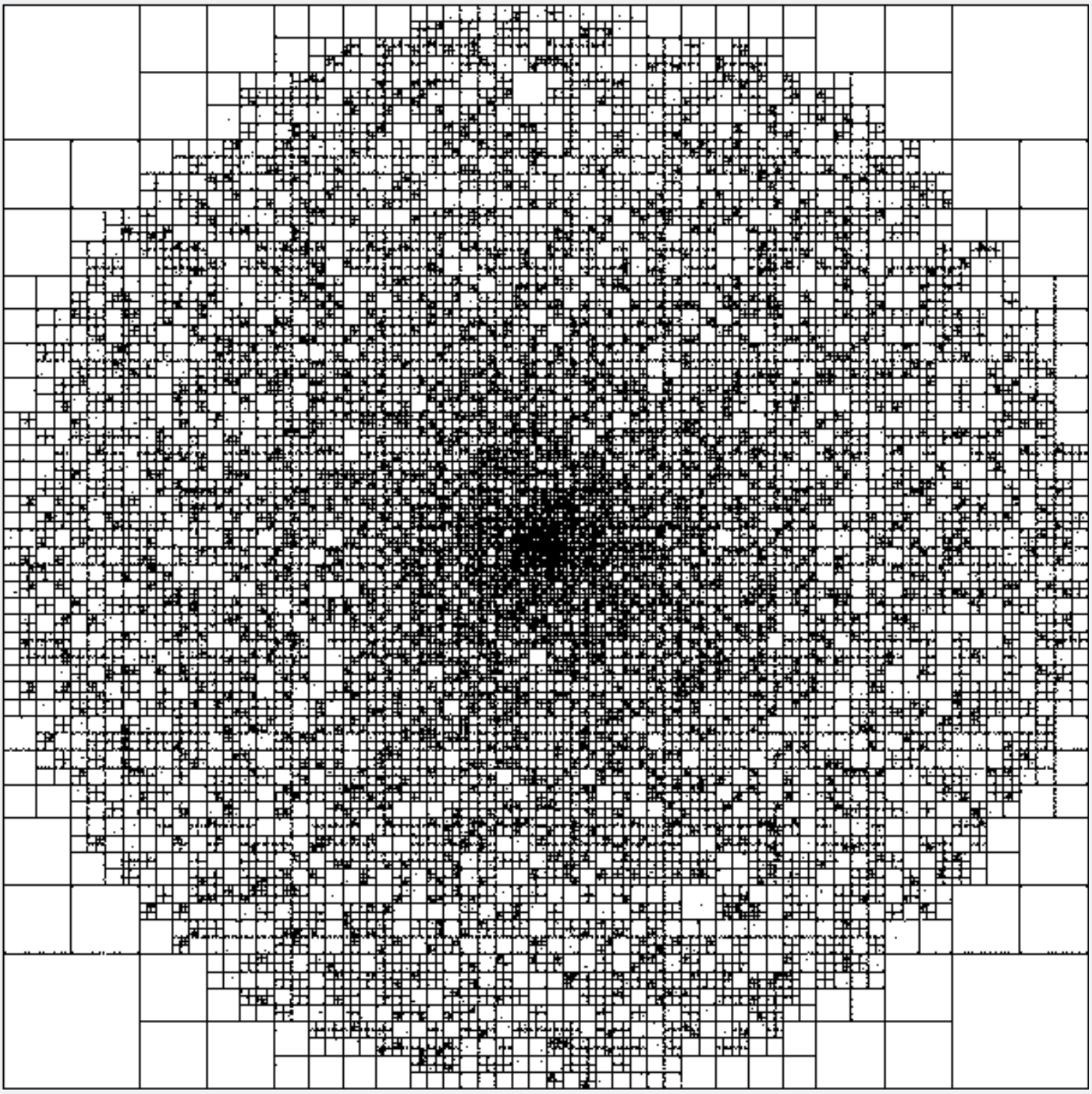
A Fast Tree Code for Many-Body Problems

Michael S. Warren and John K. Salmon



Parallel computing is bringing about a revolution in the study of the large-scale structure in the universe. Popular models, such as the cold-dark-matter model discussed in the main article, assume that a nearly homogeneous initial matter distribution has evolved through gravitational interactions to the froth-like distribution of galaxies observed in the night sky. But which of the proposed models actually predicts a matter distribution that matches those observations? Cosmologists use analytical methods to predict how the matter distribution evolves up to the time when nonlinear growth becomes important. If those preliminary results do not rule out a particular model, cosmologists turn to computer simulations to follow the nonlinear growth of structure. Only now, however, through the use of the latest massively parallel computers, can the simulations include enough particles to resolve simultaneously the many scales relevant to the problem.

We have developed a fast tree code that can handle tens of millions of particles and used it in the first high-resolution test of the cold-dark-matter model. The simulations were run on the Intel Touchstone Delta, a prototype massively parallel machine; results are presented in "Experimental Cosmology and the Puzzle of Large-Scale Structure." Here we discuss the innovative aspects of our tree code and its applicability not only to cosmology but also to a wide range of hydrodynamic and other many-body problems.

The challenge of simulating millions of particles as they move under the influence of mutual gravitational attraction is quite formidable. Because gravity is a long-range force (falling off only as the square of the distance), an exact algorithm would require calculating the force between each pair of particles at each timestep of the simulation. If there are N particles, there are $(N^2-N)/2$ pairs. Thus when N is large, the computation time for the force calculation is proportional to N^2—in the language of computer science, the time is $O(N^2)$. A brute-force simulation involving millions of particles would have required months on what is currently the fastest computer at Los Alamos (and arguably the fastest computer in the world), the 1024-processor CM-5 Connection Machine. Fortunately, there are approximation methods known as hierarchical tree methods that reduce the time needed for the force calculation from $O(N^2)$ to $O(N\log N)$ or $O(N)$, a dramatic reduction when N is in the millions.

Hierarchical tree methods were invented in 1985 and have been applied successfully to simulations of large-scale structure on scalar and vector computers. However, all those simulations lacked resolution on the desired range of scales desired for cosmology. Our particular challenge was to implement a tree method to simulate millions of particles on a massively parallel computer.

Three main difficulties arise in adapting tree codes to such a computer. First, the problem must be divided into similar or identical parts, one for each processor in the parallel machine, such that the parts require roughly equal amounts of computation; in computer jargon, the processors should be load-balanced. A standard method for problems of this type is to divide space into regions called processor domains and have each processor perform the calculations for the particles in the domain assigned to it. However, since the particles in cosmological simulations are distributed irregularly, there is no a priori division of space into domains that are load-balanced by virtue of having roughly equal numbers of particles. Moreover, since the particles move with respect to one another, domains that are initially load-balanced do not necessarily remain load-balanced.

The second difficulty relates to communication between processors. Most currently popular parallel computers have a distributed memory; that is, each processor is connected to its own memory, which stores the data for its domain. Communication between one processor and another is slow. Unfortunately, because we are studying long-range forces, considerable communication between processors is inevitable. If the communication is not minimized, the program will take prohibitively long to run.

The third difficulty relates to how the instructions for interprocessor communication are written. "Data-parallel" languages such as FORTRAN 90 hide the details of interprocessor communication from the programmer only when the same steps are performed simultaneously on a fixed large number of data elements, as in arithmetic involving large arrays. Since the calculations for particles in tree codes depend on how many other particles are nearby, the communication instructions must be programmed explicitly. Of the few calculations achieving high performance on massively parallel computers, most have been relatively regular and static problems that did not present difficulties comparable to those presented by complex many-body problems.

We have overcome these difficulties by applying what is called a key scheme for storing and retrieving data. That scheme provides an efficient way to describe both the particle locations and the organization of particle data into a computational tree. The key scheme, along with other innovations described below, produces a considerable advance in computing speed and resolution in the solution of N-body problems. For example, we know of

another astrophysical simulation, which was run on an IBM vector supercomputer, that used approximately the same number of particles as ours (marginally fewer). Our simulation provided twenty times greater spatial resolution than that simulation while requiring only a twentieth of the computation time. This article will discuss our tree code, focusing on a few aspects that can be described briefly. A more complete discussion appears in our paper "A Parallel Hashed Oct-Tree N-Body Algorithm," listed in the Further Reading.

We have taken care to produce a "friendly" code. Each part of the calculation is performed by a different section of the code, and the sections are as independent of one another as possible. The modularity was achieved through a large expenditure of programming time—we had to start from scratch, rather than modify a less well-organized program that we had originally written for sequential computers. But the modular structure has a large payoff: The code is easily adapted to solve other *N*-body problems. As the few modules specifically determined by cosmology can easily be replaced with modules describing interactions other than gravity, users do not need to be familiar with the details of the code that deal with parallel computation. An additional payoff is portability. We have developed modules for the machine-dependent parts, such as input-output and interprocessor communication of the program that allow it to run on computers ranging from ordinary sequential machines to clusters of workstations to massively parallel machines such as the CM-5 Connection Machine.

Our program is now serving several purposes. It has been indispensable in performing statistical analyses and data processing on the output of our simulations, since their size prohibits analysis on anything but a parallel supercomputer. We have written a module that can calculate both the three-dimensional dynamics of compressible fluids using smoothed-particle hydrodynamics (with or without gravity) and have adapted it to do three-dimensional incompressible hydrodynamics by a vortex-particle method as well. We plan to use smoothed-particle hydrodynamics to investigate galaxy formation, a critical step in connecting our cosmological studies to observations.

Our code can be applied to a wide variety of problems where long-range pairwise interactions dominate the computational cost. In addition to the areas mentioned above, accelerator beam dynamics, computational biology (protein folding), chemistry (molecular structure and thermodynamics), electromagnetic scattering, fluid mechanics with the panel method (commonly used to design subsonic airplanes), molecular dynamics, and plasma physics are those we know of, and there are certainly more. We are establishing collaborations with other researchers who we hope can successfully apply our code to current problems in a number of those fields. In addition, we are testing the efficiency of different computational approaches. Our *N*-body program may be unique in that it is being used both as a production astrophysics code and as a testbed for algorithms and interdisciplinary applications.

Computational Methods

The overall structure of the program is straightforward. First the net force on each particle from the others is calculated. Then the position and velocity of each particle at a slightly later time are computed from that force and its present position and velocity. This procedure (called a timestep) is repeated as often as the user wants or has computer time for. As mentioned above, the force calculation is the time-consuming part. Here we discuss our methods for reducing the amount of time consumed by the force calculation.

The tree method. The tree method is a way to take advantage of the basic approximation method of our program: the multipole expansion. A group of particles at a distance exerts almost the same force as a large single particle at the group's center of mass; approximating the group as a single particle is equivalent to using only the monopole term in the multipole expansion. However, when the group of particles is close to the particle on which the force is being calculated (especially when the distance is small compared to the size of the group), the monopole approximation is less accurate. In that situation, one can improve the approximation by using higher terms in the multipole expansion. Our program offers that option, but we have found that another approach leads to a faster program. We improve the approximation by using the basic idea of tree codes: dividing the group into smaller groups. Then each of the smaller groups can be treated as a single particle.

In calculating the force from a group on a given particle, one should divide the group finely enough to obtain good accuracy. However, dividing it more finely than necessary results in more calculations than necessary. To divide each group in an efficient way for each particle, one can set up a "tree," a hierarchy of finer and finer levels of detail, and use the coarsest acceptable level. In tree codes, space is divided hierarchically into a tree of cells. Figure 1 shows a two-dimensional tree analogous to the three-dimensional trees in our astrophysical simulation. The largest cell, the root of the tree, is the entire region of space. That cell is di-

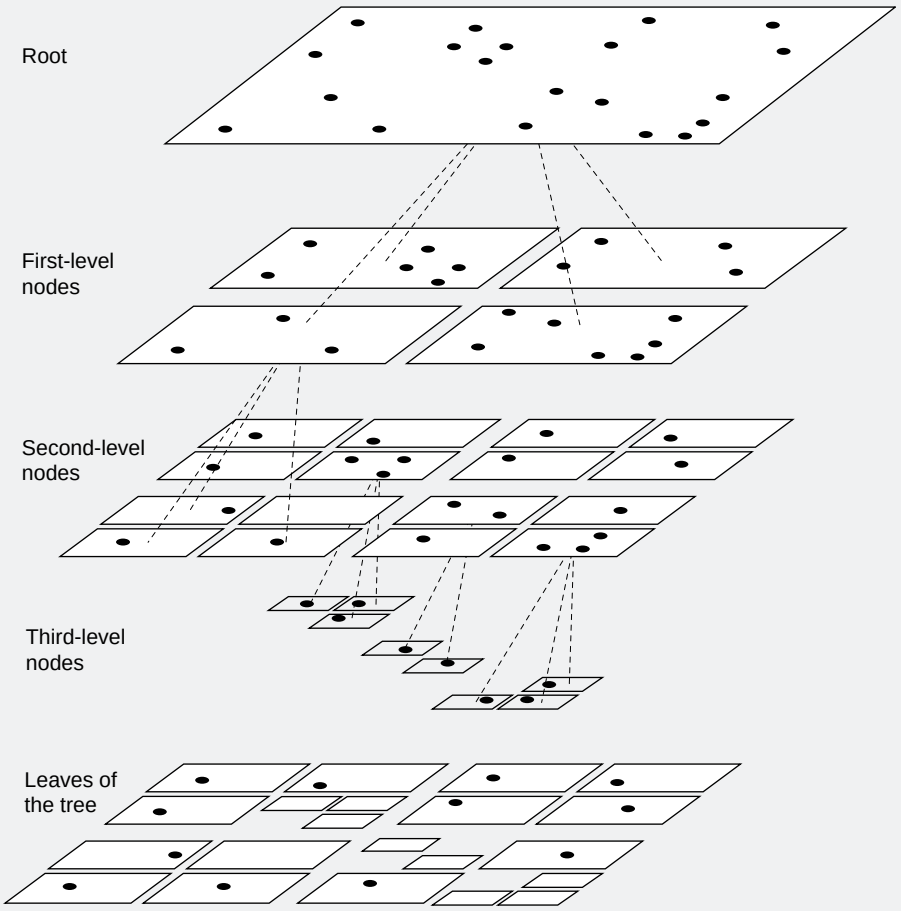


Figure 1. A Regular Two-Dimensional Tree

The tree has been produced by dividing space into square cells so that each particle is in a separate cell. The branches of the tree connect each square to the smaller squares or rectangles made by cutting through its center. The smaller squares in turn can branch out by being further divided. The leaves are the squares and rectangles that contain exactly one particle and therefore do not need to be divided further. At the bottom is a "flat" representation of the tree structure induced by the particles. Since the cells are squares and are divided orthogonally through the center, a cell can have at most four daughters. Such a tree is called a quad-tree. The analogous three-dimensional trees in our cosmological simulations are called oct-trees.

vided into smaller "daughter" cells, and they are in turn divided into smaller cells, and so forth. Cells containing one particle are not divided; they are the leaves of the tree. (Areas of space containing no particles are ignored.) Thus the structure of the tree adapts to the positions of the particles, having many levels of refinement where particles are densely clustered. The structure must be recalculated every time the particle positions are updated.

Our tree method must include a criterion for determining when to approximate the force due to a group of particles as a force due to a monopole at the group's center of mass. The minimum distance is called the critical radius, r_c. The method of calculating the critical radius, or "multipole-acceptance criterion," is crucial to the speed and accuracy of the tree code. As described below, we calculate the r_c of a group of particles from its distribution—in particular, from its higher multipole moments, which are precisely the terms discarded by the monopole approximation—in such a way that the error of the monopole approximation is less than a limit set by the user.

To calculate the force on a given particle from the others, one "traverses" the tree node by node, starting at the root and going to finer and finer levels of detail. As illustrated in Figure 2, whenever the monopole approximation is acceptable, one skips all daughters and further descendants of that cell, thus saving the time that would be needed to calculate the force from each particle in that cell individually. If instead the cell is closer than r_c, one repeats the process with each of the cell's daughters, each of which has its own r_c. The process of examining smaller and smaller cells can continue until forces from individual particles are calculated, if necessary. The execution time of the tree traversal described here is $O(N \log N)$. The enormous speedup when N is large justifies spending a relatively small amount of time in the program to rebuild the tree at every timestep—and also justifies our spending a good deal of our own time in developing the program.

Keys. A structure as complex as a tree is difficult to implement on a parallel computer. The description of the tree, which includes the coordinates of each cell and properties such as center-of-mass location and multipole moments, must also provide a way of finding the daughters of any cell so that the

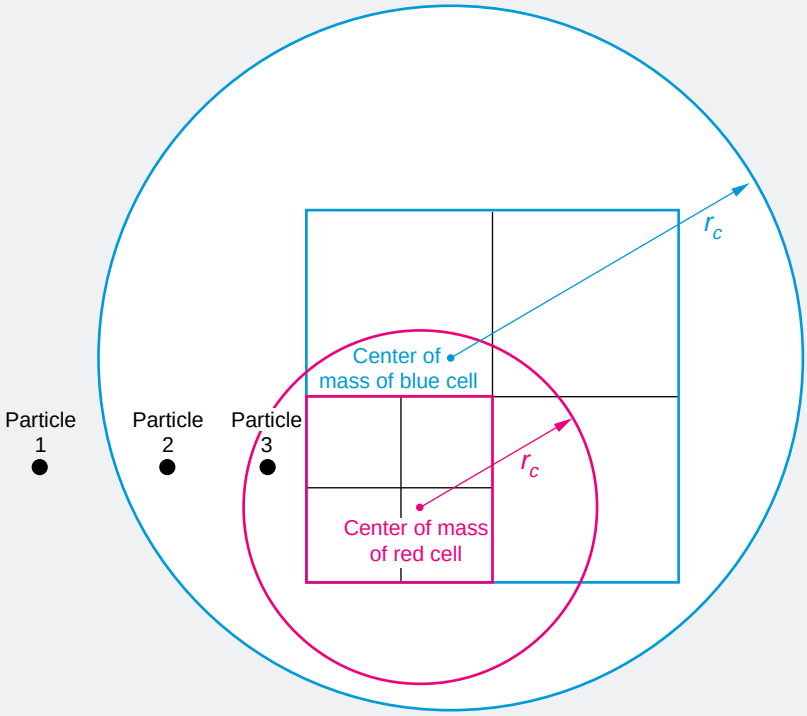


Figure 2. The Multipole-Acceptance Criterion

When the distance between a particle and a cell is greater than the critical radius of the cell, all the mass in the cell can be treated as a single point mass; that is, the force exerted by the particles in the cell can be calculated with the use of the monopole approximation. The radii of the blue and red circles are the critical radii of the cells shown in blue and red respectively. Particle 1 is outside the critical radius of the cell shown in blue, so the monopole approximation is used to find the force on particle 1 due to all the particles in the blue cell together. Particle 2 is inside the critical radius of the blue cell, so the monopole approximation is not applied to the force exerted on it by the particles in the blue cell. However, since the particle is outside the critical radius of the red cell, the monopole approximation is used to find the force exerted on it by the particles inside the red cell (and similarly for the other three daughters of the blue cell). Finally, particle 3 is inside the critical radius of the red cell, so the monopole approximation can be used only for the force exerted on it by even smaller cells (with boundaries in black) within the red cell.

tree can be traversed. Trees are most often described by storing the addresses of each cell's daughter cells along with the data (such as the mass) for the cell. Those addresses are called pointers to the daughter cells.

The pointer method has two disadvantages for parallel programs. First, since each pointer is dynamically determined by such considerations as the address of the next available memory location, the memory location the pointer points to has nothing to do with the spatial location of the cell. Second, when one processor must retrieve information about cells in the domain of another processor, the pointers in a parent cell in one processor must be somehow

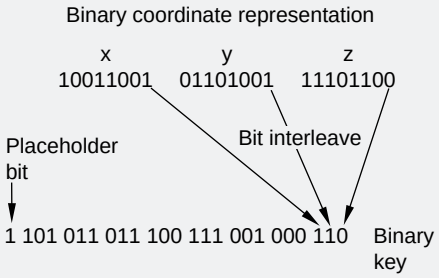


Figure 3. The Key Mapping

The key for each particle is generated from its coordinates measured from a corner of the region of space (a cube). The bits of the coordinates are interleaved and a 1-bit is attached to the beginning of the key as a place-holder, to distinguish particle keys from cell keys. The key derived from three single-precision floating-point numbers fits nicely into a single 64-bit integer or a pair of 32-bit integers. In this example, the 8-bit x, y, and z coordinates are mapped to a 25-bit key.

translated into valid addresses of daughter cells in another processor.

One solution is for each processor to retrieve all the data it could possibly need at an early stage in the program. It can then build its own "private" copy of the tree structure, and proceed with the rest of the calculation without having to worry about where the data are (because it has already guaranteed it has all the data it will need). In fact, our first version of a parallel tree code worked in that way. However, determining beforehand which data are needed can be somewhat complicated. In our current program the multipole-acceptance criterion requires knowing the contents of each particular cell (which aren't known until the tree is traversed). In this case, an easier communication method is for each processor to ask for data when they are needed, not before. This method requires a mechanism to "ask" for each piece of data and retrieve it from another processor in an efficient manner.

The identifier of each particle in our simulation is a key derived from the particle's coordinates, as shown in Figure 3. We translate keys into addresses

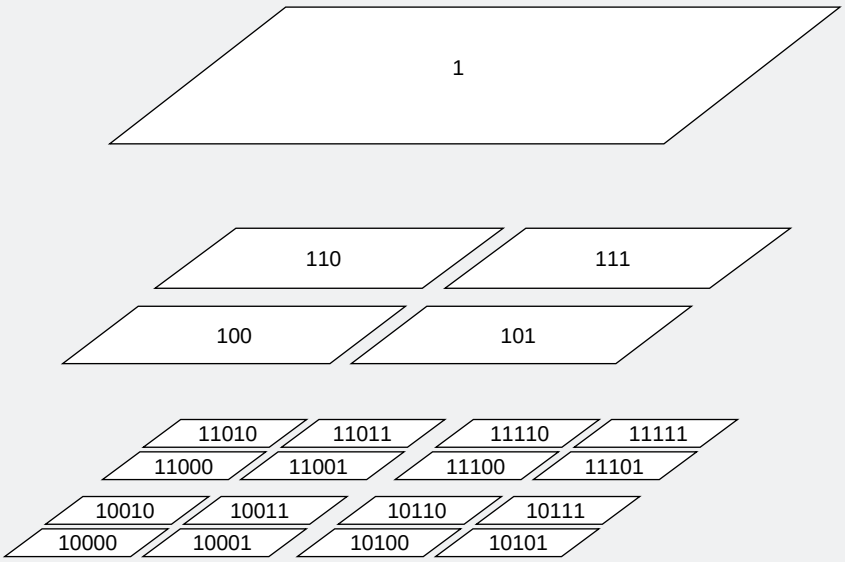


Figure 4. Cell Keys

The figure shows a two-dimensional tree together with the key associated with each node. The keys are generated by interleaving bits of the coordinates in the same way that particle keys are generated from particle coordinates. In fact, the coordinates of each cell are the initial bits that the coordinates of all the particles in that cell share. When particles are very close together, cell coordinates may have as many bits as particle coordinates. In order to distinguish the higher-level nodes of the tree from the lower-level nodes, we attach an additional 1-bit to the most significant bit of every key (the place-holder bit). Without the place-holder bit, there would be no distinction between the keys 11 and 000011, for instance. The root node is represented by the key 1.

of cell data through a standard technique called hashing. Given the key of a particle, the data for the particle can be rapidly retrieved, even by one processor from another; the key scheme provides a uniform addressing mechanism. We use a similar scheme to generate the address of each cell from its coordinates, as shown in Figure 4. Here the key scheme allows us to code tree traversals as simply as when using pointers, because we can find the keys of daughter or parent cells by performing simple bit arithmetic on the key of a cell. It also allows us to find any node of the tree in time that is $O(1)$, that is, independent of N. (In contrast, if we want to find a particular node of a tree whose topology is described by pointers, we must start at the root of the tree and traverse until we find the desired node, which takes $O(\log N)$ time.)

The key method is particularly advantageous when we sort the particles in the numerical order of their keys. Because of the way the keys depend on the coordinates, particles whose addresses are near each other in the sorted list are usually near each other in space. That property is useful in several parts of the program, as we shall see below.

Features of the Code

The organization of our program is sketched in Figure 5. Each part must be made to work efficiently on a massively parallel computer. Techniques for doing so are discussed in "A Parallel Hashed Oct-Tree N-Body Algorithm." Here we discuss a few techniques that can be described simply.

The multipole-acceptance criterion and analytic error bounds. The multipole-acceptance criterion is crucial to the accuracy and efficiency of the program. Our criterion rejects all particle-cell interactions where the approximation would introduce large errors, while accepting as many interactions as possible where the error is small so as to avoid unnecessary computation. Until now, multipole-acceptance criteria in many-body simulations have incorporated information about the size of a cell, but no information about its contents other than the position of its center of mass. Some of them also had the disadvantage that the worst-case errors arising from the approximation were unbounded, and the errors in realistic situations could be quite large. The popular "fast multipole method" does have a well-defined worst-case error bound, but has proved to be quite slow in three dimensions.

We developed a multipole-acceptance criterion that directly depends on the contents of the cell, specifically, on the largest distance of a particle in the cell from the cell's center of mass and on the first two multipole moments discarded by the approximation—the dipole and quadrupole moments, when we keep only the monopole term in the expansion. (The calculation of r_c is then particularly simple; because negative masses don't exist, the dipole moment of any mass distribution about its center of mass vanishes.) Because the criterion depends on the information discarded by the approximation, we can set an error bound and know that our use of the multipole approximation

introduces no errors larger than that bound. We have tested the speed and accuracy of our multipole-acceptance criterion using several sets of initial conditions. Our criterion moderately decreased the root-mean-square error and decreased the maximum error by factors ranging from 3 to 10. Details are given in the article "Skeletons from the Treecode Closet," listed in Further Reading.

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

Our method is to cut the list of particle keys into a number of pieces equal to the number of processors. The divisions are placed so that the pieces require equal total amounts of work; the work for each particle is readily approximated by counting the number of cells and particles the given particle interacted with on the previous timestep. The method tends to produce processor domains that consist of spatially grouped particles. The grouping greatly improves the efficiency of the traversal stage of the algorithm, since the amount of data needed from other processors is roughly proportional to the surface area of the processor domain. Figure 6 illustrates how this divides a centrally clustered two-dimensional set of particles among 16 processors. One source of inefficiency is that our method of generating keys from coordinates creates a number of spatial discontinuities in the sorted list. A processor domain can span one of those discontinuities and thus consist of two spatially separated groups of particles. We have used a different ordering, which does not contain discontinuities, but it improves performance only slightly.

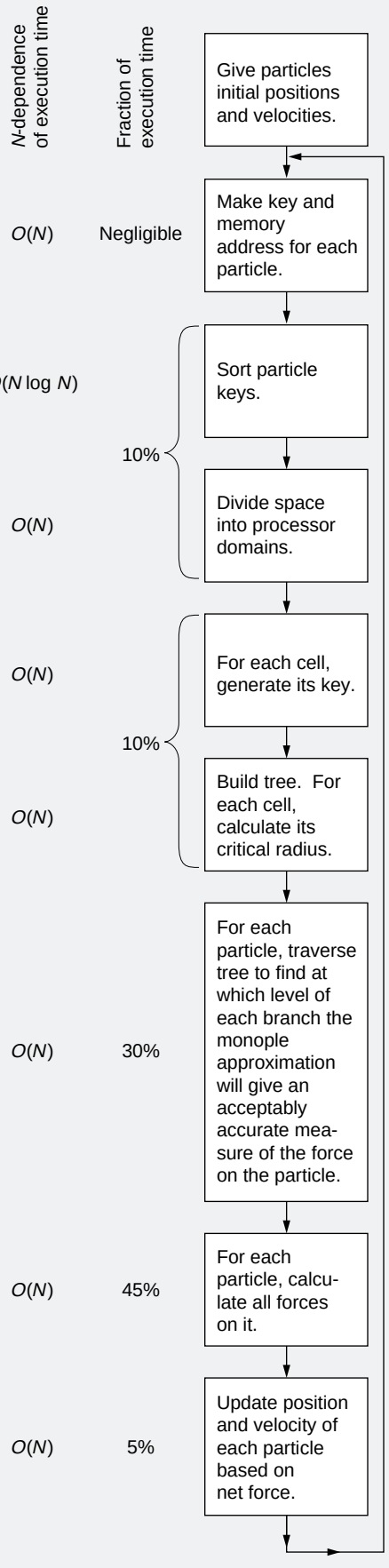


Figure 5. The Organization of our Algorithm

All parts are done in parallel. After the initialization of particle positions and velocities, the process is repeated the number of times specified by the user—usually hundreds or thousands of timesteps.

Tree construction. Sorting the particle keys is also advantageous in tree construction. In fact, our original reason for sorting these identifiers was to improve the tree-construction stage. In the usual algorithm for constructing a tree, each particle is inserted at the root of the partially constructed tree. The algorithm determines which of the top-level cells includes its position, then which of that cell's daughters, and thus the particle moves downward one node at a time until a new cell is created to be its leaf. That process is $O(\log N)$ for each particle. In our code, however, the particles are added to the tree in the sorted order, and each one is inserted not at the root but at the location of the last particle inserted. Since particles near each other in the sorted list are usually near each other in space, moving a particle to its correct location in the tree is now on average $O(1)$. This single increase in efficiency makes up for the time spent in sorting.

Memory hierarchy and access patterns. Tree codes place heavy demands on the memory subsystems of modern computers because the amount of data that must be transferred between processors is large and not well-ordered. We have encouraged a more orderly and efficient memory-access pattern by arranging the order of computation to take advantage of the underlying structure of the algorithm. In

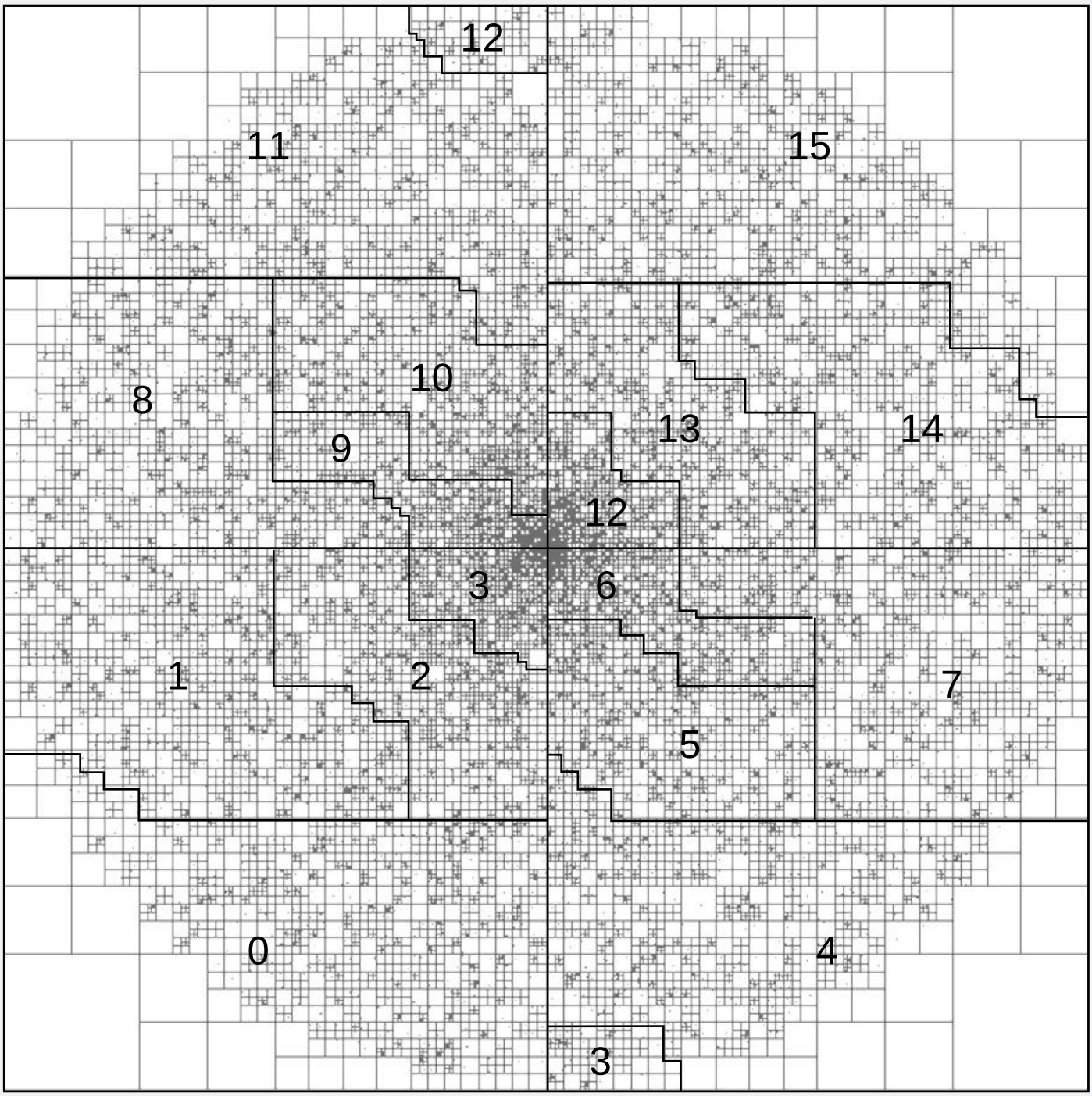


Figure 6. Data Decomposition

Shown are the processor domains assigned to all 16 processors by a data decomposition for a clustered system of particles. Each domain shown is the smallest that contains all the particles in a section of the sorted particle-key list. The compactness of the domains reduces the amount of interprocessor communication required in evaluating the interaction of the particles in the domain.

particular, the computation runs fastest when each processor retrieves most of the data it uses from the memories to which it is connected directly. We wish to keep data for as long as possible in the fastest level of the hierarchy that comprises registers, cache, local memory, other processors' memory, and virtual memory. A helpful property of tree algorithms is that particles that are near each other tend to interact with almost the same sets of cells; thus the calculations of those interactions require almost the same data. By updating the particles' positions in the order of the sorted key list, we greatly reduce the fraction of slow memory accesses. On parallel computers we could extend

this "virtual cache" model even further by erasing from each processor's memory information from other processors that has not been used recently. We expect that implementing this technique will allow significantly larger simulations to take place by eliminating copies of cells from other processors (which currently uses the largest amount of memory, and thus limits the maximum size of the simulation).

Portability. A portable program is one that can be run on a variety of computer systems after only a small amount of time is spent changing the program for each particular computer. In contrast, a non-portable program may take nearly as long to rewrite for each system as it took to write in the first place. Computational scientists would rather spend that time solving new problems or writing better algorithms.

In the modern world of rapidly evolving parallel architectures, a particular brand of computer may be the best available for a time measured in months. If one wants to solve physics problems on the best parallel computers, one does not have much time to get a program working on each new model. Thus portability increases the problem-solving efficiency of the combined system of scientist, programmer, and computer. However, one must always keep in mind the tradeoffs between portability and other desirable features of a program (such as speed). In some cases the problem-solving efficiency is increased instead by methods (such as writing often-executed inner loops in a particular computer's assembly language) that sacrifice some portability for increased speed.

A concept related to portability is the "adaptability" of a program. One would rather not solve almost the same programming problem time after time. The addition of more complex physical laws to a simulation and the application of it to very different physical problems that have a similar computational structure are common needs. If one can isolate individual parts of a program as modules that they can be easily replaced without affecting the behavior of the other modules in the program, one can write software that is much more adaptable to different problems.

We have tried very hard to design our code to meet the dual goals of portability and adaptability, which present an even greater challenge than usual when one is using parallel machines. Our efforts have resulted in a code that allows scientists in many disciplines to benefit from the enormous speedup offered by tree methods, not only to solve problems on supercomputers that were simply insoluble before (such as our cosmological problems), but also to perform on workstations computations that formerly required supercomputers. We have adapted the code to run on many different computers, from workstations to the latest massively parallel machines. That adaptation has required the definition of a few "standard" message-passing functions that can be easily implemented on any type of distributed memory parallel computer. To adapt the code to a new machine, one need only change the implementations of a few functions in a single file. We have also implemented these calls with the standard Parallel Virtual Machine (PVM) library and the newly established Message Passing Interface (MPI) standard. Thus, the program should run without modification on any machine that implements these standards. We hope that our code can become a successful model for a "standard library" that can be used by scientists who do not have (or do not want) a detailed knowledge of how parallel computers work.

Performance

We timed the various stages of the algorithm on the 512-processor Intel Touchstone Delta installed at Caltech and partially owned by the Laboratory, which is a prototype of Intel's Paragon supercomputer. The timings are from an 8.8-million-particle production run simulating the formation of structure in a cold-dark-matter universe. During the initial stages of the calculation, the particles are spread uniformly throughout the spherical computational volume. We set an absolute bound on the error of the acceleration due to each interaction; the error bound is 10^{-3} times the mean acceleration of particles in the simulation. This bound results in 2.2×10^{10} interactions per timestep in the initial unclustered system. At this stage the program runs at 5.8 billion floating-point instructions per second (gigaflops).

Computation Stage	Time (s)
Domain Decomposition	7
Tree Building	10
Tree Traversal	33
Data Communication	6
Force Evaluation	54
Load Imbalance	7
Total (5.8 Gflops)	114

In 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. At this stage the program runs at 4.9 gigaflops.

Computation Stage	Time (s)
Domain Decomposition	19
Tree Building	0
Tree Traversal	55
Data Communication	4
Force Evaluation	60
Load Imbalance	12
Total (4.9 Gflops)	160

Almost half of the execution time is spent in the force-calculation routine. 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 hand-coded the force-calculation routine in assembly language. The resulting routine runs at a speed of 28 megaflops per processing node.

If we count as "useful work" only the floating-point operations performed in the force-calculation routine (30 flops per interaction) the overall speed of the code is about 5–6 gigaflops. 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, but in tree traversal and data-structure manipulation. The integer-arithmetic and addressing speeds of the processor are as important as the floating-point performance. We hope that evaluation of processors does not become overbalanced toward floating-point speed at the expense of integer arithmetic and memory bandwidth. Our code provides a good example of why a balanced processor architecture is necessary for good overall performance.

Conclusion

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 that we find useful. We will continue to use it not only to study the process of galaxy formation, but also to investigate multipole algorithms. We have been studying the addition of cell-cell interactions (similar to those used in the fast multipole method), which reduces the N-dependence of the algorithm from $O(N\log N)$ to $O(N)$. Cell-cell interactions are performed by approximation of the gravitational field at each particle in a cell by Taylor expansion about the center of the cell. Preliminary results indicate that this new method reduces the number of interactions by a factor of 5 in a simulation of 1 million bodies.

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

- The fundamental ideas in this algorithm are, for the most part, standard tools of computer science (key mapping, hashing, sorting). In combination, they form the basis of a clean and efficient parallel algorithm. Such an 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 computing speed of the code on an extremely irregular, dynamically changing set of particles that require global data for their update, using a large number of processors (512), is comparable with the performance quoted for much more regular and static problems, which are sometimes identified as the only type of "scalable" algorithms that 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 like those 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. □

Further Reading

John K. Salmon and Michael S. Warren. 1992. Skeletons from the treecode closet. *Journal of Computational Physics* 111:136–155.

John K. Salmon, Michael S. Warren, and Grégoire S. Winckelmans. 1994. Fast parallel treecodes for gravitational and fluid dynamical N-body problems. *International Journal of Supercomputer Applications* **8**, in press.

Michael S. Warren and John K. Salmon. 1993. A parallel hashed oct-tree N-body algorithm. In *Supercomputing '93*. IEEE Computer Society Press.

John K. Salmon is a research fellow in physics at California Institute of Technology. He holds a B.S. in physics and a B.S. in electrical engineering and computer science from Massachusetts Institute of Technology, an M.S. in physics from the University of California, Berkeley, and a Ph.D. in physics from the California Institute of Technology. His research interests include applications of parallel fast particle methods to problems in astrophysics, computational fluid dynamics and other areas of computational science. With Michael Warren, he won the 1992 Gordon Bell Prize for achievement in high-performance computing.

The biography of co-author Michael S. Warren appears on page 81.