:

- The interaction distance is larger (larger d).
- The cell is smaller (smaller b_{max}).
- More terms in the multipole expansion are used (larger p).
- The truncated multipole moments are smaller (smaller $B_{(p+1)}$).

Having a per-interaction error bound is an overwhelming advantage when compared to existing multipole acceptance criteria, which assume a worst-case arrangement of bodies within a cell when bounding the interaction error. The reason is that the worst-case interaction error of an *arbitrary* strength distribution is usually many times larger than the error bound on a *particular* strength distribution. This causes an algorithm which knows nothing about the strength distribution inside a cell to provide *too much* accuracy for most multipole interactions. This accuracy is wasted, however, because of the few multipole interaction errors which do approach the worst-case error bound that are added into and pollute the final result. A data-dependent per-interaction error bound is much less prone to this problem, since the resulting error *bound* is much tighter, even though the actual error in the computation is exactly the same.

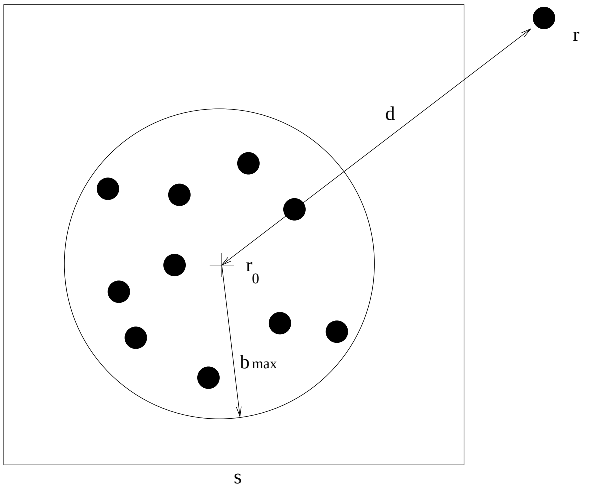

Figure 2: An illustration of the relevant distances used in the error bound equation.

The implementation of an algorithm using a fixed per-interaction error bound poses little difficulty. One may simply solve for r_c in,

$$\Delta a_{(p)}(r_c) \leq \Delta_{interaction}, \quad (3)$$

where $\Delta_{interaction}$ is a user-specified absolute error tolerance. Then, r_c defines the smallest interaction distance allowed for each cell in the system. For the case of $p = 1$, the critical radius can be analytically derived from Eq. 1 if we use the fact that $B_3 \geq 0$:

$$r_c \geq \frac{b_{max}}{2} + \sqrt{\frac{b_{max}^2}{4} + \sqrt{\frac{3B_2}{\Delta_{interaction}}}}. \quad (4)$$

B_2 is simply the trace of the quadrupole moment tensor. In more general cases (using a better bound on B_3, or with $p > 1$), r_c can be computed from the error bound equation (Eq. 1) using Newton's method. The overall computational expense of calculating r_c is small, since it need only be calculated once for each cell. Furthermore, Newton's method need not be iterated to high accuracy. The MAC then becomes $d > r_c$ for each displacement d and critical radius r_c (Fig. 3). This is computationally very similar to the Barnes-Hut opening criterion, where instead of using a fixed box size, s, we use the distance r_c, derived from the contents of the cell and the error tolerance. Thus, our data dependent MAC may replace the MAC in existing algorithms with minimal additional coding.

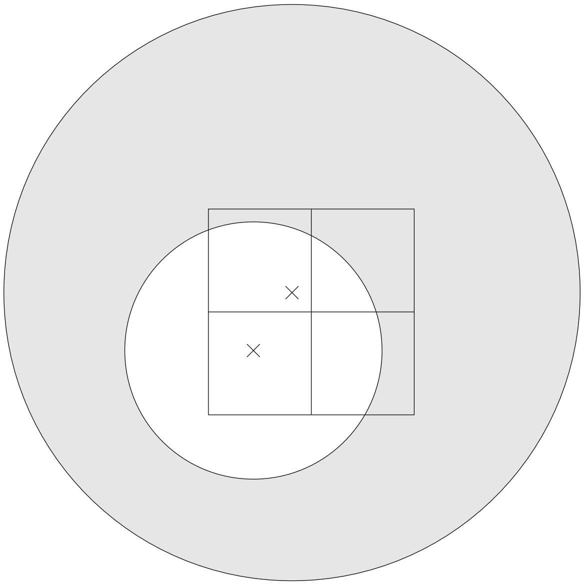

Figure 3: Each cell has its own critical radius. The critical radii of a cell and one of its daughters are shown here as circles. For the specified accuracy, a particle must lie outside the critical radius of a cell. The shaded region shows the spatial domain of all particles which would interact with the lower left daughter cell. Those particles outside the shaded region would interact with the parent cell, and those within the unshaded region would interact with even smaller cells inside the daughter cell.

3 Computational Approach

Parallel treecodes for distributed memory machines are discussed in [9, 10], and their application to the analysis of galaxy formation may be found in [11, 12]. Further analysis and extensions of the computational methods may be found in [13, 14, 15]. The MAC described above is problematical for these previous methods because the parallel algorithm requires determination of locally essential data before the tree traversal begins. With the data-dependent MAC it is difficult to pre-determine which non-local cells are required in advance of the traversal stage. The problem becomes particularly acute if one wishes to impose error tolerances which vary from particle to particle.

It is for this reason that the algorithm described here was developed. It does not rely on the ability to identify *a priori* locally essential data; instead it provides a mechanism to retrieve non-local data as it is needed during the tree traversal. The decision to abandon our previous parallel N-body algorithm was also motivated by the desire to produce a more “friendly” code, with which a variety of research could be performed in computational science as well as physics. The old code, which was the result of porting a previously existing sequential algorithm, was a maze of complications, brought about by the haphazard addition of pieces over several years. We took full advantage of the opportunity to start over with a clean slate, with the additional benefit of several years of hindsight and experience.

When one considers what additional operations are necessary when dealing with a tree structure distributed over many processors, it is clear that retrieval of particular cells required by one processor from another is a very common operation. When using a conventional tree structure, the pointers in a parent cell in one processor must be somehow translated into a valid reference to daughter cells in another processor. This required translation led us to the conclusion that pointers are not the proper way to represent a distributed tree data structure (at least without significant hardware and operating system support for such operations).

Instead of using pointers to describe the topology of a tree, we use keys and a hash table. We begin by identifying each possible cell with a *key*. By performing simple bit arithmetic on a key, we are able to produce the keys of daughter or parent cells. The tree topology is represented implicitly in the mapping of the cell spatial locations and levels into the keys. The translation of keys into memory locations where cell data is stored is achieved via hash table lookup. Thus, given a key, the corresponding data can be rapidly retrieved. This scheme also provides a uniform addressing mechanism to retrieve data which is in another processor. This is the basis of the hashed oct-tree (HOT) method.

3.1 Key construction and the Hashing Function

We define a key as the result of a map of d floating point numbers (body coordinates in d-dimensional space) into a single set of bits (which is most conveniently represented as a vector of integers). The mapping function consists of translating the floating point numbers into integers, and then interleaving the bits of the d integers into a single key (Fig. 4). Note that we place no restriction on the dimension of the space, although we are physically motivated to pay particular attention to the case of $d = 3$. In this case, the key derived from 3 single precision floating point numbers fits nicely into a single 64 bit integer or a pair of 32 bit integers.

Apart from the trivial choice of origin and coordinate system, this is identical to Morton ordering (also called Z or N ordering, see Chapter 1 of [16] and references therein,

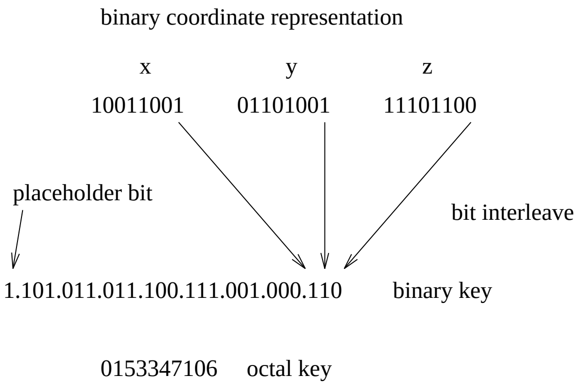

Figure 4: An illustration of the key mapping. Bits of the coordinates are interleaved and a place-holder bit is prepended to the most significant bit. In this example, the 8-bit x, y and z values are mapped to a 25-bit key.

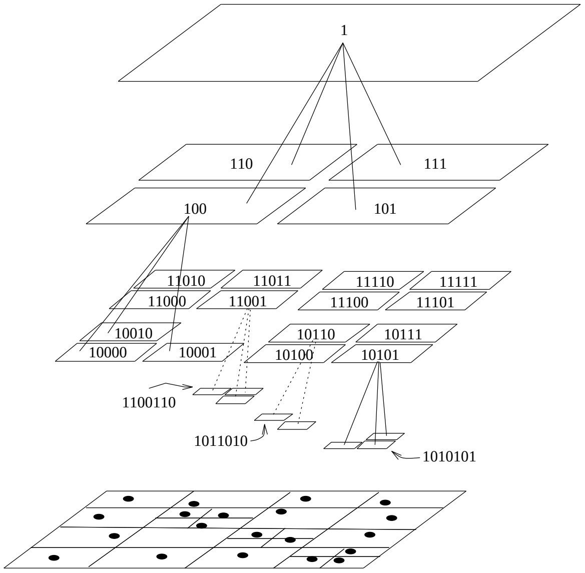

Figure 5: A quad-tree shown along with the binary key coordinates of the nodes. At the bottom is a "flat" representation of the tree topology induced by the 20 particles. The rest of the figure demonstrates the relation of the key coordinates at each level to the tree topology. Many of the links from parent to daughter cells are omitted for clarity.

and also [17]). This function maps each body in the system to a unique key. We also wish to represent nodes of the tree using this same type of key. In order to distinguish the higher level internal nodes of the tree from the lowest level body nodes, we prepend an additional 1-bit to the most significant bit of every key (the place-holder bit). We may then represent all higher level nodes in the tree in the same key space. Without the place-holder bit, there would be an ambiguity amongst keys whose most significant bits are all zeroes. The root node is represented by the key 1. A two-dimensional representation of such a tree is shown in Fig. 5.

In general, each key corresponds to some composite data describing the physical data inside the domain of a cell (the mass and center-of-mass coordinates, for example). To map the key to the memory location holding this data, a hash table is used. A table with a length much smaller than the possible number of keys is used, with a hashing function to map the k-bit key to the h-bit long hash address. We use a very simple hashing function, which is to AND the key with the bit-mask $2^h - 1$, which selects the least significant h bits.

Collisions in the hash table are resolved via a linked list (chaining) [18]. The incidence of collisions could degrade performance a great deal. Our hashing scheme uses the simplest possible function; a one instruction AND. However, it is really the map of floating point coordinates into the key that performs what one usually would consider "hashing." The structure of the hierarchical key space and selection of the least significant bits of the key performs extraordinarily well in reducing the incidence of collisions. For the set of all keys which contain fewer significant bits than the hash mask, the hashing function is "perfect." This set of keys represents the upper levels of the tree, which tend to be accessed the most often. At lower levels of the tree (where the number of bits in a key exceeds the length of the hash mask), distinct keys can result in the same hash address (a collision). However, the map of coordinates into the keys keeps these keys spatially separated. On a parallel machine, many of the keys which would results in collisions become distributed to different processors.

The key space is very convenient for tree traversals. In order to find daughter nodes, the parent key is left-shifted by d bits, and the result is added (or equivalently OR'ed) to daughter numbers from 0 to $2^d - 1$. Also, the key retrieval mechanism is much more flexible in terms of the kinds of accesses which are allowed. If we wish to find a particular node of a tree in which pointers are used to traverse the tree, we must start at the root of the tree, and traverse until we find the desired node (which takes of order $\log N$ operations). On the other hand, a key provides immediate ($O(1)$) access to any object in the tree.

An entry in the hash table (an hcell) consists of a pointer to the cell or body data, a pointer to a linked list which resolves collisions, the key, and various flags which describe properties of the hcell and its corresponding cell. In order to optimize certain tree traversal operations, we also store in each hcell 2^d bits that describe which daughters of the cell actually exist. This redundant information allows us to avoid using hash-table lookup functions to search for cells which don't exist. The data structure is detailed in Fig. 6.

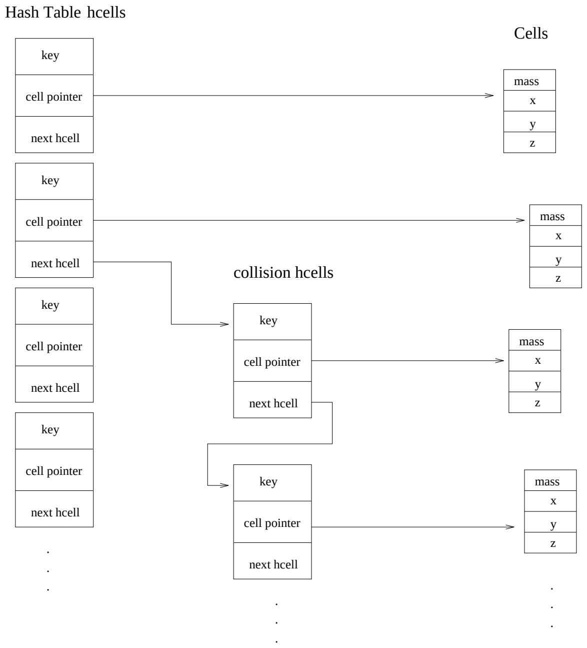

Figure 6: An illustration of the data structures used in the hashed tree. This shows the fixed number (2^h) of `hcells` in the hash table, the extra hcells created to resolve collisions, and the `cells` which contain the physical data derived from the particles within the cell.

The use of a hash table offers several important advantages. First, the access to data takes place in a manner which is easily generalized to a global accessing scheme implementable on a message passing architecture. That is, non-local data may be accessed by requesting a key, which is a uniform addressing scheme, regardless of which processor the data is contained within. This type of addressing is not possible with normal pointers on a distributed memory machine. We can also use the hash table to implement various mechanism for caching non-local data and improving memory system performance, as is described in § 3.7.

3.2 Tree Construction

The higher level nodes in the tree can be constructed in a variety of ways. The simplest is analogous to that which was described in [3]. Each particle is loaded into the tree by starting at the root, and traversing the partially constructed tree. When two particles fall within the same leaf node, the leaf is converted to a cell which is entered into the hash table, and new leaves are constructed one level deeper in the tree to hold each of the particles. This takes $O(\log N)$ steps per particle insertion. After the topology of the tree has been constructed, the contents (mass, charge, moments, etc.) of each cell may be initialized by a post-order tree traversal.

A faster method is possible by taking advantage of the spatial ordering implied in the key map. We first sort the body keys, and then consider the bodies in this list in order. As bodies are inserted into the tree, we start the traversal at the location of the last node created (rather than at the root). With this scheme, the average body insertion requires $O(1)$ time. We still require $O(N \log N)$ time to sort the list in the first place, but keeping the body list sorted will facilitate our parallel data decomposition as well. Furthermore, it is also usually the case that numerous tree constructions will take place in the course of a simulation. Since the positions of the bodies in the key space typically do not change a great deal between timesteps, one can take advantage of a more efficient $O(N)$ insertion sort [18], of the almost-sorted data after the first timestep.

3.3 Parallel Data Decomposition

The parallel data decomposition is critical to the performance of a parallel algorithm. A method which may be conceptually simple and easy to program may result in load imbalance which is unacceptable. A method which attempts to balance the work precisely may take so long that performance of the overall application suffers.

We have implemented a method which can rapidly domain decompose a d-dimensional set of particles into load balanced spatial groups which represent the domain of each processor. We take advantage of the properties of the mapping of spatial coordinates to keys to produce a "good" domain decomposition. The idea is to simply cut the one-dimensional list of sorted body key ordinates (see Fig. 7) into N_p (number of processors) equal pieces, weighted by the amount of work corresponding to each body. The work for each body is readily approximated by counting the number of interactions the body was involved in on the previous timestep. This results in a spatially adaptive decomposition, which gives each processor an equal amount of work. Additionally, the method keeps particles spatially grouped, which is very important for the efficiency of the traversal stage of the algorithm, since the amount of non-local data needed is roughly proportional to the surface area of the processor domain. An illustration of this method on a two-dimensional set of particles is illustrated in Fig. 9 for a highly clustered set of particles (that which was shown in Fig. 1) with $N_p = 16$. One source of inefficiency in the Morton ordered decomposition is that a processor domain can span one of the spatial discontinuities. A possible solution is to use Peano-Hilbert ordering for the domain decomposition, which is shown in Fig. 8.

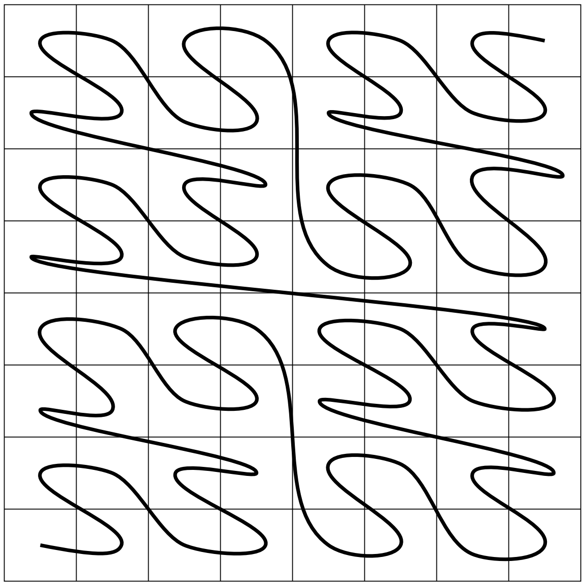

Figure 7: The path indicates the one-dimensional symmetric self-similar path which is induced by the map of interleaved bits (Morton order). The domain decomposition is achieved by cutting the one-dimensional list into N_p pieces.

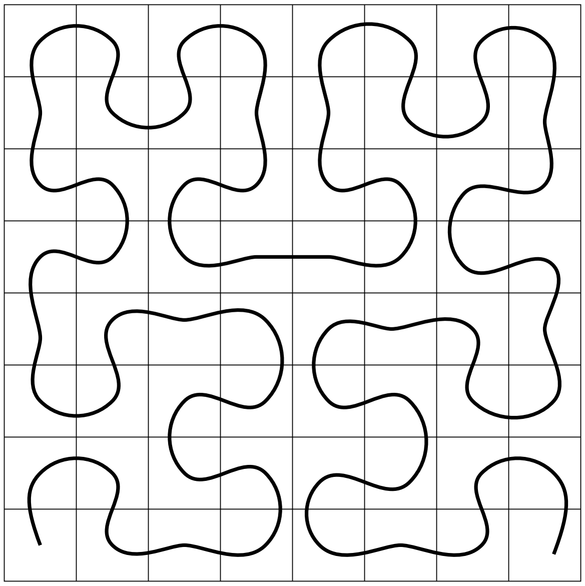

Figure 8: A non-symmetric path that does not contain discontinuities (Peano-Hilbert order) which should produce a better decomposition, but which has not been implemented yet.

3.4 Parallel Tree Construction

After the domain decomposition, each processor has a disjoint set of bodies. The initial stage in parallel tree building is the construction of a tree made of the local bodies. A special case occurs at each processor boundary in the one-dimensional sorted key list, where the terminal bodies from adjacent processors could lie in the same cell. This is taken care of by sending a copy of each boundary body to the adjacent processor, which allows the construction of the proper tree nodes. Then, copies of *branch* nodes from each processor are shared among all processors. This stage is made considerably easier and faster since the domain decomposition is intimately related to the tree topology (unlike the orthogonal recursive bisection method used in our previous code [10]). The branches make up a complete set of cells which represent the entire processor domain at the coarsest level possible. These branch cells are then globally communicated among the processors. All processors can then "fill in" the missing top of the tree down to the branch cells. The address of the processor which owns each branch cell is passed to the destination processor, so the hcell created is marked with its origin. A traversal routine can then immediately determine which processor to request data from when it needs access to the daughters of a branch cell. The daughters received from other processors are also marked in the same fashion. We have also tried implementing the branch communication step in a more computationally clever manner which does not globally concatenate the branches, but its complexity has tended to outweigh its benefit. This does not rule out the possibility of finding a better method for this stage of the algorithm, however.

3.5 Tree Traversal

A tree traversal routine may be cast in recursive form in a very few lines of C code:

```
Traverse(Key_t key, int (*MAC)(hcell *),
          void (*postf)(hcell *))
{
    hcell *pp;
    unsigned int child;

    if ((pp=Find(key)) && MAC(pp)) return;
    key = KeyLshift(key, NDIM);
    for (child = 0; child < (1<<NDIM); child++)
      Traverse(KeyOrInt(key, child), MAC, postf);
    postf(pp);
}
```

This code applies an arbitrary MAC to determine whether to continue traversing the children of a cell. If the children are traversed, than another function, `postf`, is called upon completion of the descendants. By appropriate choice of the `MAC` and `postf` one can execute pre-order or post-

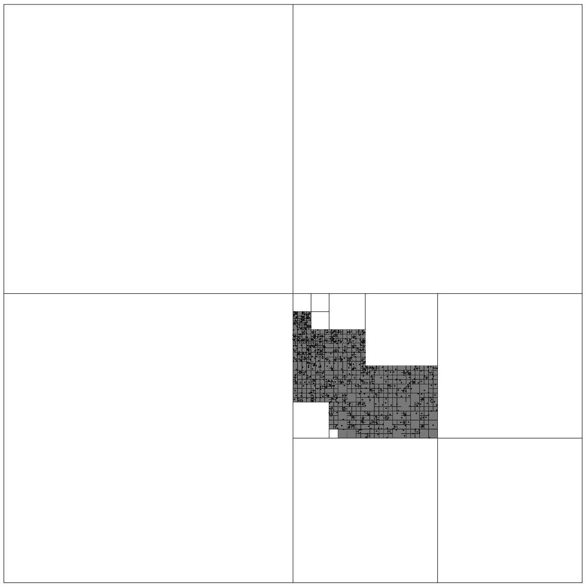

Figure 9: A processor domain for one of 16 processors in a data decomposition for the clustered system of bodies shown in Fig. 1. The domain shown is a result of the decomposition strategy outlined in the text.

order traversals with or without complex pruning strategies (i.e., multipole acceptability criteria).

On a parallel machine, one may add additional functionality to the `Find` function, in order to handle cases where the requested node is in the memory of another processor. The additional code would request non-local data, wait to receive it, and insert it into the tree. This allows the same traversal code fragment to work without further modification on a distributed memory computer. However, the performance of such an approach is bound to be dismal. Each request of non-local data is subject to the full interprocessor communication latency. Computation stalls while waiting for the requested data to arrive.

It is possible to recast the traversal function in a form which allows the entire *context* of the traversal to be stored. In this case, when a request for non-local data is encountered, the request is buffered, and the computation may proceed. Almost all of the latency for non-local data requests may be hidden, by trading communication latency for a smaller amount of complexity overhead.

The traversal method we have chosen is breadth-first list based scheme. It does not use recursion, and has several useful properties. We shall discuss the plain sequential method first, and then show the additions to allow efficient traversals on a parallel machine.

The input to the list-based traversal is a *walk list* of hcell nodes. On the first pass, the walk list contains only the root hcell. Each daughter of the input walk list nodes is tested against the MAC. If it passes the MAC, the corresponding cell data is placed on the *interaction list*. If a daughter fails the MAC, it is placed on the output walk list. After the entire input list is processed the output walk list is copied to the walk list and the process iterates. The process terminates when there are no nodes remaining on the walk list. This method has an advantage over a recursive traversal in that there is an opportunity to do some vectorization of the intermediate traversal steps, since there are generally a fair number of nodes which are being tested at a time. It also results in a final interaction list which can be passed to a fully vectorized force calculation routine. The details are too intricate to allow us to present real `C` code, so we present the algorithm in pseudocode instead:

```
ListTraverse((*MAC)(hcell *))
{
  copy root to walk_list;
  while (!Empty(walk_list)) {
    for (each item on walk_list) {
      for (each daughter of item) {
        if (MAC(daughter))
          copy daughter to interact_list;
        else
          copy daughter to output_walk_list;
      }
    }
    walk_list = output_walk_list;
  }
}
```

When the traversal is complete, the `interact_list` contains a vector of items that must undergo interactions (according to the particular MAC). The interactions themselves may be computed separately, so that code may be vectorized and optimized independently of the tree traversal method.

3.6 A Latency Hiding Tree Traversal

On a parallel machine, the traversal will encounter hcells for which the daughters are not present in local memory. In this case we add some additional lists which allow computation to proceed, while the evaluation of the non-local data is deferred to some later time. Each hcell is labeled with a `HERE` bit. This bit is set if the daughters of the hcell are present in local memory. This bit is tested in the traversal before the attempt to find the daughters. If the `HERE` bit is not set, the key and the source processor address (which is contained in the hcell) are placed on the *request* list, and another copy of the key is placed on a *defer* list. We additionally set a `REQUESTED` bit in the hcell, to prevent additional requests for the same data. This allows processing to continue on the hcells in the input walk list. As the traversal proceeds, additional requests will occur, until a final state is reached, where as much progress as possible has been made on the given traversal (using only data in local memory). In this state, there are a number of keys and processor addresses in the request list, and an equal number of keys in the defer list, which require non-local data to be received before the traversal may continue.

The request list is periodically translated into a series of interprocessor messages which contain requests for data. Upon receipt of such a message, the appropriate hcells are packaged into a reply, and the answer is returned via a second interprocessor message. When a reply is received, an appropriate entry is made in the hash table, and subsequent `Find` requests will return the data. It is possible to implement this request/reply protocol either loosely synchronously or asynchronously. The decision is governed by the level of support and relative performance offered by the hardware and operating system.

Upon receipt of some replies (it is not necessary to wait for all replies to arrive), the defer list can be renamed as the `walk_list`, and the traversal can be restarted with the newly arrived data. Alternatively, one can begin an entirely separate traversal to compute, e.g., the force on another particle. With appropriate bookkeeping and a modest amount of memory, one can tolerate very long latencies by implementing a circular queue of active traversals (with a shared request list). We have used a circular queue with 30 active traversals, so that after 30 traversals have been deferred, we restart the first traversal by copying its defer list to its walk list. The requested data has usually arrived in the interim.

3.7 Memory Hierarchy and Access Patterns

Treecodes place heavy demands on the memory subsystems of modern computers. The quasi-random access to widely separated memory locations during the tree traversal receives little benefit from a small on-chip cache, and can in fact overwhelm the translation look-a-side buffer (TLB) on microprocessors similar to the `i860`. This results in very poor performance from algorithms which have been designed without consideration of memory bandwidth and the limitations inherent in memory hierarchies which are intended to function with DRAM significantly slower than the processor cycle time.

Since the tree traversal is so stressful to most memory architectures, we have arranged the order of computation to take advantage of the underlying structure of the algorithm, which helps encourage a more orderly memory access pattern. A useful property of treecode algorithms is that particles which are spatially near each other tend to have very similar cell interaction lists. By updating the particles in an order which takes advantage of their spatial proximity, we can reduce the number of memory accesses which miss the cache and TLB. A convenient and efficient ordering once again uses the same sorted key list used in the tree construction. By updating particles in the order defined by the key map (Fig. 7), we achieve this goal.

An additional technique to improve memory access speed is through the rearrangement of data in the linked list of collisions in the hash table. By moving data which has been recently accessed to the top of the linked list, it is possible to create a “virtual cache” by keeping often used data in the contiguous memory locations making up the hash table. This also allows one to obtain good performance with a hash table much smaller than one would naively expect.

The more extended memory hierarchy in a distributed memory parallel computer (possibly with virtual memory on each node) can benefit from this scheme as well. We wish to keep things for as long as possible in the fastest level of the hierarchy that includes registers, cache, local memory, other processors memory, and virtual memory. We could extend the “virtual cache” model even further, by erasing data which has come from another processor which has not been used recently. Although this has not been implemented, we expect that it will allow significantly larger simulations to take place, since the majority of the memory used consists of copies of cells from other processors.

4 Performance

Here we provide timings for the various stages of the algorithm on the 512 processor Intel Touchstone Delta installed at Caltech. The timings listed are from an 8.8 million particle production run simulation involving the formation of structure in a cold dark matter Universe [12]. During the initial stages of the calculation, the particles are spread uniformly throughout the spherical computational volume. We set an absolute error bound on each partial acceleration of 10^{-3} times the mean acceleration in the system. This results in 2.2×10^{10} interactions per timestep in the initial unclustered system. The timing breakdown is as follows:

computation stage	*time* (sec)
Domain Decomposition	7
Tree Build	7
Tree Traversal	33
Data Communication During Traversal	6
Force Evaluation	54
Load Imbalance	7
Total (5.8 Gflops)	114

At later stages of the calculation the system becomes extremely clustered (the density in large clusters of particles is typically 10^6 times the mean density). The number of interactions required to maintain the same accuracy grows moderately as the system evolves. At a slightly increased error bound of 4×10^{-3}, the number of interactions in the clustered system is 2.6×10^{10} per timestep.

computation stage	*time* (sec)
Domain Decomposition	19
Tree Build	10
Tree Traversal	55
Data Communication during traversal	4
Force Evaluation	60
Load Imbalance	12
Total (4.9 Gflops)	160

It is evident that the initial domain decomposition and tree building stages take a relatively larger fraction of the time in this case. The reason is that in order to load balance the force calculation, some processors have nearly three times as many particles as the mean value, and over ten times as many particles as the processor with the fewest. The load balancing scheme currently attempts to load balance only the work involved in force evaluation and tree traversal, so the initial domain decomposition and tree construction work (which scales closely with the particle number within the processor) becomes imbalanced.

Note that roughly 50% of the execution time is spent in the force calculation subroutine. This routine consists of a few tens of lines of code, so it makes sense to obtain the maximum possible performance through careful tuning. For the Delta's `i860` microprocessor we used hand coded assembly language to keep the three-stage pipeline fully filled, which results in a speed of 28 Mflops per processing node in this routine.

If we count only the floating point operations performed in the force calculation routine as "useful work" (30 flops per interaction) the overall speed of the code is about 5–6 Gflops. However, this number is in a sense unfair to the overall algorithm, since the majority of the code is not involved in floating point operations at all, but with tree traversal and data structure manipulation. The integer arithmetic and addressing speed of the processor are as important as the floating point performance. We hope that in the future, evaluation of processors does not become overbalanced toward better floating point speed at the expense of integer arithmetic and memory bandwidth, as this code is a good example of why a balanced processor architecture is necessary for good overall performance.

5 Multi-purpose Applications

Problems of current interest in a wide variety of areas rely heavily on N-body and/or fast multipole methods. Accelerator beam dynamics, astrophysics (galaxy formation, large-scale structure), computational biology (protein folding), chemistry (molecular structure and thermodynamics), electromagnetic scattering, fluid mechanics (vortex method, panel method), molecular dynamics, and plasma physics, to name those we are familiar with, but there are certainly more. In some of these areas, N^2 algorithms are still the most often used, due to their simplicity. However, as problems grow larger, the use of fast methods becomes a necessity. Indeed, in the case of problems such as electromagnetic scattering, a fast multipole method reduces the operation count for solving the second-kind integral equation from $O(N^3)$ for Gaussian elimination to $O(N^{4/3})$ per conjugate-gradient iteration [19]. Such a vast improvement allows one to contemplate problems which were heretofore simply impossible. Alternatively, one can use a workstation to solve problems that had previously been in the sole domain of large supercomputers.

We have spent substantial effort in this code keeping the data structures and functions required by the "application" away from those of the "tree". With suitable abstractions and ruthless segregation, we have met with some success in this area. We currently have a number of physics applications which share the same tree code. In general, the addition of another application only requires the definition of a data structure, and additional code is required only with respect to functions which are physics related (e.g., the force calculation).

We have described the application of our code to gravitational N-body problems above. The code has also been indispensable in performing statistical analyses and data processing on the end result of our N-body calculations, since their size prohibits analysis on anything but a parallel supercomputer. The code also has a module which can perform three-dimensional compressible fluid dynamics using smoothed particle hydrodynamics (with or without gravity). Also, of special note, Winckelmans has used the code to implement a vortex particle method. It is a simple matter to use the same program to do physics involving other force laws. Apart from the definition of a data structure and modification of the basic force calculation routine, one only need derive the appropriate MAC using the method described in Salmon & Warren [8].

6 Future Improvements

The code described here is by no means a "final" version. The implementation has been explicitly designed to easily allow experimentation, and inclusion of new ideas which we find useful. It is perhaps unique in that it is serving double duty as a high performance production code to study the process of galaxy formation, as well as a testbed to investigate multipole algorithms.

Additions to the underlying method which we expect will improve its performance even further include the addition of cell-cell evaluations (similar to those used in the fast multipole method) and the ability to evolve each particle with an independent timestep (which improves performance significantly in systems where the timescale varies greatly). We expect that the expression of the algorithm in the C++ language will produce a more friendly program by taking advantage of the features of the language such as data abstraction and operator overloading. In the near future we expect to port and evaluate our code on the latest parallel machines such as the Connection Machine 5 and the Intel Paragon. We expect that the majority of effort needed to port this code to the CM-5 machine will be CDPEAC coding of the inner loop of the force-evaluation routine. The remainder of the code is sufficiently portable to require only minor modification (we hope).

7 Conclusion

In an overall view of this algorithm, we feel that these general items deserve special attention:

- The fundamental ideas in this algorithm are, for the most part, standard tools of computer science (key mapping, hashing, sorting). We have shown that in combination, they form the basis of a clean and efficient parallel algorithm. This type of algorithm does not evolve from a sequential method. It requires starting anew, without the prejudices inherent in a program (or programmer) accustomed to using a single processor.

- The raw computing speed of the code on an extremely irregular, dynamically changing set of particles which require global data for their update, using a large number of processors (512), is comparable with the performance quoted for much more regular static problems, which are sometimes identified as the only type of "scalable" algorithms which obtain good performance on parallel machines. We hope we have convinced the reader that even difficult irregular problems are amenable to parallel computation.

We expect that algorithms such as that described here, coupled with the extraordinary increase in computational power expected in the coming years, will play a major part in the process of understanding complex physical systems.

Acknowledgments

We thank Sanjay Ranka for pointing out the utility of Peano-Hilbert ordering. We thank the CSCC and the CCSF for providing computational resources. JS wishes to acknowledge support from the Advanced Computing Division of the NSF, as well as the CRPC. MSW wishes to acknowledge support from IGPP and AFOSR. This research was supported in part by a grant from NASA under the HPCC program. This research was performed in part using the Intel Touchstone Delta System operated by Caltech on behalf of the Concurrent Supercomputing Consortium.

References

[1] A. W. Appel, "An efficient program for many-body simulation," *SIAM J. Computing*, vol. 6, p. 85, 1985.

[2] K. Esselink, "The order of Appel's algorithm," *Information Processing Let.*, vol. 41, pp. 141–147, 1992.

[3] J. Barnes and P. Hut, "A hierarchical O(NlogN) force-calculation algorithm," *Nature*, vol. 324, p. 446, 1986.

[4] L. Greengard and V. Rokhlin, "A fast algorithm for particle simulations," *J. Comp. Phys.*, vol. 73, pp. 325–348, 1987.

[5] L. Greengard and W. D. Gropp, "A parallel version of the fast multipole method," *Computers Math. Applic*, vol. 20, no. 7, pp. 63–71, 1990.

[6] F. Zhao and S. L. Johnsson, "The parallel multipole method on the connection machine," *SIAM J. Sci. Stat. Comp.*, vol. 12, pp. 1420–1437, Nov. 1991.

[7] K. E. Schmidt and M. A. Lee, "Implementing the fast multipole method in three dimensions," *J. Stat. Phys.*, vol. 63, no. 5/6, pp. 1223–1235, 1991.

[8] J. K. Salmon and M. S. Warren, "Skeletons from the treecode closet," *J. Comp. Phys.*, 1992. (in press).

[9] J. K. Salmon, *Parallel Hierarchical N-body Methods*. PhD thesis, California Institute of Technology, 1990.

[10] M. S. Warren and J. K. Salmon, "Astrophysical N-body simulations using hierarchical tree data structures," in *Supercomputing '92*, IEEE Comp. Soc., 1992. (1992 Gordon Bell Prize winner).

[11] M. S. Warren, P. J. Quinn, J. K. Salmon, and W. H. Zurek, "Dark halos formed via dissipationless collapse: I. Shapes and alignment of angular momentum," *Ap. J.*, vol. 399, pp. 405–425, 1992.

[12] W. H. Zurek, P. J. Quinn, J. K. Salmon, and M. S. Warren, "Formation of structure in a CDM universe: Correlations in position and velocity," *Ap. J.*, 1993. (in preparation).

[13] J. P. Singh, J. L. Hennessy, and A. Gupta, "Implications of hierarchical N-body techniques for multiprocessor architectures," Tech. Rep. CSL-TR-92-506, Stanford University, 1992.

[14] J. P. Singh, C. Holt, T. Totsuka, A. Gupta, and J. L. Hennessy, "Load balancing and data locality in hierarchical N-body methods," *Journal of Parallel and Distributed Computing*, 1992. (in press).

[15] S. Bhatt, M. Chen, C. Y. Lin, and P. Liu, “Abstractions for parallel N-body simulations,” Tech. Rep. DCS/TR-895, Yale University, 1992.

[16] H. Samet, *Design and Analysis of Spatial Data Structures*. Reading, MA: Addison-Wesley, 1990.

[17] J. E. Barnes, “An efficient N-body algorithm for a fine-grain parallel computer,” in *The Use of Supercomputers in Stellar Dynamics* (P. Hut and S. McMillan, eds.), (New York), pp. 175–180, Springer-Verlag, 1986.

[18] D. E. Knuth, *The Art of Computer Programming: Sorting and Searching*, vol. 3. Reading, Mass.: Addison Wesley, 1973.

[19] N. Engheta, W. D. Murphy, V. Rokhlin, and M. S. Vassiliou, “The fast multipole method (FMM) for electromagnetic scattering problems,” *IEEE Transactions on Antennas and Propagation*, vol. 40, no. 6, pp. 634–642, 1992.