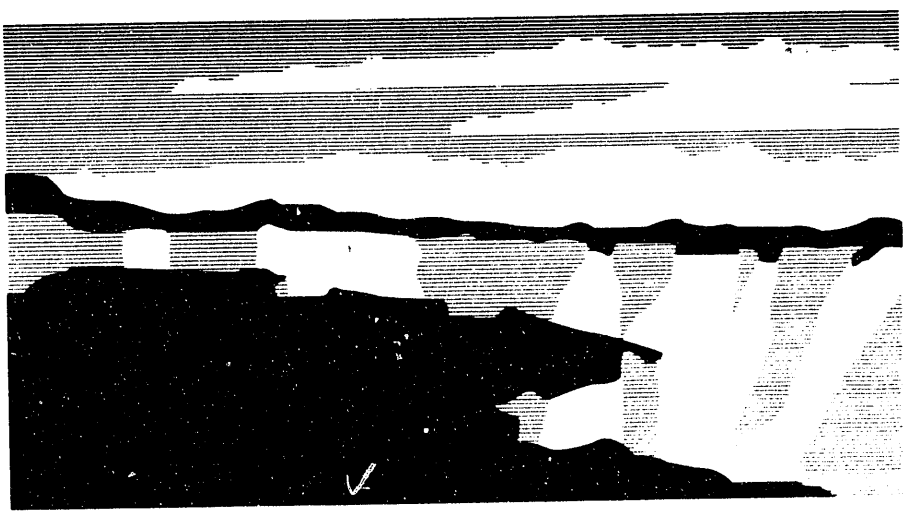
Conf-931115--3

LA-UR- 93-1244

Title: A PARALLEL HASHED OCT-TREE N-BODY ALGORITHM

Author(s): Michael S. Warren, T-6
John K. Salmon, Caltech

Submitted to: Proceedings of Supercompuitng '93
Nov. 11-15, 1993, Portland, Washington



Los Alamos
NATIONAL LABORATORY

3/29/93

A Parallel Hashed Oct-Tree N-Body Algorithm

Michael S. Warren[1]

Theoretical Astrophysics

Mail Stop B288

Los Alamos Natl. Lab.

Los Alamos, NM 87545

msw@eagle.lanl.gov

(505) 665-5023

FAX: (505) 665-4055

John K. Salmon

Physics Department

206-49

Caltech

Pasadena, CA 91125

johns@ccsf.caltech.edu

(818) 356-2907

FAX: (818) 584-5917

March 29, 1993

[1]*presenting author*, Department of Physics, University of California, Santa Barbara

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 us to access data in an efficient manner across multiple processors using a virtual shared memory model. 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 or van der Waals force) one can follow the dynamical evolution of a system of N particles, 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 particles, as is the case in some molecular dynamics simulations). More bodies mean 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 in trying to reduce the computational complexity of such simulations, while retaining acceptable accuracy. One approach is to discretize the phase-space on a lattice, and transform the force field using a fast Fourier transform. This converts the complexity to $O(N \log N)$. The drawback to this method is that dynamics on scales smaller than the lattice 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 unacceptable 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, 6this 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 simulations to be performed rapidly, with much better accuracy than the methods mentioned above. They all have in common the use of the *multipole approximation* to reduce the complexity to $O(N)$ or $O(N \log N)$. To compute the gravitational force on an apple, one generally treats the Earth as a point mass, rather than a distributed collection of atoms. Multipole approximations of this type are not always valid. In fact, they don't converge if the point at which the force is desired is inside a sphere that circumscribes the extended body. Generally, the approximation is better if the measurement point is far from the other masses. The scale by which one should judge "near" and "far" is simply the diameter of the sphere that encloses the mass points.

The basic idea of an N-body algorithm based on the multipole approximation is to divide up an arbitrary collection of bodies in such a manner that the multipole approximation can be applied to the pieces, while maintaining sufficient accuracy of the force on each particle. In general, the methods represent a system of N *bodies* in a hierarchical manner by the use of a *tree* data structure. Aggregations of bodies at various levels of detail form the internal nodes of the tree, and are called *cells*. The decision of which interactions to calculate between the cells and bodies (which affects the speed and accuracy of a method) is controlled by a function which we shall call the multipole acceptance criterion (MAC). 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 $O(N \log N)$, 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 quadtree) is show in Fig. 1. Construction of such trees are much faster than methods such as Appel's. 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. Cell-cell interactions are not computed, and the method scales as $N \log N$.

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. A 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.

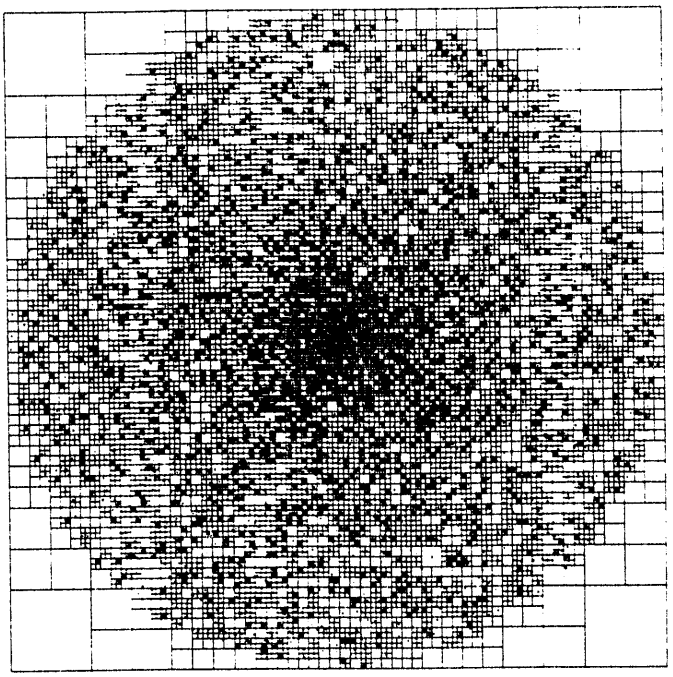


Figure 1: A representation of a regular tree structure in two-dimensions which contains 10 thousand particles which are centrally clustered.

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

An added benefit of the analysis is the derivation of a strict error bound which can be applied to any fast multipole method. This error bound is superior to those used previously in an important way. The method uses information about the bodies contained within a cell to produce a "better" error bound. 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:

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

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

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

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 within the cell from the center. (see Fig. 2). This equation is basically an exact 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 interactoin error of an arbitrary strength distribution is many times larger than the mean error. 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 interactions errors which do approach the error bound that are added into the final result. A data-dependent per-interaction error bound does not suffer from this problem, and the mean error is only a factor of 3 or so less than the worst case error. Thus, our "better" error bound is not better in the sense of being more strict, but a better *estimate* of the error, while maintaining the same worst-case error bound.

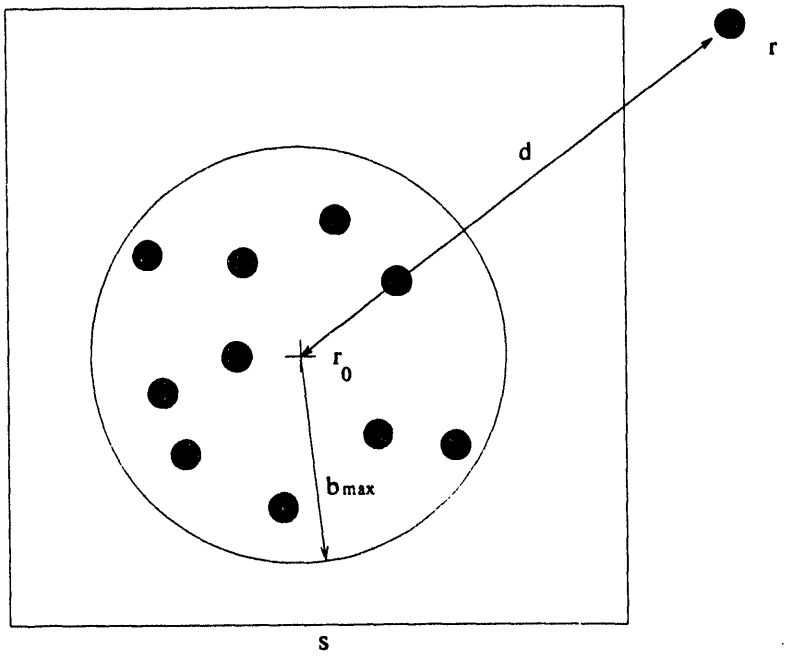


Figure 2: An illustration of the relevant distances used in the error bound equation.

The implementation of an algorithm using a fixed per-interaction error bound poses little difficulty. One may simply transform the error bound equation into a form from which may be derived a critical radius, which defines the smallest interaction distance for each cell in the system. For the case of $p = 1$ the critical radius can be analytically derived from Eq. 2 if we use a weak lower bound of 0 for B_3:

$$r_c = \frac{b_{max}}{2} + \sqrt{\frac{b_{max}^2}{4} + \sqrt{\frac{3B_2}{\alpha}}} \tag{3}$$

B_2 is simply the trace of the quadrupole moment tensor, and α is a user specified absolute acceleration tolerance. In more general cases (using a better bound on B_3, or with $p > 1$), r_c can be derived from the error bound equation using Newton's method. The overall computational expense of calculating r_c is small, since it need only be calculated once for each cell. 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 bound. Thus, our data-dependent MAC may replace the MAC in existing algorithms with minimal additional coding.

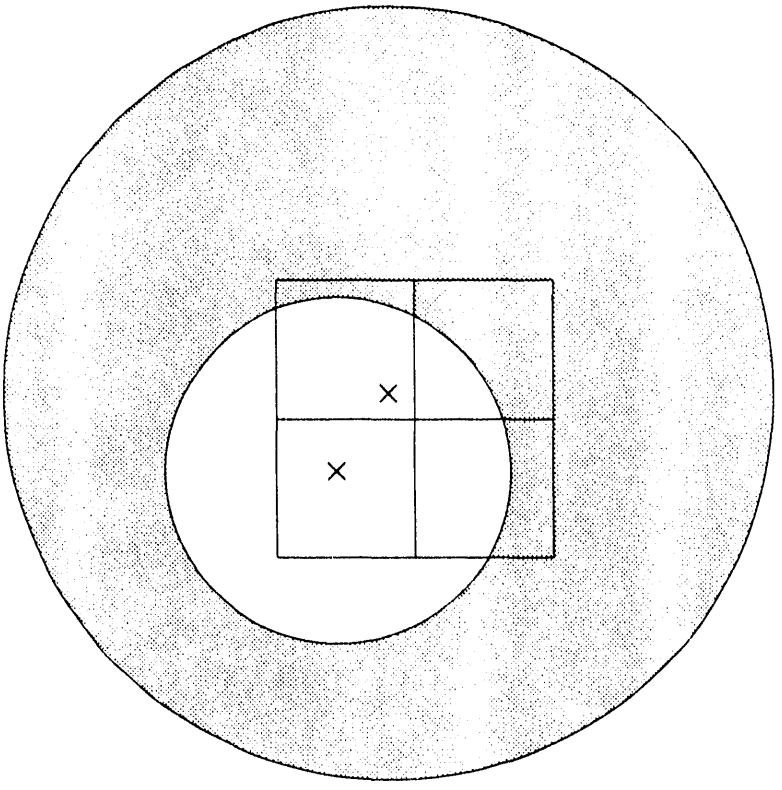


Figure 3: Each cell has its own critical radius. The critical radii of a cell and one of its daughters are shown here as circles. For the specified accuracy, a particle must lie outside the critical radius of a cell. The shaded region shows the spatial domain of all particles which would interact with 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.

However, one may also wish to use a relative error bound, where the accuracy is fixed as a fraction of the total force on a particle. It may also be useful to use a per-particle error bound, where the opening criterion is dynamically modified, based on the current sum of interaction errors. In this case, it can not be determined *a priori* which cells will need to be opened. This defeats parallel methods which are discussed in [9, 10]. The reason is that these methods require determination of locally essential data before the tree traversal begins. In contrast, this new class of opening criteria require access to non-local cells which can not be determined before the traversal stage.

3 Computational Approach

It is for the reason mentioned above that the algorithm described here was developed. It does not rely on any particular function which describes locally essential data; instead it provides a mechanism to retrieve non-local data as it is requested 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.

Instead of using pointers to describe the topology of a tree, we identify 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 (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.

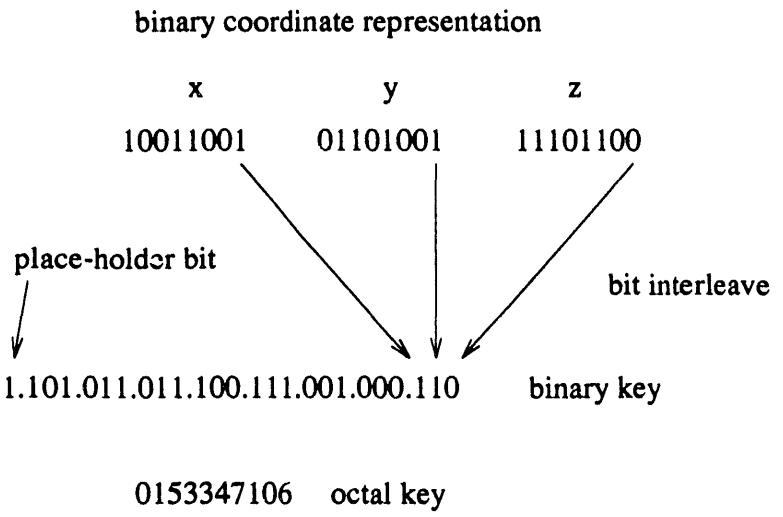


Figure 4: An illustration of the key mapping. Bits of the coordinates are interleaved and a place-holder bit is appended to the most significant bit. In this example, the 8-bit x, y and z values are mapped to a 25-bit key.

This is very similar to Morton ordering [11]. 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 append an additional 1-bit to the most significant bit of the body keys (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 a degeneracy among bodies and nodes whose most significant bits were all 0. The level immediately above the body level in the tree consists of keys which have d fewer bits. 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 pointer reference to this data, a hash table is used. A table with a length much smaller

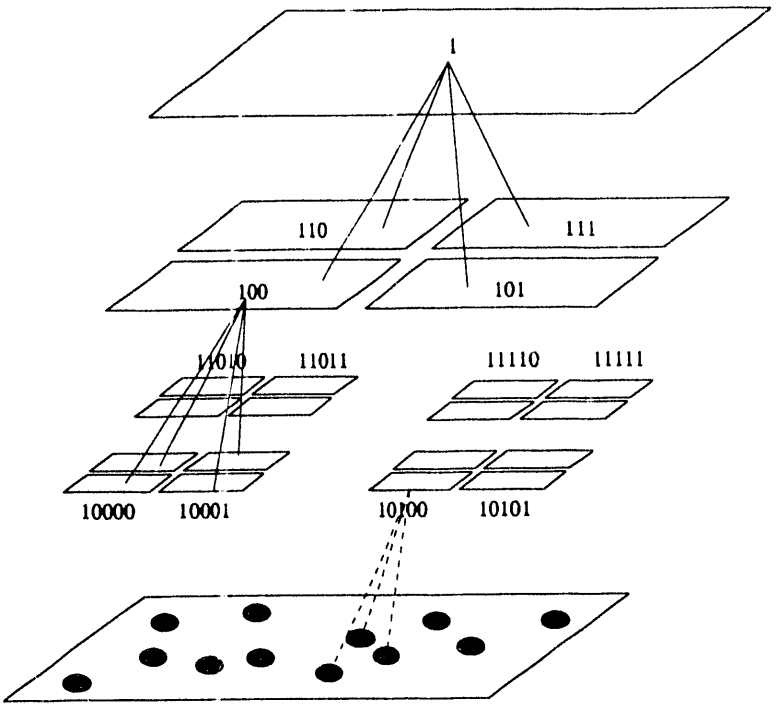


Figure 5: A quadtree shown along with the binary key coordinates of some of the internal nodes.

than the possible number of keys is used, and a hashing function is used to reduce a k-bit key to the h-bit long hash address. We use a very simple hashing function, which is to AND the key with $2^h - 1$. Collisions in the hash table are resolved via a linked list (chaining). The incidence of collisions, which can degrade performance a great deal, are discussed in a later section where we examine the memory access pattern of a typical tree traversal.

This 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 access (of $O(1)$ operations).

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 an integer with various flags which describe properties of the hcell and its corresponding cell. The data structure is detailed in Fig. 6.

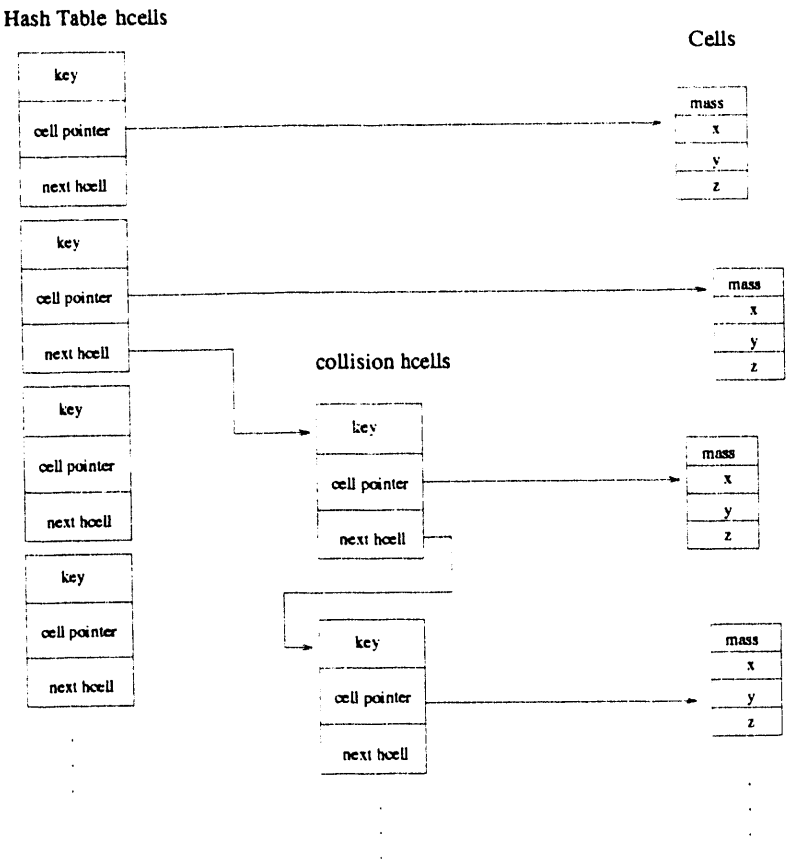


Figure 6: An illustration of the data structures used in the hashed tree. This shows the fixed number of *hcells* in the hash table, the extra *hcells* created to resolve collisions, and the *cells* which contain the physical data derived from the particles within the cell.

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 virtual shared memory parallel architecture model. 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.

3.2 Tree Construction

The higher level nodes in the tree can be constructed in a variety of ways. The simplest is that which was described in [3], but we now use key arithmetic in the traversal. Another idea is to first sort the key list into ascending order. This allows the construction to proceed more quickly, and

has other important advantages, as we shall see in later sections. After the keys are sorted, the cells in the tree are constructed by a linear traversal of the list, and construction and insertion of the cell nodes into the hash table. After the topology of the tree has been constructed, the contents (mass, etc.) of each cell may be updated by a tree traversal.

3.3 Domain Decomposition

The domain decomposition method is critical in 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. The idea is to simply cut the one-dimensional list of sorted body key ordinates 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 stages 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. 7 for a highly clustered set of particles (that which was shown in Fig 1) with $N_p = 16$.

We take advantage of the properties of the mapping of spatial coordinates to keys to produce a "good" domain decomposition (see Fig. 8). The method is very simply a sort of the keys across N_p processors. We can thus take advantage of highly tuned sorting algorithms on various machines. The best sorting algorithm is known to take approximately $N \log N$ time. However, it is usually the case that numerous tree constructions will take place in the course of a simulation. One should note that the positions of the bodies in the key space does not change significantly between timesteps, so one can take advantage of a more efficient $O(N)$ sort of the almost sorted data such as the Batcher merge algorithm [12], after the first timestep.

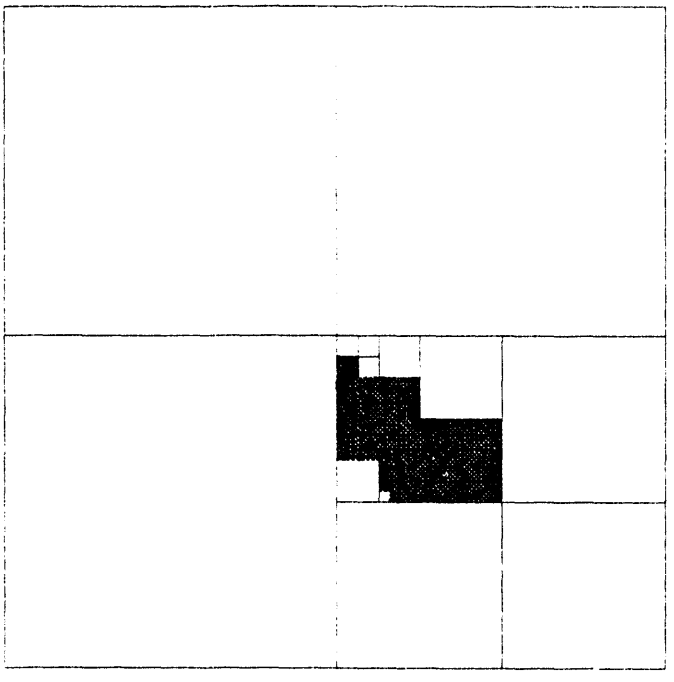


Figure 7: A processor domain for a clustered system of particles. The domain shown is a result of the decomposition strategy outlined in the text.

3.4 Tree Traversal

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

```
void
TraverseRecursive(Key_t key, void (*func)(hcell *))
{
    hcell *pp;
    unsigned int childnum;
    pp = Find(key);
    if (pp == NULL) return;
    func(pp);
    key = KeyLshift(key, NDIM);
    for (childnum = 0; childnum < (1<<NDIM); childnum++)
```

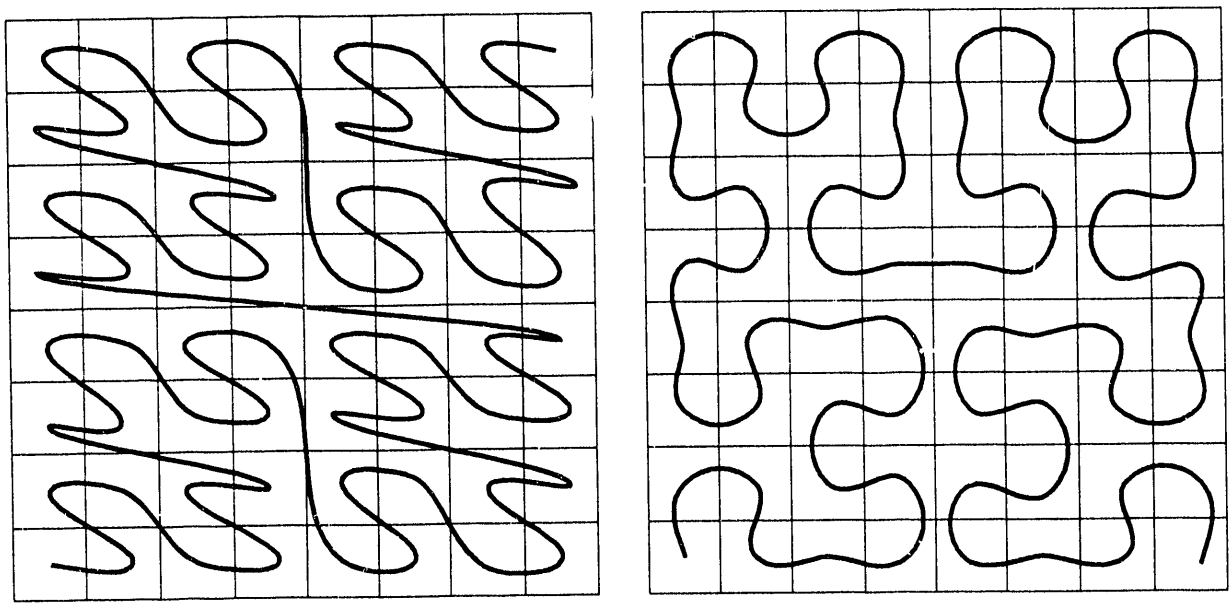


Figure 8: On the left, the path indicates the one-dimensional self-similar path which is induced by the map of interleaved bits. The domain decomposition is achieved by cutting the one-dimensional list into N_p pieces. On the right is a path that does not contain discontinuities which should produce a better decomposition, but which has not been implemented yet.

```
    TraverseRecursive(KeyOrInt(key, childnum), func);
}
```

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. 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 the request is being processed.

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 nice properties. We shall discuss the plain sequential method first, and then show the additions to allow efficient traversals on a parallel machine.

As an example, we shall use the traversal required for finding the cells which interact with a given particle, selected by some multipole acceptance criterion (MAC). The input to the list-based traversal is a *walk list* of hcell nodes. On the first pass, the walk list is simply a single entry of 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 and the partial traversal function returns, it is called again on the walk list returned from the previous call. The process terminates when there are no nodes returned on the walk list. The end result is the entire interaction list for the given particle, which may then be passed to the force calculation routine. 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.

3.5 Tree Traversal in a Virtual Shared Memory Model

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 is 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 in the request list, and an equal number of hcells on the defer list, which

require non-local data to be received before the traversal may continue.

The request list is then processed into single messages for each processor which has data requested from it. The defer list is saved until later, when it can be passed as a walk list to the traverse function. The traversal for the next particle can now proceed, without waiting for the data to return from the last request. With a modest amount of memory it is possible to store the context (which is completely described by the defer and interaction lists) for about 30 traversals. After 30 traversals have been deferred, we process the first deferral, since the data we have requested has probably arrived in the interim.

3.6 Memory Hierarchy and Access Patterns

Second, since the tree traversal is stressful to most memory architectures, it is very helpful to be able to relocate data which is used often to a smaller contiguous memory location. Hashing with collision resolution via chaining is a natural way to allow the program to perform it's own "virtual caching." This is also an advantage in a parallel implementation, since data which has been sent from another processor can be "erased" after it is likely to not be used again, which represents a large savings in parallel memory use efficiency. Lastly, the method is a bit more memory efficient, since pointers need not be stored in the cells. In three dimensions, rather than eight pointers in each cell, there is an overhead of two pointers and a key for each hash table entry.

3.7 Miscellaneous Tricks

One trick we have found useful to keep arithmetic operations independent of the number of spatial dimensions is to use the C preprocessor macro facility to perform vector operations. We define a macro "VV" which will perform vector arithmetic on vectors of a desired dimension:

```
#define VV(a, b) do { \
        a[0] b[0]; \
        a[1] b[1]; \
        a[2] b[2]; \
```

```
} while(0)
```

Thus, `VV(x, += y);` will add the 3-vector `y` to `x`, and `VV(x, = 3.0 * y);` will set `x` to 3y. This idea may be expanded in various directions (most of which are beyond the bounds of good taste), but it is a useful abstraction for programs which have not gone as far as the operator overloading features of C++.

We have also implemented a `Msg` library which performs a necessary service in the absence of a debugger. The `Msg` function allows us to print debugging and status information to individual files, and to control which information is provided via runtime arguments. We have found this mechanism far preferable to the *ad hoc* addition of `printf` statements to the code when trying to track down a problem.

A further library has been implemented which defines a flexible binary file format. The Self Describing File (SDF) format uses an ascii header which resembles a C structure definition to describe the names and types of the binary data in the file. The user code can then read data by requesting a "name" without knowing the details of how the data is arranged in the file. An additional benefit is that other file formats can be converted to SDF format by simply creating a suitable ascii header.

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. 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} 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	4
Tree Traversal	31
Data Communication During Traversal	6
Force Evaluation	52
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 10^8 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 system, 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 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 with number crunching, 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

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, Wincklemans 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 and Warren [8].

6 Future Improvements

The code described here is by no means a "final" version. This algorithm 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. 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 Conclusions

Problems of current interest in a wide variety of areas rely heavily on N-body methods. Accelerator beam dynamics, astrophysics (galaxy formation, large-scale structure), computational biology (protein folding, thermodynamics in aqueous solution), 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 interaction [13]. Such a vast improvement allows problems which are beyond the capabilities of a supercomputer (using conventional methods) to be solved on a small workstation. 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 of complex physical systems.

Acknowledgments

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. This work was supported in part by a grant from NASA under the HPCC program.

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

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

[13] 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.

END

DATE
FILMED
10/15/93