Load Balancing and Data Locality in Adaptive Hierarchical N-body Methods: Barnes-Hut, Fast Multipole, and Radiosity

Jaswinder Pal Singh, Chris Holt, Takashi Totsuka,
Anoop Gupta and John L. Hennessy

Computer Systems Laboratory
Stanford University
Stanford, CA 94305

Abstract

Hierarchical N-body methods, which are based on a fundamental insight into the nature of many physical processes, are increasingly being used to solve large-scale problems in a variety of scientific/engineering domains. Applications that use these methods are challenging to parallelize effectively, however, owing to their nonuniform, dynamically changing characteristics and their need for long-range communication.

In this paper, we study the partitioning and scheduling techniques required to obtain effective parallel performance on applications that use a range of hierarchical N-body applications. To obtain representative coverage, we examine applications that use the three most promising methods used today. Two of these, the Barnes-Hut method and the Fast Multipole Method, are the best methods known for classical N-body problems (such as molecular or galactic simulations). The third is a recent hierarchical method for radiosity calculations in computer graphics, which applies the hierarchical N-body approach to a problem with very different characteristics.

We find that straightforward decomposition techniques which an automatic scheduler might implement do not scale well, because they are unable to simultaneously provide load balancing and data locality. However, all the applications yield very good parallel performance if appropriate partitioning and scheduling techniques are implemented by the programmer. For the Barnes-Hut and Fast Multipole applications, we show that simple yet effective partitioning techniques can be developed by exploiting some key insights into both the solution methods and the classical problems that they solve. Using a novel partitioning technique, even relatively small problems achieve 45-fold speedups on a 48-processor Stanford DASH machine (a cache-coherent, shared address space multiprocessor) and 118-fold speedups on a 128-processor simulated architecture. The very different characteristics of the radiosity application require a different partitioning/scheduling approach to be used for it; however, it too yields very good parallel performance.

1 Introduction

Hierarchical N-body methods, which are based on a fundamental insight into the nature of many physical processes, are increasingly being used to solve a wide variety of scientific and engineering problems. Since these methods also afford substantial parallelism, applications that use them are likely to be among the dominant users of high-performance multiprocessors. Such applications, however, typically have characteristics that make it challenging to partition and schedule them for effective parallel performance. In particular, the workload distribution and communication patterns across the logical units of parallelism (the bodies, for example) are both nonuniform and also change as the computation proceeds, thus complicating the simultaneous incorporation of load balancing and data locality.

In this paper, we study partitioning and scheduling issues in representative applications that use three important hierarchical N-body methods. Two of these methods—the Barnes-Hut method [3] and Greengard and Rokhlin's Fast Multipole Method (FMM) [15]—are the best methods known for classical N-body problems, such as those in astrophysics, electrostatics and molecular dynamics. In addition to classical problems, the hierarchical

N-body approach has recently been applied with great success to many different domains, including a problem with very different characteristics: radiosity calculations for global illumination in computer graphics [16]. This constitutes our third application, and we believe the three together provide adequate coverage of the use of the hierarchical N-body approach today.

For each application, we first examine partitioning/scheduling techniques that are conceptually natural, and are the best that a compiler or automatic scheduler might implement if it knew the appropriate parallelism. We find that these techniques do not provide scalable parallel performance—although they suffice for small-scale parallelism on the classical applications—since they are unable to simultaneously provide load balancing and the desired locality of data access. However, all the applications can be made to yield excellent and scalable parallel performance if more sophisticated techniques are implemented by the programmer.

The only other parallel version of a nonuniform hierarchical N-body application is a message-passing implementation of an astrophysical Barnes-Hut application [23]. It uses an orthogonal recursive bisection (*ORB*) partitioning technique to obtain both load balancing and data locality. (It is also substantially complicated by the lack of a shared-address-space programming model [28]). We propose a new partitioning technique called *costzones* for the classical applications. *Costzones* is much simpler to implement than *ORB*, and performs better on shared address space machines, particularly as the number of processors increases. We also extend both the *ORB* and *costzones* techniques to obtain good performance on our other classical application, the FMM. We find that *costzones* is more easily extended and yields better performance than *ORB* on the FMM as well. Using *costzones*, we obtain 45-fold speedups on a 48-processor Stanford DASH machine, a cache-coherent shared address space multiprocessor, on both classical applications, and 118-fold speedups on a 128-processor, simulated shared address space architecture for the Barnes-Hut application, even though the problem sizes we run are relatively small.

Finally, although the radiosity application uses the hierarchical N-body approach, the characteristics of this application are different enough from those of classical N-body applications that the *costzones* and *ORB* techniques are neither easily implementable nor useful in partitioning it. Although the classical applications require periodic repartitioning, explicit repartitioning by the user suffices in those cases and no on-the-fly stealing of tasks from other partitions is necessary for load balancing. The radiosity application, on the other hand, is far more unpredictable at runtime, and requires dynamic task-stealing. Fortunately, there is substantial reuse of data within a task in this application, and simple techniques to preserve locality across tasks are successful in keeping the communication overhead low despite task-stealing. We obtain 27-fold speedups on a 32-processor DASH machine for the radiosity application.

Section 2 of this paper demonstrates the importance and the range of application of hierarchical N-body methods in solving physical problems. Section 3 addresses our choice of applications and the coverage of hierarchical N-body methods that we achieve with them. In Section 4, we describe the applications we study and the algorithms they employ. Section 5 describes the multiprocessor environments that we run our applications on. In Section 6, we outline the goals of our parallel implementations and the common characteristics of the applications that complicate the achievement of these goals. Section 7 describes the metrics we use to compare different partitioning and scheduling techniques, and outlines the organization of our experiments. Sections 8 to 10 discuss partitioning and scheduling techniques for the classical (Barnes-Hut and FMM) and radiosity applications in detail, and present the results we obtain. Finally, and Section 11 concludes the paper.

2 Importance of Hierarchical N-body Methods

Hierarchical algorithms that efficiently solve a wide range of physical problems have recently attracted a lot of attention in scientific computing. These algorithms are based on the following fundamental insight into the physics of many natural phenomena: Many physical systems exhibit a large range of scales in their information requirements, in both space and time. That is, a point in the physical domain requires progressively less information less frequently from parts of the domain that are further away from it. Hierarchical algorithms exploit the range of spatial scales to efficiently propagate global information through the domain. Prominent

among these algorithms are N-body methods, multigrid methods, domain decomposition methods, multi-level preconditioners, adaptive mesh-refinement algorithms, and wavelet basis methods [5]. Our focus in this paper is on hierarchical N-body methods.

The classical N-body problem models a physical domain as a system of discrete bodies, and studies the evolution of this system under the influences exerted on each body by the whole ensemble.[1] Computing the interactions among bodies is expensive, particularly when long-range interactions cannot be ignored. For examples, if all pairwise interactions are computed directly, the complexity of computing interactions is ^{2}, which is prohibitive for large systems. Hierarchical tree-based methods have therefore been developed that reduce the complexity to for general distributions [2, 3], or even for uniform distributions [15], without losing much accuracy in long-range interactions.

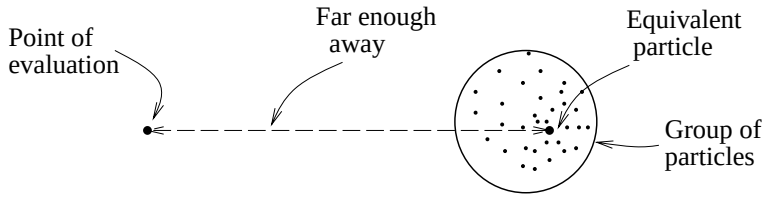


Figure 1: Approximation of a group of particles by a single equivalent particle.

The particular form that the above insight takes in N-body problems dates back to Isaac Newton in the year 1687: If the magnitude of interaction between particles falls off rapidly with distance, then the effect of a large group of particles may be approximated by a single equivalent particle, if the group of particles is far enough away from the point at which the effect is being evaluated (see Figure 1). The hierarchical application

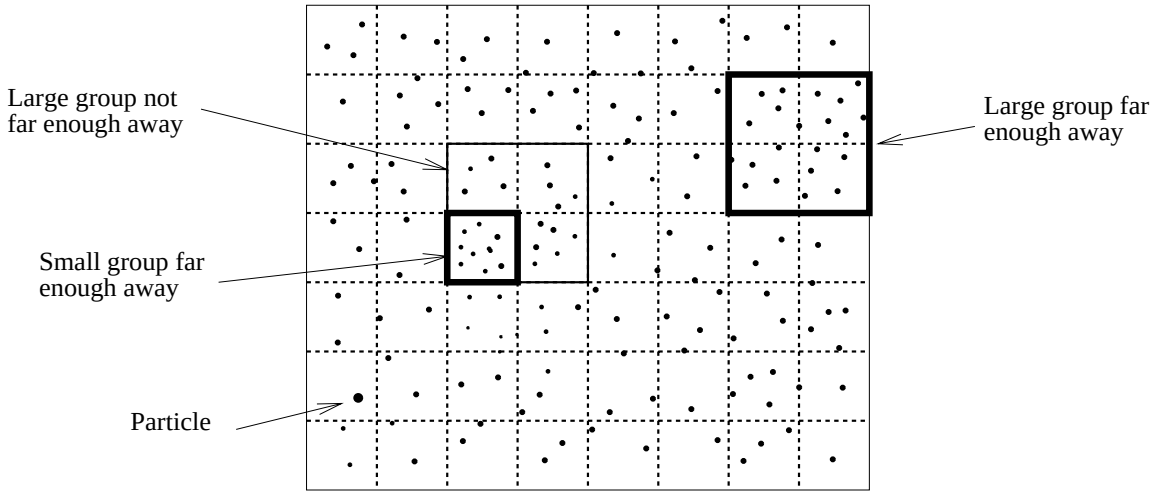


Figure 2: Hierarchical approximation of cells by equivalent particles.

of this insight—first used by Appel [2] in 1985—implies that the farther away the particles, the larger the group that can be approximated by a single particle (see Figure 2). Although Newton arrived at his powerful insight in the context of gravitation, hierarchical N-body methods based on it have found increasing applicability in various problem domains. To demonstrate the wide-ranging applicability of these methods and their consequent importance for high-performance computing, we list some of the problem domains below [14]:

1. **Astrophysics**: The bodies in the system are stars or planets in a galaxy, and the governing interaction law is gravitational.
2. **Plasma Physics**: The bodies are ions or electrons, and the governing law is electrostatic.

[1] We use the term "body" and "particle" interchangeably in this paper.

3. **Molecular Dynamics**: The bodies in this case are molecules or atoms, usually in fluid state. Many of the forces between atoms decay fast enough with distance that long-range interactions can be ignored. The simulation of polar fluids, however, introduces a Coulombic interaction term as well, so that long-range interactions must be considered to study some important properties (such as dielectric properties, for example).
4. **Fluid Dynamics**: The vortex blob method [6] for solving the Navier-Stokes equations requires the interactions among vortex blobs, where the long-range interaction law is Coulombic.
5. **Boundary Value Problems**: Integral equations resulting from boundary value problems can be solved rapidly by N-body methods, where is the number of nodes in the discretization of the boundary [21].
6. **Numerical Complex Analysis**: Many problems in this field can be reduced to computing a Cauchy integral, which can itself be viewed as equivalent to an electrostatic problem.
7. **Computer Graphics**: The dominant computation in solving global illumination problems with radiosity techniques has been the ^{2} computation of the light transport between all pairs of elements that the scene is decomposed into. A recent algorithm [16], which we study in this paper, uses a hierarchical N-body technique to compute these interactions in time.

3 Choice of Applications

In this section, we discuss the problems and solution methods we choose to study in this paper, and the coverage of hierarchical N-body methods that they provide.

Classical N-body Problems The first three domains in the list presented above are classical N-body problem domains. In these cases, the system being simulated actually consists of physical entities (planetary bodies, electrons, molecules, etc.). The same hierarchical methods are applicable to all these domains, and their characteristics relevant to partitioning and scheduling are similar across domains as well. The simulation of galaxies under gravitational force laws is the domain that hierarchical methods have been used most widely in so far, and we use it as being representative of nonuniform classical domains.

Several hierarchical methods have been proposed to solve classical N-body problems [2, 18, 3, 15, 29, 7]. The most widely used and promising among these are the Barnes-Hut method [3] and the Fast Multipole Method [15]. Between them, these two methods also capture all the important characteristics of hierarchical methods for classical N-body problems (see Section 4.1.2). By using these methods to solve our galactic problem, we therefore obtain good coverage of hierarchical methods for classical problems.

Other Problems The fourth through sixth domains in the list use the same hierarchical methods as the classical domains, and are expected to have similar characteristics. The seventh domain, the radiosity method from computer graphics, has very different structure and characteristics than the classical problems. Including it as the third application in our study allows us to demonstrate that the partitioning/scheduling approaches which work for the methods used in classical problems do not work for all applications of the hierarchical N-body approach.

4 The Problems and Algorithms

Let us now examine the particular applications we study and the hierarchical methods they use. We first present the gravitational problem, and the Barnes-Hut and Fast Multipole methods that our applications use to solve it. Then, we describe the radiosity application. In every case, we discuss the available parallelism and the levels at which we exploit it.

4.1 The Classical, Gravitational Problem

Our example classical problem is a galactic simulation, which studies the evolution of a system of particles (a galaxy or set of galaxies) under the influence of Newtonian gravitational attraction. A particle is modeled as a point mass.

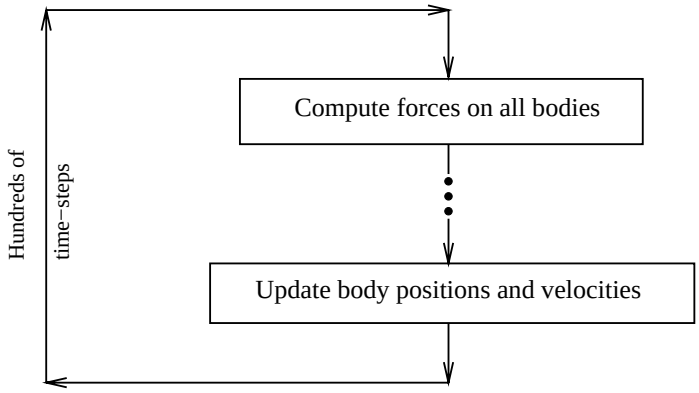


Figure 3: Top-level flowchart of the classical N-body problem.

The time period for which the physical system's evolution is studied is discretized into hundreds or thousands of time-steps for the purpose of computer simulation (Figure 3). Every time-step involves several phases of computation, such as computing the forces acting on each particle and updating particle properties such as position and velocity. The phase which computes the interactions (forces) among particles is by far the most time-consuming in typical applications, and it is this phase that hierarchical methods speed up.

All the hierarchical methods for classical problems first build a tree-structured, hierarchical representation of physical space, and then compute interactions by traversing the tree. This tree representing physical space is the main data structure in both the Barnes-Hut and Fast Multipole methods. The root of the tree represents a space cell containing all the particles in the system. The tree is built by recursively subdividing space cells until some termination condition, usually specified as the maximum number of particles allowed in a leaf cell, is met. In three dimensions, every subdivision of a cell results in the creation of eight equally-sized children cells, leading to an octree representation of space; in two dimensions, a subdivision results in four children leading to a quadtree. Let us now describe the methods and their use of the trees in some detail.

4.1.1 The Barnes-Hut Method

In the Barnes-Hut method, the force-computation phase within a time-step is expanded into three phases, as shown in Figure 4. Let us describe the resulting phases in a time-step.

1. **Building the tree**: The current positions of the particles are first used to determine the dimensions of the root cell of the tree (since particles move between time-steps, the bounding box or root cell that contains the entire distribution may change and has to be recomputed every time-step). The tree is then built by adding particles one by one into the initially empty root cell, and subdividing a cell into its eight children as soon as it contains more than a single particle (the termination condition for the Barnes-Hut method). The result is a tree whose internal nodes are space cells and whose leaves are individual particles. Empty cells resulting from a cell subdivision are ignored. The tree (and the Barnes-Hut algorithm) is therefore adaptive in that it extends to more levels in regions that have high particle densities. Although we actually use a three-dimensional Barnes-Hut problem, Figure 5 shows a small two-dimensional example domain and the corresponding quadtree. Since particles are being loaded into an initially empty tree, and since the expected height of the tree when the particle is inserted is , the expected computational complexity of this phase is 1 ! .

2. **Computing cell centers of mass**: Once the tree is built, the next step is to compute the centers of mass of all internal cells, since cells are approximated by their centers of mass when permitted by the hierarchical

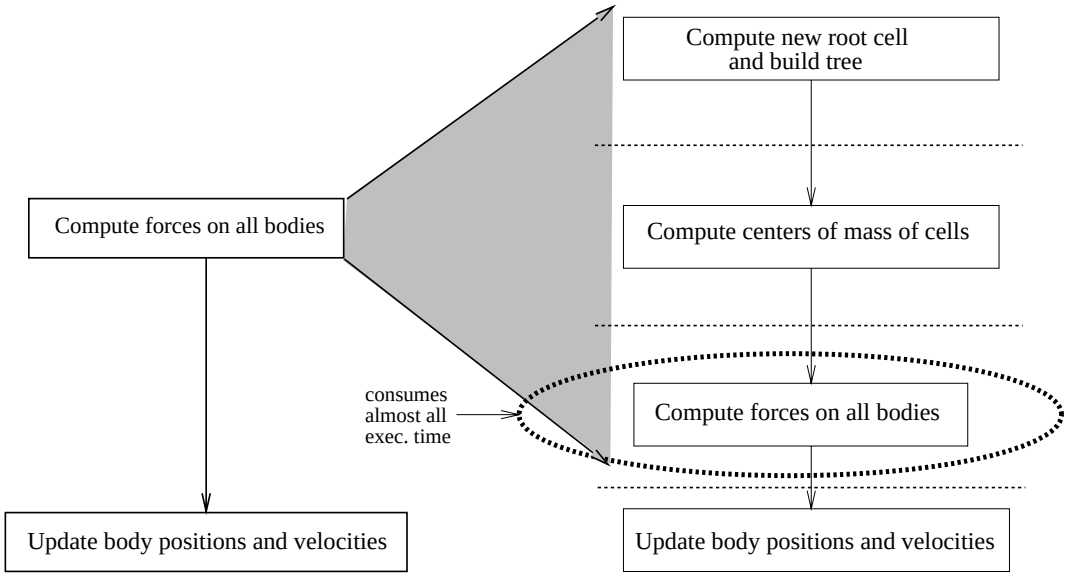
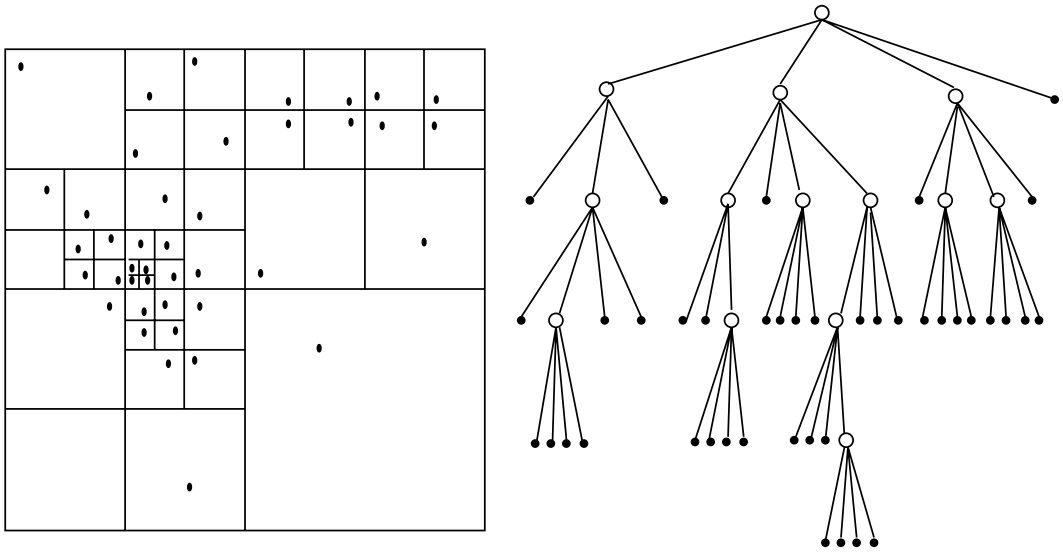


Figure 4: Flowchart of a time-step in the Barnes-Hut application.



(a) The Spatial Domain (b) Quadtree Representation

Figure 5: A two-dimensional particle distribution and the corresponding quadtree.

force-computation algorithm (see below). An upward pass is made through the tree, starting at the leaves, to compute the centers of mass of internal cells from the centers of mass of their children. The expected number of cells in the tree, and hence the expected computational complexity of this phase, is .

3. **Computing Forces**: The tree is then used to compute the forces acting on all the particles. The force-computation phase consumes well over 90% of the sequential execution time in typical problems, and is described in detail below.

4. **Updating Particle Properties**: Finally, the force acting on a particle is used to update such particle properties as acceleration, velocity and position. The computational complexity of this phase is .

A fifth, partitioning, phase is added in our parallel implementation. Let us look at the force-computation algorithm in more detail.

The Force Computation Algorithm In the Barnes-Hut method, the tree is traversed once per particle to compute the net force acting on that particle. The force-computation algorithm for a particle starts at the root of the tree and conducts the following test recursively for every cell it visits: If the cell's center of mass is far enough away from the particle, the entire subtree under that cell is approximated by a single "particle" at the cell's center of mass, and the force that this center of mass exerts on the particle is computed; if, however, the center of mass is not far enough away from the particle, the cell must be "opened" and each of its subcells visited. A cell is determined to be far enough away if the following condition is satisfied:

$$\frac{l}{d} < \theta \qquad (1)$$

where l is the length of a side of the cell, d is the distance of the particle from the center of mass of the cell, and θ is a user-defined parameter used to control the accuracy of the computed forces. In this way, a particle traverses more levels of those parts of the tree which represent space that is physically close to it, and groups particles at a hierarchy of length scales. The complexity of the force-computation phase scales as $\frac{1}{\theta^2} \log n$ for realistic values of n [17].

The Available Parallelism Each of the phases in a time-step can be executed internally in parallel. We do not exploit parallelism across phases or time-steps explicitly, except to avoid synchronizing between phases when possible.[2] The natural unit of parallelism in all phases is a particle, except in computing the cell centers of mass, where the natural unit is a cell.

The tree-building and center of mass computation phases require both interprocessor communication and synchronization. In the tree-building phase, different processors may simultaneously try to modify the same part of the tree, so that cell-level mutual exclusion is required to preserve correctness. In the center of mass computation phase, a processor computing the center of mass of a cell has to wait until the centers of mass of all the cell's children are computed, which requires cell-level event synchronization to preserve dependencies.

The force-computation phase requires communication but not synchronization. While the force computation for a particle requires position and mass information from other particles and cells, that information is not modified during the force-computation phase (positions are modified only later on in the update phase, and masses are not modified at all).

Finally, the work for a particle in the update phase is entirely local to that particle and requires neither communication nor synchronization.

4.1.2 The Fast Multipole Method

The Fast Multipole Method (FMM) also uses a recursive decomposition of the computational domain into a tree structure, as well as a strategy of approximating cells of the tree by equivalent single "particles" when they are far enough away. In this method, a cell is considered far enough away or "well-separated" from another cell b if its separation from b is greater than the length of b. Figure 6 illustrates well-separatedness. Cells and in the figure are well-separated from each other. Cell is well-separated from cell , but cell is not well separated from cell . In fact, both the FMM and the Barnes-Hut method have been shown to satisfy the same recursive set of equations, only using different elementary functions [19].

The primary differences between the two methods are:

> While the Barnes-Hut method directly computes only particle-particle or particle-cell interactions, the FMM also computes interactions between internal cells directly. This is one of the key distinguishing features among hierarchical N-body methods [23], and it is the direct computation of cell-cell interactions that makes the force-computation phase in the FMM rather than for uniform particle distributions.

[2] It is possible to exploit fine-grained dependence information and replace global inter-phase synchronization by finer-grained (cell- or particle-level) synchronization. This is not likely to be useful for the problem and machine sizes we use, but may become more useful on much larger-scale machines.

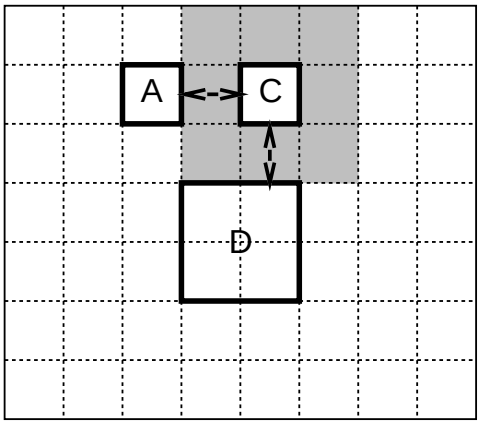


Figure 6: Well-separatedness in the FMM.

Whether a particle/cell is far enough away or "well-separated" from another cell is determined differently. In the FMM, whether two cells are well-separated from each other is determined entirely by their lengths and the distance between them (see above), not by a user-adjustable parameter like the in the Barnes-Hut method.

The FMM approximates a cell not by its center of mass but by a higher-order series expansion of particle properties about the geometric center of the cell. This series expansion is called the *multipole expansion* of the cell. In practice, only a finite number of terms is used in a multipole expansion, and this number determines the accuracy of the representation.

The FMM uses analytic observations about its series expansions to establish tighter bounds on the errors in its approximations. The accuracy of force computation is controlled not by changing which cells are considered far enough away (as in Barnes-Hut), but by changing the number of terms used in the series expansions.

The Barnes-Hut algorithm is inherently adaptive; that is, it does not make any assumptions about the distribution of particles. The FMM, however, comes in two distinct flavors: a simpler version that is well-suited to uniform particle distributions, and a substantially more complex version that adapts to arbitrary distributions. We use the adaptive FMM in this paper.

While the mathematics of the Barnes-Hut algorithm are the same in three dimensions as they are in two, the FMM uses a different, more complex mathematical foundation in three dimensions [14]. The new mathematics substantially increases the constant factors in the time-complexity expression, so that the three-dimensional FMM is far more expensive than the two-dimensional FMM. The structures of the two- and three-dimensional algorithms are the same, however, as are the issues in parallelization. Since a sequential version of the three-dimensional adaptive FMM had not been implemented at the time of this work, we use the two-dimensional adaptive FMM.

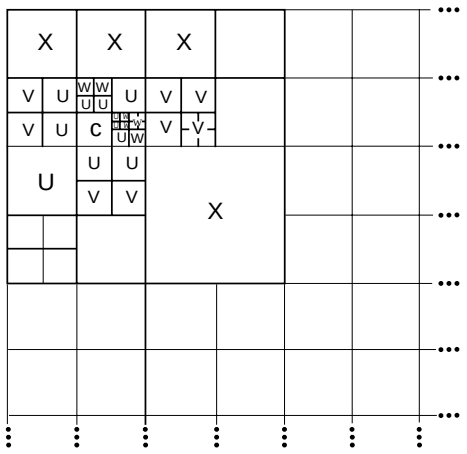
The FMM is far more complicated to program than the Barnes-Hut method.

The efficiency of the FMM is improved by allowing a larger maximum number of particles per leaf cell in the tree than the single particle allowed in the original Barnes-Hut method. We allow a maximum of 40 particles per leaf cell, as Greengard suggests [14], unless otherwise mentioned. Thus, both the leaves and the internal nodes of the tree represent space cells in the FMM, with the leaves directly containing particles.

The phases in a time-step are essentially the same as in the Barnes-Hut application: (i) building the tree, (ii) computing multipole expansions of all cells about their geometric centers in an upward pass through the tree, (iii) computing forces, and (iv) updating particle properties. In fact, three of the phases—all but force-computation—are identical in structure to the corresponding phases in Barnes-Hut. Let us examine the force-computation phase of the adaptive FMM in more detail.

The Force Computation Algorithm For efficient force calculation, every cell (be it leaf or internal) divides the rest of the computational domain into a set of lists of cells, each list containing cells that bear a certain

spatial relationship to . The lists are described in Figure 7. The first step in the force-computation phase is to construct these lists for all cells. Then, the interactions of every cell are computed with the cells in its lists. The hierarchical nature of the algorithm and list-construction ensures that no cell ever computes interactions with cells that are well-separated from its parent (the blank cells for cell in Figure 7). Interactions with these distant cells are accounted for by 's ancestors at higher levels of the tree. Details of the different types of interactions, beyond those indicated in Figure 7, are not important for our purposes and can be found in [14]. What is relevant is that the types of interactions computed with cells in different lists are different.



List U (leaf cells only): all leaf cells adjacent to C.
These cells cannot be approximated in any way .

List V (all cells): Children of the colleagues of C's parent that are well−separated from C.
These are well−separated from C but not from C's parent.

List W (leaf cells only): Descendants of C's colleagues whose parents are adjacent to C, but which are not themselves adjacent to C.
Multipole expansions of these cells can be evaluated at particles in C; their parents' multipole expansions cannot; and their expansions cannot be translated to local expansions about C's center

List X (all cells): Dual of W list: cells such that C is in their W list.
Their multipole expansions cannot be translated to C's center, but their individual particle fields can.

Figure 7: Interaction lists for a cell in the adaptive FMM.

Internal cells compute interactions with only two lists (lists and in Figure 7). The results of these interactions are stored as *local expansions*[3] in the internal cells. Leaf cells compute these cell-cell interactions as well, but also compute interactions of their particles with other particles (in list) and cells (in list), storing the resulting forces directly with their particles in the leaf cells. When all the list interactions have been computed, the influences of distant cells on a leaf cell's particles are contained in the local expansions of its ancestors. These influences are propagated to the leaf cells by transferring local expansions from parents to children in a downward pass through the tree. Finally, the local expansions of leaf cells are evaluated on their particles, and the resulting forces added to those already computed for the particles.

For the galactic simulations we have run, almost all the sequential execution time is spent in computing the list interactions. While the breakdown of time is dependent on the particle distribution, in our nonuniform galactic simulations the majority of this time (about 70% for typical problems) is spent in computing list interactions, next list interactions (about 20%) and finally the and list interactions (about 10%). Building the tree and updating the particle properties take less than 1% of the time in sequential implementations. As in the Barnes-Hut application, the complexities of the tree-building phase and the upward pass are and , respectively. We show in [24] that the asymptotic complexity of the list interaction phases is not as simple as the ^{3} claimed in [14], where is the number of terms used in the multipole and local expansions. If the analysis in [14] is performed correctly, the complexity is ^{2} , where is the number of levels in the tree. This number of levels can be viewed as either a large constant (certainly larger than), or as a quantity that depends on (in a like fashion). In practice, however, the main determinant of execution time is the number of cells in the tree, which is roughly for most practical problems. Thus, the overall execution time usually scales quite linearly with the number of particles in practice.

The Available Parallelism Parallelism is exploited in all phases of the application, just as in the Barnes-Hut case, and in all steps of the force-computation phase. The key difference from Barnes-Hut is that the

[3]The local expansion of a cell is another series expansion, which represents the influence exerted on it by all cells that are far enough away to be approximated by their multipole expansions.

unit of parallelism is a cell rather than a particle. From the above description, a cell is clearly the natural unit of granularity in the force-computation algorithm: Partitioning in terms of particles would introduce many unnecessary overheads, particularly since different processors would be likely to own particles in the same cell.

The tree-building and multipole expansion phases, as well as the downward pass in the force-computation phase, clearly require both communication and synchronization. The list interactions, however, involve read-only communication and can be executed without synchronization. Finally, the update phase can be performed without synchronization or communication.

Clearly, both the Barnes-Hut and FMM applications afford abundant parallelism.

4.2 The Radiosity Application

Computing the global illumination in a scene is a critical problem in computer graphics. The two dominant, and very different, approaches to solving this problem are the ray-tracing and radiosity methods. The radiosity method [8], which is view-independent and based on the physics of light transport, has been most successful in producing realistic computer-generated images of complex scenes. Since this method accounts for both direct illumination by light sources and indirect illumination through multiple reflections, it is an essential tool in fields such as architectural design which require "photo-realistic" images.

The radiosity of a surface is defined as the light energy leaving the surface per unit area. Given a description of a scene, the radiosity problem is to compute the radiosities of all surfaces, and hence the equilibrium distribution of light in the scene. In the traditional radiosity approach, the large polygonal surfaces that describe the scene (such as walls or desktops) are first subdivided into small enough elements that the radiosity of an element can be approximated as being uniform. The radiosity of an element can be expressed as a linear combination of the radiosities of all other elements . The coefficients in the linear combination are the "form factors" between the elements, where the form factor from element to element is the fraction of light energy leaving element which arrives at element . This leads to a linear system of equations, which can be solved for the element radiosities once all the form factors are known. Computing the form factors between all pairs of elements has ^{2} time complexity. However, the form factor decays with the inverse square of the distance between elements, which allows the use of an hierarchical algorithm.

Although the hierarchical radiosity algorithm is based on the same basic approach as the algorithms we have seen for the classical N-body problem, it is very different in several ways. First, the starting point in the hierarchical radiosity algorithm is not the finally undivided *elements* or "bodies", but the much smaller number of large *input polygons* (walls, desktops) that initially represent the scene. These input polygons are hierarchically subdivided as the algorithm proceeds, generating a quadtree per polygon (see Figure 8). In fact, the number of resulting leaf-level elements ( ) in this forest of quadtrees is not even known *a priori*. We use the term *patch* for any intermediate pieces of surface in the quadtree, including the input polygons; it is the equivalent of a "cell" or "internal cell" in the classical N-body applications[4].

The second difference is that the interaction law in radiosity is not simply a function of the interacting entities and the distance between them. The interaction is the transport of light between patches, which can be occluded by other patches, making it necessary to compute the visibility between patches as part of determining form factors. Third, while the N-body computations proceed over hundreds of time-steps, the radiosity computation proceeds over only a small number of iterations until the total radiosity of the scene converges. And finally, unlike the particles in the N-body applications, the patches in radiosity do not move with time. Patches may, however, get further subdivided in successive iterations. These differences are what cause very different partitioning/scheduling techniques to be required for the radiosity application, as we shall see.

[4]We sometimes use the term patch to include leaf-level elements in our description of the algorithm.

4.2.1 The Sequential Algorithm

The hierarchical radiosity [16] algorithm proceeds as follows. The input polygons that comprise the scene are first inserted into a binary space partitioning (BSP) tree [11] to facilitate efficient visibility computation between a pair of patches. (The BSP tree and its use in visibility computation are described in Appendix A). Every input polygon is initially given a list of other input polygons which are potentially visible from it, and with which it must therefore compute interactions. Then, polygon radiosities are computed by an iterative algorithm that may be described at a high-level as follows:

1. For every polygon, compute its radiosity due to all polygons on its interaction list, subdividing it or other polygons hierarchically as necessary (see Figure 9, in which binary trees are shown instead of quadtrees for clarity, and only one polygon's interactions lists are shown).
2. Add all the area-weighted polygon radiosities together to obtain the total radiosity of the scene, and compare it with that of the previous iteration to check for convergence within a fixed tolerance. If the radiosity has not converged, return to step 1. Otherwise, go to step 3.
3. Smooth the solution for display by computing the radiosities at the vertices of the leaf-level elements. The radiosity of a vertex is computed by interpolating the radiosities of the patches adjacent to that vertex. Since this phase is performed only once at the end of the application, and since it is a very simple phase from the viewpoint of parallelism, we do not discuss it any further.

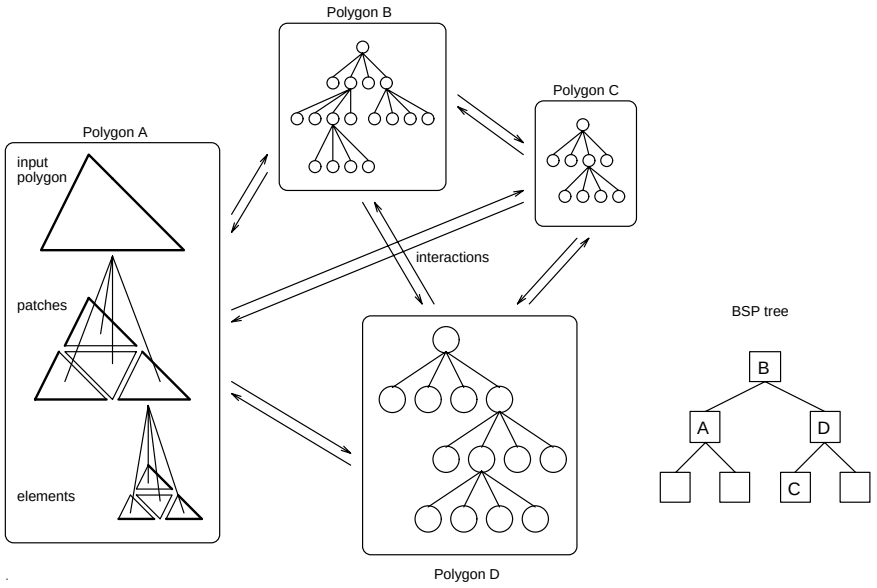


Figure 8: High-level picture of polygons and interactions in the radiosity application.

Every input polygon can be viewed as the root of a quadtree that it will be decomposed into (see Figure 8). Thus, the "tree" data structure used in this method is not a single tree that represents a spatial decomposition of the computational domain, but rather a forest of quadtrees representing individual polygons. The roots of these quadtrees (the input polygons themselves) are in turn the leaves of the BSP tree used for visibility testing (see Appendix A). In every iteration, each of the (not yet fully subdivided) quadtrees is traversed depth-first starting from its root. At every quadtree node visited in this traversal, interactions of the patch[5] (patch i, say) at that node are computed with all other patches, j, in its interaction list (see Figure 8). The interaction between two patches involves computing both the visibility and the unoccluded form factor between them, and multiplying the two to obtain the actual form factor ( ). The actual form factor ( ) multiplied by the radiosity ( ) of patch j yields the magnitude of light transfer from j to i. Both the unoccluded form

[5] A "patch" here could be an input polygon, an intermediate patch, or an element.

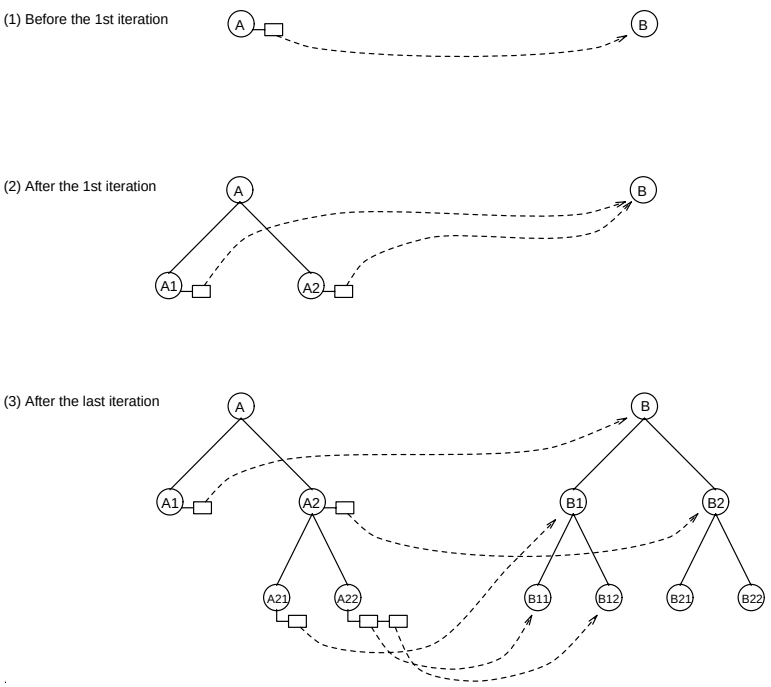


Figure 9: Hierarchical interactions in the radiosity application.

factor and the visibility are computed approximately, introducing an error in the computed ( ). An estimate ( ) of this error is also computed. If the magnitude of error in the light-transfer ( ) is larger than a user-defined tolerance, the patch with the larger area is subdivided to compute a more accurate interaction. Children are created for the subdivided patch in its quadtree if they do not already exist.

If the patch being visited (patch *i*) is subdivided, patch *j* is removed from 's interaction list and added to each of 's children's interaction lists. If patch *j* is subdivided, it is replaced by its children on patch *i*'s interaction list. Figure 9 shows an example of this hierarchical refinement of interactions. Patch *i*'s interaction list is completely processed in this manner before its children are visited in the tree traversal. During this downward traversal, the radiosities gathered at the ancestors of patch *i* and at patch *i* itself are accumulated and passed to *i*'s children. Finally, after the traversal of a quadtree is completed, an upward pass is made through the tree to accumulate the area-weighted radiosities of every patch's descendents into the radiosity of that patch. Thus, the radiosity of a patch is the sum of three quantities: the area-weighted radiosities of its descendants, the radiosity it gathers in its own interactions, and the radiosities gathered by its ancestors.

If the radiosity of the scene has not converged, the next iteration performs similar traversals of all quadtrees starting at their roots. At the beginning of an iteration, the interaction list of a patch in any quadtree is exactly as it was left at the end of the previous iteration: containing the patches with whom its interaction did not cause a subdivision.

4.2.2 The Available Parallelism

Parallelism is available at three levels in this application: across input polygons, across the patches that a polygon is subdivided into, and across the interactions computed for a patch. The construction of the BSP tree is done sequentially, since the order in which patches are inserted in the BSP tree affects the total running time. Besides, construction of this tree takes less than 0.1% of the uniprocessor execution time of the program, since only the original input polygons are inserted in the tree.

Parallelism across input polygons can be exploited in all other phases of the algorithm. Parallelism

across patches within a quadtree (polygon) can be exploited both in the tree traversals to compute radiosities, and in smoothing radiosities. Finally, the interactions for a patch with different patches in its interaction list can be computed in parallel. Since the quadtrees are built as the computation proceeds, all these levels of parallelism involve both communication and synchronization among processors. We will describe implementations that exploit different levels of parallelism in Section 10.

Before we present our results for the three applications, let us first describe the execution platforms that we use, the issues in effective parallelization on these platforms, and our experimental methodology.

5 The Execution Platforms

Figure 10 shows the generalized shared address space multiprocessor that we assume in our parallelization. The multiprocessor consists of a number of processing nodes, connected by some general interconnection network. Every processing node comprises a processor, a cache and an equal fraction of the total physical (main) memory on the machine. That is, the address space is shared but main memory is physically distributed. Caches are kept coherent by a hardware mechanism. We use two instantiations of this generalized multiprocessor in our experiments: the Stanford DASH Multiprocessor—a high-performance research machine—and a simulated multiprocessor.

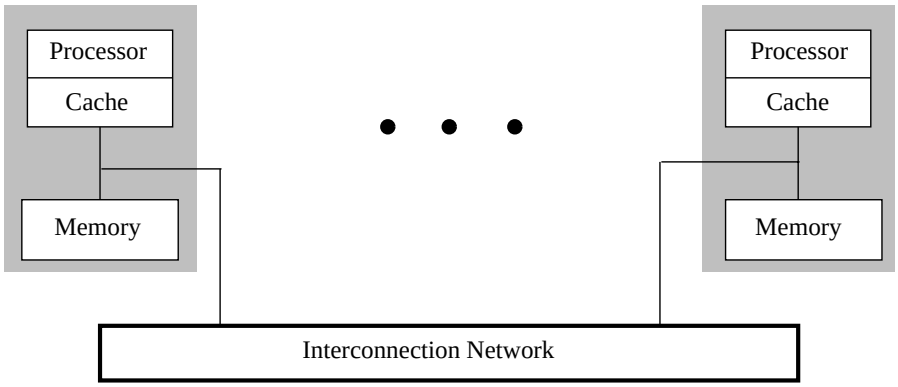


Figure 10: The simulated multiprocessor architecture.

The Stanford DASH Multiprocessor The DASH machine [20] has 48 processors organized in 12 clusters.[6] A cluster comprises 4 MIPS R3000 processors connected by a shared bus, and clusters are connected together in a mesh network. Every processor has a 64KB first-level cache memory and a 256KB second-level cache, and every cluster has an equal fraction of the physical memory on the machine. All caches in the system are kept coherent in hardware using a distributed directory-based protocol.

The Simulated Multiprocessor While an existing machine such as DASH has the advantages of being fast and real, it is inflexible in configuration and unable to track program behavior statistics of interest to us (inherent communication in the program, for example). To overcome these limitations, we also perform experiments on an event-driven simulator of an idealized shared-address-space multiprocessor [13]. The simulated multiprocessor looks exactly like that described in Figure 10, with the simple, three-level, nonuniform memory hierarchy. The timing of a simulated processor's instruction set is designed to match that of the MIPS R3000 CPU and R3010 floating point unit. Access latencies in the memory system are assumed to be as follows. Hits in the issuing processor's cache cost a single processor cycle. Read misses that are satisfied in the local memory unit stall the processor for 15 cycles, while those that are satisfied in a remote cluster (cache or memory unit) stall the

[6]The prototype actually has 64 processors in 16 clusters, but is broken up into two separate machines in usual operation.

processor for 60 cycles. Since write miss latencies can be quite easily hidden by software/hardware techniques, the corresponding numbers for write misses are 1 and 3 cycles, respectively. By using constant miss latencies, we ignore contention in various parts of the system. However, the constant latencies keep our machine model simple and are appropriate for the characteristics we want to study.

6 Issues in Effective Parallelization

In general, there are six major sources of overhead that inhibit a parallel application from achieving effective speedups over the best sequential implementation: inherently serial sections, redundant work, overhead of managing parallelism, synchronization overhead, load imbalance and communication. As in many scientific applications, the first four are not very significant in the applications we consider here. Let us therefore define the scope of our treatment of the last two, and the goals of our parallel implementations.

Load Balancing: The goal in load balancing is intuitive: Tasks[7] in the workload should be assigned to processors in such a way that the amount of time any processor spends waiting for other processors to reach synchronization points is minimized.

Data Locality: As we have seen, modern multiprocessors are built with hierarchical memory systems, in which processors have faster access to data that are closer to them in the hierarchy. Exploiting the locality in data referencing that an application affords can increase the fraction of memory references that are satisfied close to the issuing processor, and hence improve performance. Many of the aspects of locality in a multiprocessor are no different than on a uniprocessor, since the part of the memory system that is within a processing node is very similar to a uniprocessor memory hierarchy. The key difference in multiprocessors is that they extend the hierarchy by introducing communication among processing nodes, which is very expensive. For this reason, we focus our discussion of locality primarily on reducing communication by scheduling tasks that access the same data on the same processor. By doing this, we ignore several more detailed locality issues[8] as well as the impact of locality in a network topology[9]. We also do not address the impact of allocating data appropriately in physically distributed main memory in this paper, primarily because this is both very difficult to do for these applications and because experiments we have performed show that the important form of locality is reuse or temporal locality, and that data distribution does not help performance very significantly [24]. Despite our focus on communication, however, we do make all reasonable efforts to exploit locality within a node effectively.

Since there is often a tension between load balancing and data locality, effective partitioning and scheduling techniques must strive to find a good compromise between these goals. Unfortunately, several characterisitics of hierarchical N-body applications make it challenging to attain these goals simultaneously:

> The physical domain being simulated is typically highly *nonuniform*, which has implications for both load balancing and communication:
>
> – The work done per unit of parallelism (particle or cell) is not uniform, and depends on the distribution of particles in space. Equally sized spatial partitions or those with equal numbers of particles therefore do not have equal work associated with them.[10]
> – The natural communication is fine-grained, long-range, and relatively unstructured except for its underlying hierarchical nature. The amount of communication is nonuniformly distributed across particles/cells and hence across processors.
>
> In the classical applications, the amount of communication falls off with distance equally in all directions. To reduce communication frequency and volume, therefore, a processor's partition should be spatially contiguous and not biased in size toward any one direction.

[7] A task in this paper refers to a unit of work which is treated as being indivisible from the scheduling point of view.

[8] Such as prefetching data with long cache lines or in software, false-sharing of data and cache mapping collisions.

[9] Locality in the network topology is not a significant issue in real machines until the number of processing nodes becomes very large, since the time for a packet to get in and out of the network dominates the time due to the number of network hops traversed. Also, the impact of network locality is diluted if other forms of locality (in the cache or local memory) are exploited effectively, and network locality is in any case orthogonal to these other forms of locality.

[10] In the problems we have run, the standard deviation in the distribution of work across units of parallelism (particles, say) was usually several times the mean work per unit.

The *dynamic nature* of the applications—together with the nonuniformity—causes the distribution of particles and hence the work and communication distributions to change slowly across time-steps. There is no steady state. This means that static partitioning schemes are unlikely to work well.

The different phases in a time-step have different relative amounts of work associated with particles/cells, and hence different preferred partitions if viewed in isolation.

7 Experimental Methodology

Let us now describe the metrics we use to compare different partitioning/scheduling techniques, and the organization of our experiments.

7.1 Comparison Metrics

The primary metric we use for comparing different implementations is the speedup they yield over the same sequential program. We present speedups obtained on the DASH multiprocessor in all cases, and also those obtained on the simulator in cases where we could simulate large enough problems that speedups are meaningful. In addition to speedups, we also present results that separately compare the load balancing and communication behavior of different schemes. These results are obtained on the simulator, and we have corroborated their trends with the MTOOL [12] performance debugger on DASH.

We compare load balancing behavior by measuring the time that processes spend waiting at synchronization points. In comparing communication behavior, we focus on inherent communication in the program that implements a given scheme, rather than on how these inherent program characteristics interact with the details of a particular memory system configuration. For this reason, we simulate infinite per-processor caches with a small cache line size (8 bytes).

In general, cache misses in a multiprocessor can arise from four sources: (i) *communication*, both inherent and due to false sharing of cache or memory blocks; (ii) replacements of useful data due to limited cache *capacity*; (iii) replacements due to cache mapping *conflicts* caused by an associativity smaller than the cache size; and (iv) *cold-start* effects. By using infinite caches, we eliminate capacity and conflict misses. By using a small cache line, we virtually eliminate false-sharing misses. Cold-start misses are not significant in these applications in practice, since the applications are typically run for hundreds of time-steps. Since our experiments run the applications for only a very small number of time-steps, however, we explicitly ignore the cold-start period by resetting the simulator statistics after two time-steps (by which point the program has settled down and the partitioning scheme has fully taken effect). Thus, our measurement environment essentially eliminates all misses other than those due to inherent communication (other artifactual sources of communication in a multiprocessor, such as data placement in distributed main memory, are not significant either after the cold-start period with infinite caches in these applications [28]). Our metric for comparing inherent communication in this environment is the average number of cache misses per 1000 processor busy cycles.

One implication of our use of infinite caches is that performance results obtained with the simulator may be biased in favor of partitioning schemes with poor locality characteristics. The reason is that schemes with poor locality may have larger active working sets than schemes with good locality. On a real machine, schemes with good locality may therefore have an important performance advantage if the finite caches on the machine are large enough to accommodate the working sets of these schemes but not the working sets of schemes with poor locality. Infinite caches do not capture this effect. However, this is not a very significant issue in our applications since the important working sets are small [22]. Besides, infinite caches are better at measuring inherent communication, which is what we want to compare using the simulator.

7.2 Organization of Experiments

For each application, we first examine approaches that are conceptually obvious and very easy for a programmer to implement. These approaches use only information that can be obtained statically from the program itself to guide their decisions about load balancing and data locality. They do not invoke any special mechanisms to account for characteristics of the input domain, and are the techniques that a compiler or automatic scheduler would use if it knew the appropriate parallelism. After understanding the limitations of these techniques for an application, we move on to study the effectiveness of more sophisticated partitioning techniques. Let us now examine the individual applications.

8 The Barnes-Hut Application

8.1 The Straightforward Approaches

As described in Section 4.1.1, the sequential Barnes-Hut application is structured in phases, every phase consisting of an outer loop that iterates over particles (or cells, in one phase). Each iteration of these loops performs the work for a particle or cell in that phase, which may involve computing the forces on a particle or inserting a particle into the tree. Two easy partitioning techniques therefore suggest themselves. One is a fully *static* approach, in which every processor is assigned an equal set of iterations in every phase, the same iterations in every phase and time-step as far as possible. The other is a fully dynamic approach, in which processors dynamically obtain loop iterations or groups of iterations until there are none left in the loop. We call this second scheme *loadbalance*. Both of these are schemes that a compiler or automatic scheduler might implement.

Static approaches are appropriate for many scientific applications, in which the work per iteration in parallel loops is uniform and locality in iteration space translates naturally to the desired locality in the problem domain [25, 26].[11] In our hierarchical N-body applications, however, the *static* scheme described above neither guarantees load balancing nor provides data locality. The load imbalance results from the fact that an equal number of particles does not imply an equal amount of work. However, the imbalance may not be severe in practice if the number of particles per partition is large and the nonuniformities in work per particle average out. In fact, the *static* scheme is likely to compromise data locality more than load balancing for realistic problems. Although the scheme provides locality in the loop iteration space—as well as a form of *object locality* by always assigning a processor the same particles—the numbering of particles/cells in the loop iteration space has nothing to do with their positions in space and hence with the *physical locality* that is most important to performance. A processor's particles are likely to be scattered throughout the domain, and every processor is therefore likely to reference the entire Barnes-Hut tree, which leads to more communication and less data reuse than if physical locality were incorporated and every processor referenced only a fraction of the tree.

The *loadbalance* scheme does not have a load balancing problem, but it does not make any attempt to incorporate data locality. Let us now look at some more sophisticated partitioning techniques that attempt to simultaneously provide load balancing and physical locality, by taking advantage of some insights into the application and the Barnes-Hut method. We first discuss how these techniques incorporate load balancing, and then at how they provide physical locality.

8.2 Load Balancing

Whatever the technique used to provide locality, the relative amounts of work associated with different particles must be known if load balancing is to be incorporated without resorting to dynamic task stealing (which has its own overheads and compromises data locality). In this subsection, we describe how we obtain load balancing by determining the work associated with a particle.

[11] This is true of some uniform N-body applications as well, particularly those that use a fixed cutoff radius in computing interactions and a spatial directory to efficiently find the particles that are within the cutoff radius [26].

There are three problems associated with load balancing in the Barnes-Hut application. First, the relative distribution of work among particles is not the same across the phases of computation (tree-building, force calculation, update, etc.). Different phases might therefore have different preferred partitions. We focus our attention on partitioning the force calculation phase, since well over 90% of the sequential execution time is spent in this phase. Partitions that are appropriate for the force calculation phase are also likely to be appropriate for the tree-building and center-of-mass computation phases, since the three phases share the following important characteristics: (i) more work is associated with particles/cells in denser parts of the domain, and (ii) minimizing interprocessor communication calls for exploiting physical locality. The update phase, which simply calls for an equal number of particles per processor, will not be well load balanced by these partitions. However, the fraction of the execution time spent in this phase is too small to make repartitioning worthwhile, at least for our problem and machine sizes.

The second problem for load balancing is that the distribution of work among particles within the force-computation phase is not uniform (see Section 6). The solution to this problem is to associate a cost with every particle, which reflects the amount of work needed to compute the force on that particle, and use this cost rather than the number of particles as the load balancing metric.

The third problem is that the aforementioned cost of a particle is not known a priori, and changes across time-steps. However, there is a key insight into the application characteristics that allows us to solve this problem. Since classical N-body problems typically simulate physical systems that evolve slowly with time, *the distribution of particles changes very slowly across consecutive time-steps*, even though the change in distribution from the first time-step to the last is often dramatic. In fact, large changes from one time-step to the next imply that the time-step integrator being used is not accurate enough and a finer time-step resolution is needed. Since a particle's cost depends on the distribution of particles, the slow change in distribution implies that a particle's cost in one time-step is a good estimate of its cost in the next time-step.

In the Barnes-Hut application, a good measure of a particle's cost is simply the number of interactions (with other particles or cells) required to compute the net force on that particle. This measure of cost is used by both Salmon [23] and us. The programming and runtime overheads of counting interactions are negligible: A process simply increments a per-particle counter every time it computes an interaction. As we shall see, this work-counting technique is very effective. In fact, it ought to be a powerful technique for load balancing in many nonuniform problems that simulate slowly evolving, natural physical phenomena.

8.3 Providing Physical Locality

To provide physical locality, it is clearly not good enough to partition the domain space statically among processors, since this will lead to very poor load balancing. In this subsection, we describe two partitioning techniques, *orthogonal recursive bisection* and *costzones*, that provide load balancing as well as physical locality. Both techniques use the work-counting mechanism of the previous subsection for load balancing, but they differ in the approach they take to providing physical locality.

8.3.1 Partitioning Space: Orthogonal Recursive Bisection

Orthogonal Recursive Bisection (ORB) is a technique for providing physical locality in a problem domain by explicitly partitioning the domain space [10]. It was first used for hierarchical N-body problems in Salmon's message-passing Barnes-Hut implementation [23]. The idea in ORB partitioning is to recursively divide the computational domain space into two subspaces with equal costs, until there is one subspace per processor (see Figure 11).

To provide load balancing, the cost of a subspace is defined as the sum of the profiled costs of all particles in the subspace. Initially, all processors are associated with the entire domain space. Every time a space is divided, the processors associated with the space are divided equally into two processor subsets, and each subset is assigned to one of the resulting subspaces. The Cartesian direction in which division takes place is usually alternated with successive divisions, although this alternation is sometimes violated to avoid resulting

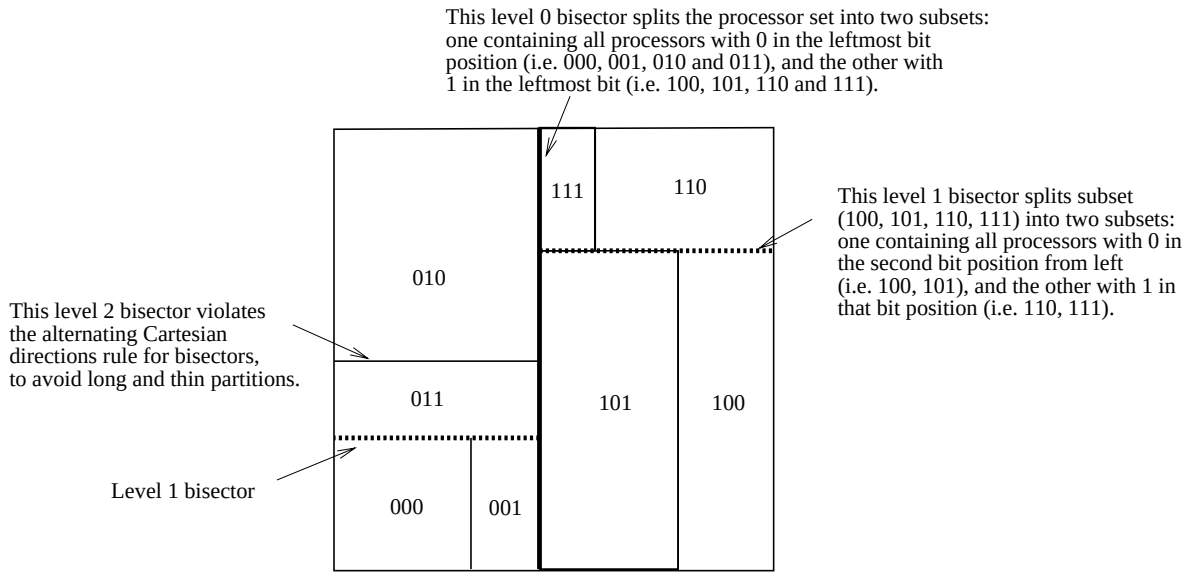


Figure 11: *ORB* partitioning.

in long and thin subspaces. A parallel median finder is used to determine where to split the current subspace in the direction chosen for the split. The worst-case load imbalance per bisection can be made as small as the maximum amount of work associated with a single particle, but the imbalance in the final partitions might be somewhat worse owing to the cumulative effects of successive bisections. A more detailed description of an ORB implementation for the Barnes-Hut problem can be found in [23]. High-level pseudocode describing the algorithm is shown in Figure 12.

Figure 12: High-level description of the *ORB* algorithm.

ORB introduces several new data structures, including a binary ORB tree that is distinct from the Barnes-Hut tree. The nodes of the ORB tree are the recursively subdivided subspaces, with their processor subsets, and the leaves are the final spatial partitions. ORB is nontrivial to implement and debug, and can have significant runtime overhead particularly as the number of processors increases (as we shall see). It also restricts the number of processors used to a power of two, although more complex variants can be developed that don't have this restriction. For these reasons, we develop a simpler partitioning technique that is more flexible and scales better with the number of processors used.

8.3.2 Partitioning the Tree: Costzones

Our *costzones* partitioning technique takes advantage of another key insight into the hierarchical methods for classical N-body problems, which is that *they already have a representation of the spatial distribution encoded in the tree data structure they use*. We can therefore partition the tree rather than partition space directly. In the *costzones* scheme, the tree is conceptually laid out in a two-dimensional plane, with a cell's children laid out from left to right in increasing order of child number. Figure 13 shows an example using a quadtree. The cost of every particle, as counted in the previous time-step, is stored with the particle. Every internal cell holds the sum of the costs of all particles that are contained within it, these cell costs having been computed during the upward pass through the tree that computes the cell centers of mass.

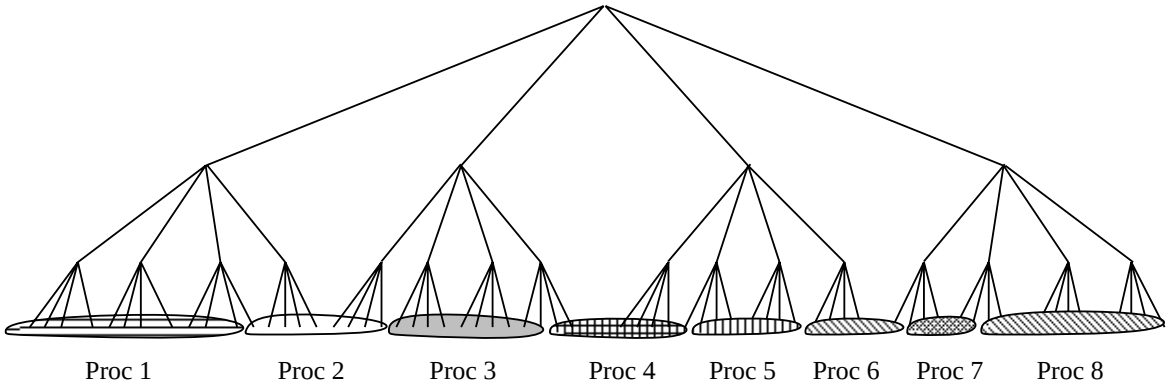


Figure 13: Tree partitioning in the *costzones* scheme.

The total cost in the domain is divided among processors so that every processor has a contiguous, equal range or zone of costs (hence the name *costzones*). For example, a total cost of 1000 would be split among 10 processors so that the zone comprising costs 1-100 is assigned to the first processor, zone 101-200 to the second, and so on. Which cost zone a particle belongs to is determined by the total cost up to that particle in an inorder traversal of the tree. In the *costzones* algorithm, processors descend the tree in parallel, picking up the particles that belong in their cost zone. A processor therefore performs only a partial traversal of the tree. To preserve locality of access to internal cells of the tree in later phases, internal cells are assigned to the processors that own most of their children.

Figure 14: *Costzones* partitioning

The *costzones* algorithm is described in Figure 14. Every processor calls the *costzones* routine with the `Cell` parameter initially being the root of the tree. The variable `cost-to-left` holds the total cost of the bodies that come before the currently visited cell in an inorder traversal of the tree. Other than this variable, the algorithm introduces no new data structures to the program. The algorithm requires only a few lines of code, and has negligible runtime overhead (see the results in Section 8.4).

Child Numbering Issues

The *costzones* technique yields partitions that are contiguous in the tree as laid out in a plane. How well this contiguity in the tree corresponds to contiguity in physical space depends on how the locations of cells in the tree map to their locations in space. This in turn depends on the order in which the children of cells are numbered (when laying them out from left to right in the planarized tree). The simplest ordering scheme—and the most efficient for determining which child of a given cell a particle falls into—is to use the same ordering for the children of every cell. Unfortunately, there is no single ordering which guarantees that contiguity in the planarized tree will always correspond to contiguity in space.

The partition assigned to processor 3 in Figure 15 illustrates the lack of robustness in physical locality resulting from one such simple ordering in two dimensions (clockwise, starting from the bottom left child, for every node). While all particles within an internal cell or subtree are indeed in the same cubical region of space, particles (or subtrees) that are next to each other in the planarized tree may not have a common ancestor until much higher up in the tree, and may therefore not be anywhere near each other in physical space.

There is, however, a simple solution that makes contiguity in the planarized tree always correspond to contiguity in space. In this solution, the order in which children are numbered is not the same for all cells.

Figure 16: *Costzones* with nonuniform numbering.

are eight ways in which a set of siblings can be ordered: four possible starting points, and two possible directions (clockwise and anticlockwise) from each starting point. It turns out that only four of the eight orderings need actually be used. Figure 16(a) shows the four orderings we use in our example, and illustrates how the ordering for a child is determined by the ordering for its parent. The arrow in a cell represents the ordering of that cell's children. For example, if the children are numbered 0, 1, 2, 3 in a counterclockwise fashion starting from the upper right, then in case (1) the children of the top-level cell are ordered 2, 1, 0, 3, and in case (2) the children are ordered 2, 3, 0, 1. The ordering of the children's children are also shown in Figure 16(a).

Figure 16(b) shows the resulting partitions given the same distribution as in Figure 15. All the partitions are physically contiguous in this case. The three-dimensional problem is handled similarly, except that there are now different orderings used instead of 4. Appendix B discusses the three-dimensional case in more detail.

Finally, we note that although the simpler, uniform ordering scheme discussed earlier does result in noncontiguous partitions in practice, this is not much of a performance issue on moderately-sized shared address space machines with caches. Since the number of processors is typically small compared to the number of particles, a processor usually picks up entire large subtrees during *costzones* partitioning. The number of transitions between contiguous regions (subtrees) is small compared to the size of each region, and enough locality is obtained within each region that the transitions are not a problem. We find that the performance differences between uniform and nonuniform ordering are measurable but very small for the problem and machine sizes we use, although they do grow with the number of processors. When presenting performance results in Section 8.4, we use the more robust, nonuniform child ordering method in the partitioning scheme we call *costzones*.

Before we measure the performance of all the schemes we have described (*static*, *loadbalance*, *ORB* and *costzones*), let us introduce a final scheme that helps us separate the effect of data locality from that of load balancing. This scheme, which we call *spacelocality*, uses the same tree traversal technique for providing physical locality as *costzones*, but uses a naive metric—unit cost for every particle—for load balancing.

8.4 Results

We now present performance results for the Barnes-Hut application. The input distribution that we use in our experiments comprises two Plummer model [1] galaxies, slightly offset from each other in each of the three Cartesian directions. The Plummer model is an empirical model of galactic clusters. The density of a cluster is very large near the center and diminishes with distance from the center.

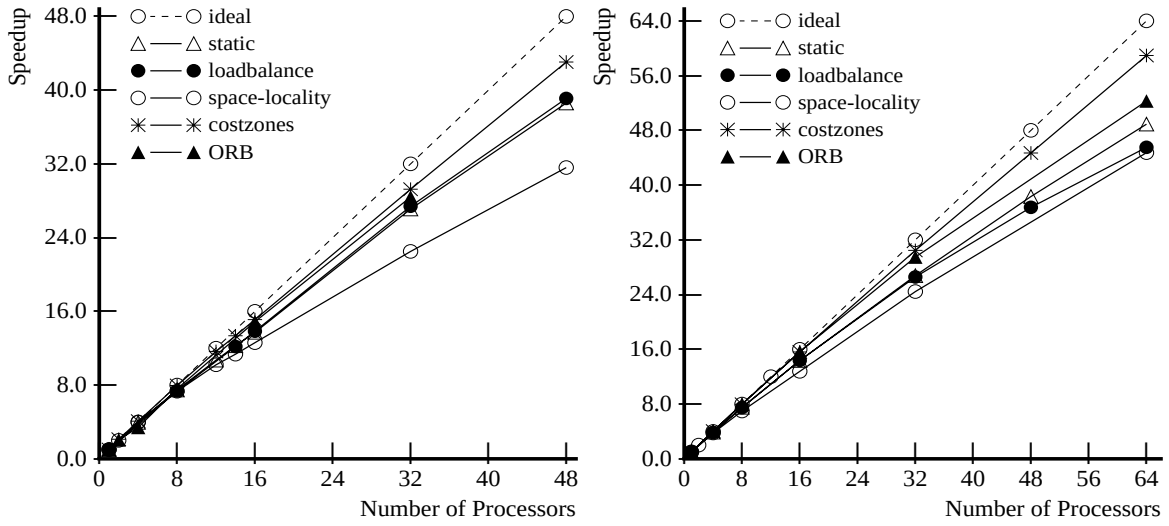
8.4.1 Speedups

Figure 17 shows speedups with 32K particles on the DASH multiprocessor, and with 8K particles on the simulator with infinite caches. The accuracy parameter (see Section 4.1.1) is set to 1.0. Measurements with other reasonable numbers of particles and accuracies show similar relative trends across partitioning schemes. All speedups are relative to a uniprocessor execution of the *static* scheme, which is almost identical to a sequential program execution. (48-processor speedups are not shown for ORB since 48 is not an integer power of 2 as required by the ORB algorithm.)

At least up to 16 processors, all schemes yield good speedups for both problem sizes, although the differences between them are clear. These differences grow as more processors are used. To facilitate the comparison of different schemes, let us look separately at their load balancing and communication behavior, using results obtained from the simulator.

8.4.2 Load Balancing and Communication

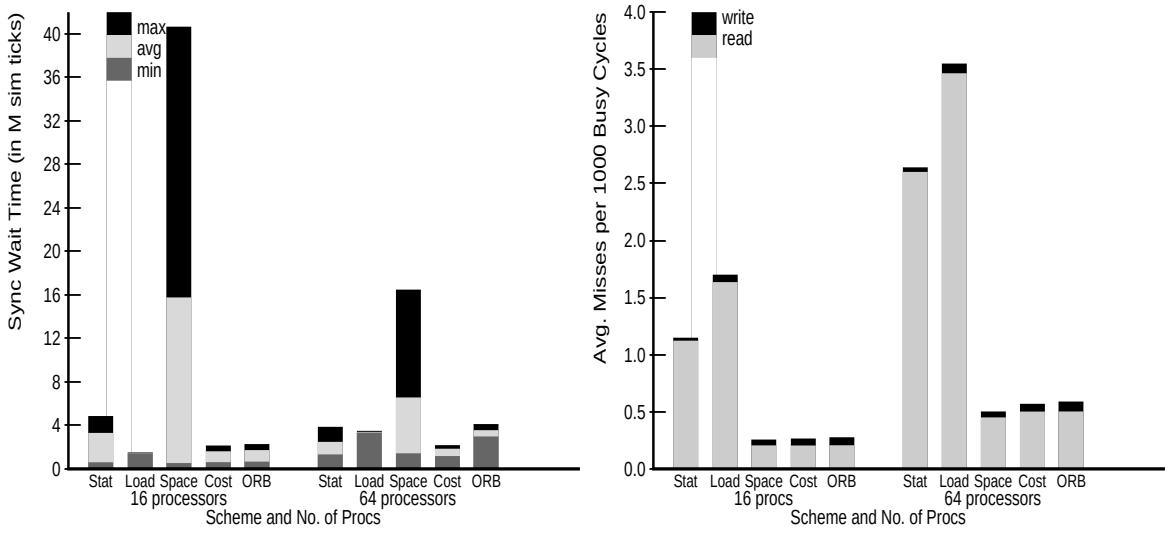
For load balancing, Figure 18(a) shows the amount of time processors spend waiting at synchronization events (locks and barriers) under the different schemes with 8K particles. The minimum, average and maximum time



(a) 32K Particles on DASH

(b) 8K Particles on Simulator

Figure 17: Speedups for the Barnes-Hut application.



(a) Synchronization Wait Times

(b) Communication

Figure 18: Barnes-Hut behavioral statistics on the simulator (= 8k, = 1.0).

(in millions of simulated processor clock ticks) over all processors are shown for 16 and 64 processor executions. The total simulated execution time for the best (*costzones*) scheme was 68 million ticks with 16 processors, and 18 million ticks with 64 processors.

For communication, Figure 18(b) shows the average number of misses in the infinite cache per thousand processor busy cycles per processor. Let us now relate the information in these figures to the performance of the different schemes in Figures 17.

The Static and Loadbalance Schemes: Figures 18(a) and 18(b) verify our expectations about the *static* and *loadbalance* schemes that a compiler or automatic scheduler might implement. For example, Figure 18(a) shows that the load balancing behavior of the *static* scheme is not bad since the workload characteristics average out over the large number of particles in every partition (of course, this is not guaranteed for all distributions).

And Figure 18(b) shows that although the communication behavior of the *static* scheme is better than that of *loadbalance*, the absence of physical locality makes this communication behavior quite poor relative to that of the best schemes.

It is interesting that for the problem sizes used here (which are in the small to medium range of the problem sizes used by astrophysicists on high-performance computers today), the *static* and *loadbalance* schemes are able to obtain quite good performance with these small numbers of processors, despite their shortcomings in incorporating locality. This is because the amount of communication relative to computation is small in absolute terms for these ratios of problem size to number of processors (even for the relatively small problem in Figure 18(b)). For example, although the *loadbalance* scheme has a communication to computation ratio that is 7 times that of the *costzones* scheme for the problem depicted in Figure 18(b) with 64 processors, its ratio is still less than 4 misses per 1000 processor busy cycles.

There are two reasons for the low communication to computation ratio. First, although the amount of private (not shared) data in the application is small, most of the references in the application are actually to private data. For every piece of shared data that a processor reads, it performs a reasonably large amount of computation (a particle-cell interaction, say) in which it primarily references temporary, private data structures that are reused in other computations. For example, for an 8K particle Barnes-Hut simulation with =1.0, only 15% of the references (23% of reads and 0.7% of writes) are to shared data. Second, although the *static* and *loadbalance* schemes cause every processor to reference data from almost the entire tree during force computation, many of these data—particularly at higher levels of the tree—are still reused substantially by the processor.[12] The impact of communication and hence physical locality is likely to be much more significant when larger machines are used, however, as indicated by Figure 17. This is particularly true since the number of particles is not expected to scale linearly with the number of processors [27, 24].

Besides performance benefits on large machines, physically contiguous partitions have other important advantages as well. They allow us to use a more efficient tree-building algorithm, as we shall see later in this section, which helps alleviate an important performance bottleneck as the number of processors grows. A lack of physical contiguity also increases the working set size in the force computation phase, and has important implications for the programming complexity and sizes of problems that can be run on message-passing machines [28, 24].

The Spacelocality Scheme: Incorporating physical locality with a static notion of load balancing does indeed lead to dramatically better communication behavior in the *spacelocality* scheme (see Figure 18(b)). However, since a processor's particles are physically clumped together, giving every processor an equal number of particles introduces structural load imbalances: A processor with particles in a dense region of the distribution has much more work to do than a processor with particles in a sparse region. Compromising load balancing has a more severe performance impact than compromising locality in this application, at least for reasonable problem sizes running on moderate numbers of processors. The *spacelocality* scheme therefore has the worst overall performance for nonuniform distributions.

The Costzones and ORB Schemes: The *costzones* scheme has low synchronization wait times as well as low communication. It performs best among all schemes in our experiments, and its performance also scales best with the number of processors. (We were also able to perform a 16K particle run on 128 processors of the simulated architecture, in which case the *costzones* scheme yielded a speedup of 118.). This demonstrates that while dynamic load balancing and dynamic locality are needed for scalable parallel performance, both can be easily provided under programmer control (without resorting to task stealing for load balancing) by taking advantage of certain insights into the application.

[12] The speedups of the *static* and *loadbalance* schemes are even closer to those of *costzones* on DASH than on the simulator, because of the effect of cache-to-cache sharing on the bus within a 4-processor cluster. Instead of every processor incurring a very high-latency read miss on every datum that has to come in from outside the cluster, only one processor in the cluster incurs this remote miss latency—other processors obtain the data from that processor's cache with a much lower latency. This additional, cluster level of the memory hierarchy makes the impact of read-sharing communication smaller on DASH than on the simulator.

Figure 18(b) shows that the communication and synchronization overheads incurred by using *costzones* and the more complicated *ORB* are quite similar. However, the speedup curves in Figure 17 show *costzones* performing significantly better overall. Let us try to understand why. Figure 19 shows profiles of execution time for the simulator runs whose speedups were shown in Figure 17(b). Every bar shows the percentage of time spent in different phases of the application, with the number of processors shown under that bar. The percentages are normalized so that the execution time of the *costzones* scheme with the given number of processors is 100%.

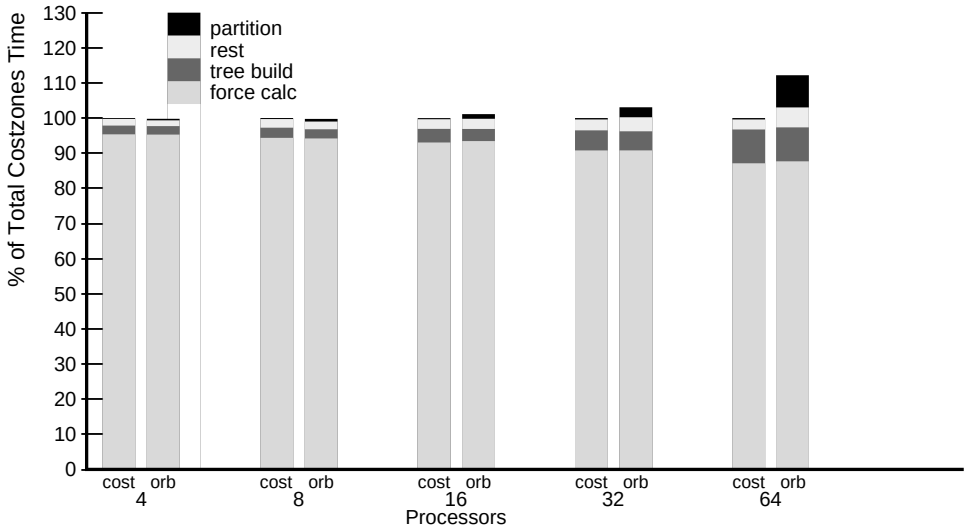


Figure 19: Barnes-Hut: Execution time profiles for *costzones* and *ORB* on the simulator.

The profiles show that the two schemes are almost equally effective at obtaining their goals in partitioning the force computation phase. However, what makes the overall performance of *ORB* worse is the time spent in the ORB partitioning algorithm. Despite our efforts to optimize the implementation of this algorithm [13], the time spent in it is much larger than that spent in *costzones* partitioning, particularly for larger numbers of processors. The overhead of *costzones* partitioning scales extremely slowly with the number of processors, while that of *ORB* increases much more rapidly. Similar results were observed for other problem sizes on DASH. The simpler *costzones* partitioning mechanism, which can be implemented in a few lines of code, thus yields better performance than the more complex and sophisticated *ORB,* at least for the problem sizes we have studied.

8.4.3 The Parallel Tree-Building Bottleneck

Unfortunately, Figure 19 exposes a potential performance bottleneck when using larger numbers of processors: The the fraction of the execution time spent in the tree-building phase increases with the number of processors, which means that the tree-building phase does not speed up quite as well as the other phases. Let us examine how the tree-building algorithm works, and see if we can improve its parallel performance.

The sequential tree-building algorithm works as follows. Proceeding one at a time, every particle is loaded into the root of the tree and percolated down to the current leaf cell in which it belongs. This leaf cell is either empty or already has a particle in it. In the former case, the particle is simply inserted in the cell. In the latter, the cell is subdivided and both particles (the new one and the one that was already in the cell) are moved

[13] Most of the time in ORB partitioning is spent in the root-finder that finds the bisector in a given direction. We experimented with different root finding algorithms for discrete functions, such as the bisection and Van Wijngaarden-Dekker-Brent algorithms described in [9, 4]. The best performance we obtained was from a decisection algorithm (similar to bisection, except that the currently guessed domain is split into 10 equal subdomains rather than 2 at every step in the root-finder). We use the decisection algorithm in the results presented in this paper.

into the appropriate children cells. If both particles are moved to the same child cell, that child must also be subdivided until there is at most one particle per leaf cell.

The obvious way to parallelize this tree-building algorithm is to have processors insert their particles into the shared tree concurrently, using the same basic sequential algorithm and synchronizing as necessary. This is the parallel algorithm used in our initial implementation, the one for which results are shown in Figure 19. Every time a processor wants to modify the tree in this parallel algorithm, either by putting a particle in an empty cell or by subdividing a cell, it must first lock that cell and ensure that no other processor tries to modify it at the same time. As the number of processors increases, the overhead of executing these locks and unlocks becomes substantial and hurts performance.

Contention for the above locks is reduced by the physical contiguity of the partitions yielded by *costzones and* ORB. These schemes effectively divide up the tree into distinct sections, and assign a processor to each section. This means that once the first few levels of the tree are built (incurring significant lock contention), processors construct the rest of their sections without much interference and hence contention. However, the uniprocessor overhead of executing the lock and unlock routines remains at all levels.

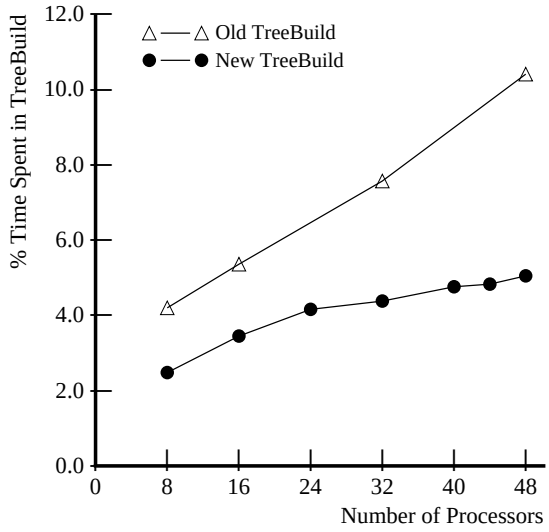


Figure 20: Barnes-Hut: Percentage of total execution time spent building the tree on DASH.

Fortunately, the same physical contiguity of partitions also allows us to construct a better parallel tree-building algorithm, with less locking overhead and even less contention. This algorithm splits the parallel tree construction into two steps. First, every processor builds its own version of the tree using only its own particles.[14] There is no synchronization in this step. Second, these individual trees are merged together into the single global tree used in the rest of the computation. Appendix C describes the algorithm in detail. Since entire subtrees, rather than individual particles, are usually merged into the global tree, the number of times a processor has to lock a cell in the global tree is very small. There is likely to be some extra computation, however, since the local trees do have to be merged after they are built.

To compare the old and new tree-building algorithms, Figure 20 shows the percentage of the total execution time of the application on the DASH multiprocessor that is spent in building the tree. The problem size used is the same as the one for which speedups are presented in Figure 17(a). The new algorithm clearly performs much better than the old one, particularly as the number of processors increases.

Of course, the improved tree-building algorithm is useful only if the partitions are mostly physically contiguous. The extra overheads of first building local trees and then merging them are likely to hurt performance

[14]This step is made more efficient by having a processor remember where in its tree it inserted its last particle, and start traversing the tree from there for the next particle, rather than from the root.

for schemes such as *static* and *loadbalance* that do not incorporate physical locality.

9 The FMM Application

9.1 The Approaches

The basic approaches to partitioning the FMM application are similar to those used in Barnes-Hut. Partitioning in the FMM, however, is complicated by three facts: (i) the basic units of partitioning are cells which, unlike particles, do not persist across time-steps (since both the particle distribution and even the bounding box change), (ii) force-computation work is associated not only with the leaf cells of the tree but with internal cells as well, and (iii) cells undergo different numbers and types of interactions, each involving a different amount of work.

Fact (i) has several implications:

There is no partitioning scheme analogous to the *static* scheme used in the Barnes-Hut application.

The work counting needed for load balancing is more complicated, since the work count for a leaf cell has to be transferred down to its particles (which do persist across time-steps) and then back up to the new leaf cells that those particles fall into in the next time-step.[15] It is significantly more expensive and less effective to do these transfers for internal cells. The cost of an internal cell is therefore simply computed from the number of cells in its list (see Section 4.1.2), since list interactions dominate over list interactions (which are the only other types of interactions that internal cells undergo).

ORB partitioning is further complicated by having cells be the unit of partitioning, as we shall see.

Fact (ii) indicates that both the *costzones* and ORB schemes used in the Barnes-Hut application may have to be extended to partition internal nodes as well. Fact (iii) complicates work counting further, even ignoring the fact that cells do not persist across time-steps. It doesn't suffice to simply count the number of interactions per cell in the adaptive FMM, or even the number of interactions of different types. We therefore have to compute a cycle count for every cell-cell interaction computed (for example, the interaction of a cell with another cell in its list). Since the structure of every type of interaction is known, the cycle count for a particular interaction is computed by a simple function for that type of interaction, parametrized by the number of expansion terms being used and/or the number of particles in the cells under consideration. The cost of a leaf cell is the sum of these counts in all interactions that the cell computes. As mentioned above, the costs of internal cells are computed from only the list interactions. As in Barnes-Hut, the work counting is done in parallel as part of the computation of cell interactions, and its cost is negligible.

Let us examine the analogs of four of the five partitioning schemes used in the Barnes-Hut application (since the *static* scheme doesn't make sense here, see above).

Loadbalance: The *loadbalance* scheme requires no structural modification for use in the FMM application, other than the use of cells rather than particles as the units of parallelism.

Spacelocality: This scheme is also very similar to the *spacelocality* scheme in the Barnes-Hut application. Costzones-style partitioning is used (see below), but every cell is assumed to have the same cost.

Costzones: The *costzones* scheme as described for Barnes-Hut in Section 8.3.2 considers only the costs of leaves of the tree when determining its partitions. This is all that is required for the Barnes-Hut method, since internal cells have no work associated with them in the force calculation phase. Our initial implementation of

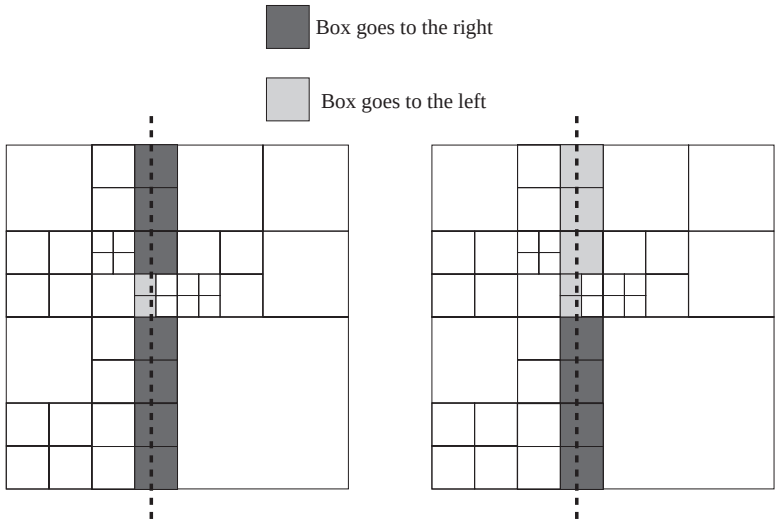
[15] We do this in the following way. The profiled cost of a leaf cell is divided equally among its particles. In the next time-step, a leaf cell examines the costs of its particles, finds the cost value that occurs most often, and multiplies this value by the number of particles to determine its cost.

costzones for the FMM, called *costzones-initial*, is a direct translation of that scheme. Internal cells are assigned in the same way as they were for locality in the Barnes-Hut case: to the processor that owns the majority of their children.

Figure 21: *Costzones-final* partitioning for the FMM

As mentioned earlier, internal cells also have substantial costs in the FMM, which should be considered in load balancing. We therefore extend the costzones scheme to partition internal cells based on their costs as well. We call this scheme *costzones-final*. In this scheme, an internal cell holds not just the sum of the costs of all *leaves* within it (as in *costzones-initial*), but the sum of the costs of all *cells* (leaf or internal) within it plus its own cost. In addition, it holds its own cost separately as well. In the traversal of the planarized tree that performs costzones partitioning, a processor examines cells for potential inclusion in its partition in the following order: the first two children (from left to right), the parent, and the next two children. [16] The pseudocode for the *costzones-final* scheme is shown in Figure 21.

ORB: The fact that the unit of parallelism is a cell rather than a particle complicates ORB partitioning in the FMM. When a space is bisected in ORB, several cells are likely to straddle the bisecting line (unlike particles, see Figure 22). In our first implementation, which we call *ORB-initial*, we try to construct a scheme that directly parallels the *ORB* scheme used in Barnes-Hut (except that both internal cells and leaf cells are included among the entities to be partitioned by ORB). Cells, both leaf and internal, are modeled as points at their centers for the purpose of partitioning, just as particles were in Barnes-Hut. At every bisection in the ORB partitioning, therefore, a cell that straddles the bisecting line (called a *border cell*) is given to whichever subspace its center happens to be in.



(a) ORB-initial (b) ORB-final

Figure 22: Partitioning of border cells in *ORB* for the FMM.

As we shall see, this treatment of border cells in the *ORB-initial* scheme leads to significant load imbalances. It is not difficult to see why. Given the fact that cells are always split in exactly the same way

[16] Recall that we use a two-dimensional FMM, so that every cell has at most four children.

(into four children of equal size), the centers of many cells are likely to align exactly with one another in the dimension being bisected. These cells are in effect treated as an indivisible unit when finding a bisector. If a set of these cells straddles a bisector, as is very likely, the entire set will be given to one or the other side of the bisector (see Figure 22(a)), potentially giving one side of the bisector a lot more work than the other. [17] The resulting load imbalance may be compounded in successive bisections.

To solve this problem, we extend the ORB method in the following way. Once a bisector is determined (by representing all cells as points at their centers, as in *ORB-initial*), the border cells straddling the bisector are identified and repartitioned. The repartitioning of border cells is done as follows, preserving the contiguity of partitions and trying to equalize costs as far as possible (see Figure 22(b)). A target cost for each subdomain is first calculated as half the total cost of the cells in both subdomains. The costs of the border cells are then subtracted from the costs of the subdomains that *ORB-initial* assigned them to in that bisection. Next, the border cells are visited in an order sorted by position along the bisecting axis, and assigned to one side of the bisector until that side reaches the target cost. The rest of the border cells are then assigned to the other side of the bisector. We call this scheme that repartitions border boxes *ORB-final*.

Note that whether or not border cells are repartitioned, the partitions yielded by *ORB* have lost the geometrically regular structure that the *ORB* partitions in the Barnes-Hut application had (see Figures 15 and 22).

9.2 Results

A similar two-cluster input distribution is used in the FMM application as in the Barnes-Hut, except that it is in two dimensions rather than three. Measurements on the DASH multiprocessor are made with 32K particles and an accuracy[18] of 10^{-10}. Measurements with some other reasonable numbers of particles and accuracies show similar trends across partitioning schemes. On the simulator, we were not able to run problems with more than 8K particles and a precision of 10^{-6} in a reasonable amount of time. With the 40 particles per leaf cell that we use on DASH and that Greengard recommends (see Section 4.1.2 and [14]), this leads to a very small number of cells and hence to large load imbalances. We therefore allow a maximum of 5 particles per leaf cell on the simulator, to obtain more cells and hence more concurrency. Since this still causes significant load imbalances with the larger numbers of processors, we do not present speedups obtained with the simulator but use it only to compare the load balancing and communication behaviors of different schemes. As in the Barnes-Hut application, all measurements are started at the end of the second time-step.

Figure 23(a) compares the performance on DASH of the four schemes that are direct analogs of the schemes used in the Barnes-Hut case: *loadbalance*, *spacelocality*, *costzones-initial* and *ORB-initial*. The *costzones-initial* scheme clearly performs better than the others, although even its performance falls off as more processors are used owing to its naive treatment of internal cells of the trees (which represent over 25% of the total force computation cost for this particle distribution). The *ORB-initial* scheme performs relatively poorly, owing to the problem with border cells which account for a significant portion of the cost at each bisection.

Figure 23(b) shows the speedups obtained when *costzones-initial* and *ORB-initial* are replaced by *costzones-final* and *ORB-final*. The best performing scheme in all cases is *costzones-final*. The improvement in partitioning parent cells clearly avoids the deterioration of speedup observed with *costzones-initial*. The *ORB-final* scheme yields substantial improvement over *ORB-initial*, which makes it almost as good as *costzones-final* for up to 32 processors with this nonuniform distribution.

Figure 24 shows that *ORB-final* and *costzones-final* have very similar communication overhead, while *ORB-final* has slightly worse load balancing behavior. The net result is similar to that in the Barnes-Hut case: *costzones-final* causes the force-computation phase to perform very slightly better, and the higher partitioning cost in *ORB-final* causes the overall application performance of *costzones-final* to become increasingly better relative to *ORB-final* as the number of processors increases.

[17] This situation is even more likely with a uniform distribution, where many cells will have their centers exactly aligned in the dimension along which a bisection is to be made.

[18] Recall from Section 4.1.2 that the force calculation accuracy governs the number of terms used in the multipole and local expansions. The number of terms is equal to , where is the accuracy and = 1.828.

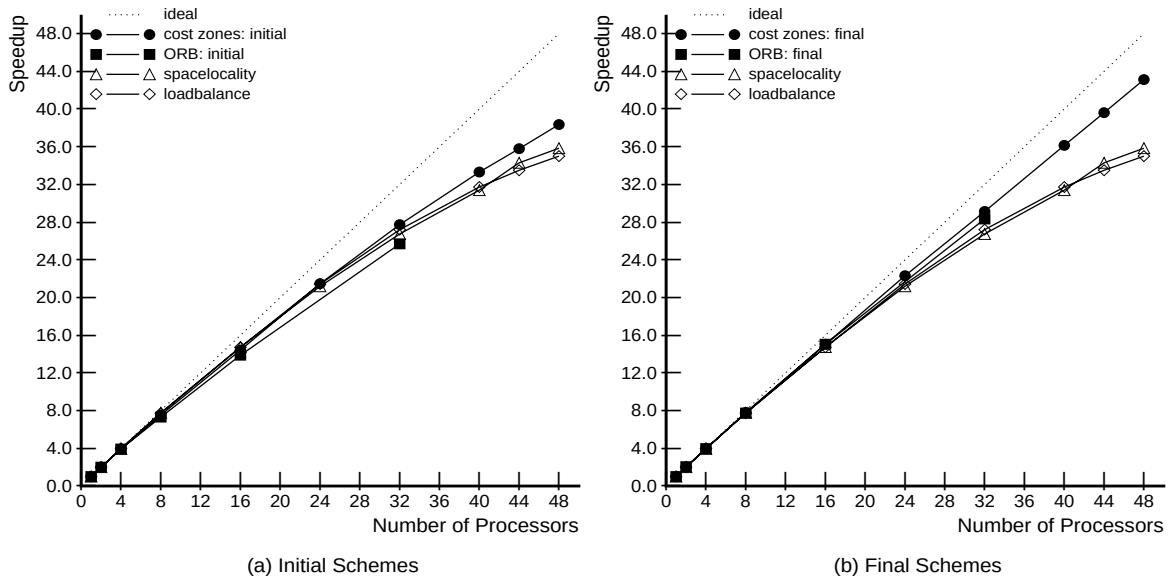


Figure 23: Speedups for the FMM application on DASH (= 32k, = 10 [10]).

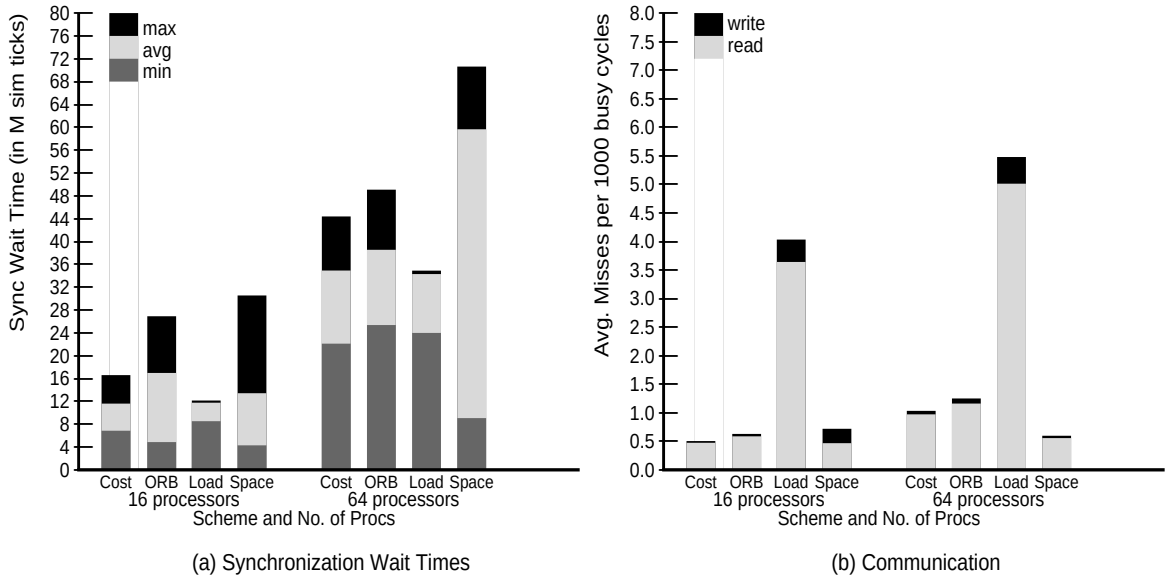


Figure 24: FMM behavioral statistics on the simulator (= 4k, = 10 [4]).

Thus, the FMM application also requires dynamic repartitioning for both load balancing and locality. However, because the units of partitioning are cells—each with many particles—and because internal nodes of the tree have significant amounts of work associated with them, direct translations of the *costzones* and *ORB* schemes as used in the Barnes-Hut case do not suffice. Substantial enhancements to these schemes are required, as are more complicated work counting and prediction mechanisms than in Barnes-Hut. Finally, the simplicity of *costzones* partitioning versus *ORB* is further emphasized by the FMM, since the partitioning complexity that the FMM adds for effective performance is greater in *ORB* than in *costzones*: Both have to incorporate the costs of internal cells, while only *ORB* has also to repartition border cells at every bisection.

By using the appropriate partitioning techniques, we find that both the prominent hierarchical methods for classical N-body problems can be made to yield excellent and seemingly scalable parallel performance. There is no need to resort to dynamic task-stealing, primarily because the physical systems that classical problems

model evolve slowly with time, and because low-overhead work profiling metrics can be identified in both algorithms. The approaches to partitioning that are effective in both the Barnes-Hut and FMM applications are quite similar. The two approaches also often require similar traversals of and operations on tree data structures. This suggests that libraries of the partitioning/manipulation techniques (which can be customized to a particular application by passing them the appropriate user-defined primitive functions) might prove very useful to parallel programmers. In the next section, we will see how the very different characteristics of the radiosity problem—from those of the classical problems—require very different kinds of techniques to obtain good speedups on the hierarchical radiosity method.

10 The Radiosity Application

10.1 The Straightforward Approaches

The main parallel loop in every iteration is a loop over all polygon-polygon interactions. Our first two approaches exploit this parallelism in much the same way as an automatic scheduler might, to produce the following partitioning schemes which are analogous to the *static* and *loadbalance* schemes, respectively, in Barnes-Hut:

> *Static*: Every processor is assigned a set of input polygons and their interactions, so that there are an equal number of initial polygon-polygon interactions per processor. The assignment is the same in every iteration. A processor works on only the interactions it is assigned, and the patch-patch interactions that they spawn.
>
> *Single-queue-polygon* (*SQ-Pol*): Processes dynamically obtain initial polygon-polygon interactions until there are none left. Once a processor obtains a polygon-polygon interaction, it performs all the computation for that interaction.

Speedups for these schemes on the DASH multiprocessor (as well for several other schemes which we will describe) are shown in Figure 25. The input is a room scene used in the paper that presented the sequential algorithm [16]. The speedup with the *static* scheme saturates at about 4 processors, and with the *single-queue-polygon* scheme at about 10 processors. In the case of *single-queue-polygon*, the reason for the saturation is that the granularity of a task (a polygon-polygon interaction) is very large compared to the number of tasks. For the same reason, load balancing in the *static* scheme is not helped by the averaging effect of large partitions, as it was in Barnes-Hut, particularly since the work per input polygon varies widely across polygons (depending on their radiosities, areas and relationships to other polygons). The naive schemes thus perform poorly in this application.

10.2 Techniques for Load Balancing and Data Locality

Let us now see what it takes to incorporate data locality and load balancing in this application. We begin with a discussion of locality, and then examine load balancing.

10.2.1 Data Locality

One important difference in the radiosity application from the classical N-body problems is that physical locality in the problem domain is not nearly as important. Since the distance between two patches is not the only criterion governing the degree of interaction—the angle and visibility between the patches are also very important, as is the radiosity of each patch—partitioning techniques analogous to *costzones*, *ORB* and *space-locality* do not apply.

There are three primary forms of locality in this application. First, a form of object locality can be obtained by having a processor work on interactions involving the same input polygons and their subpatches

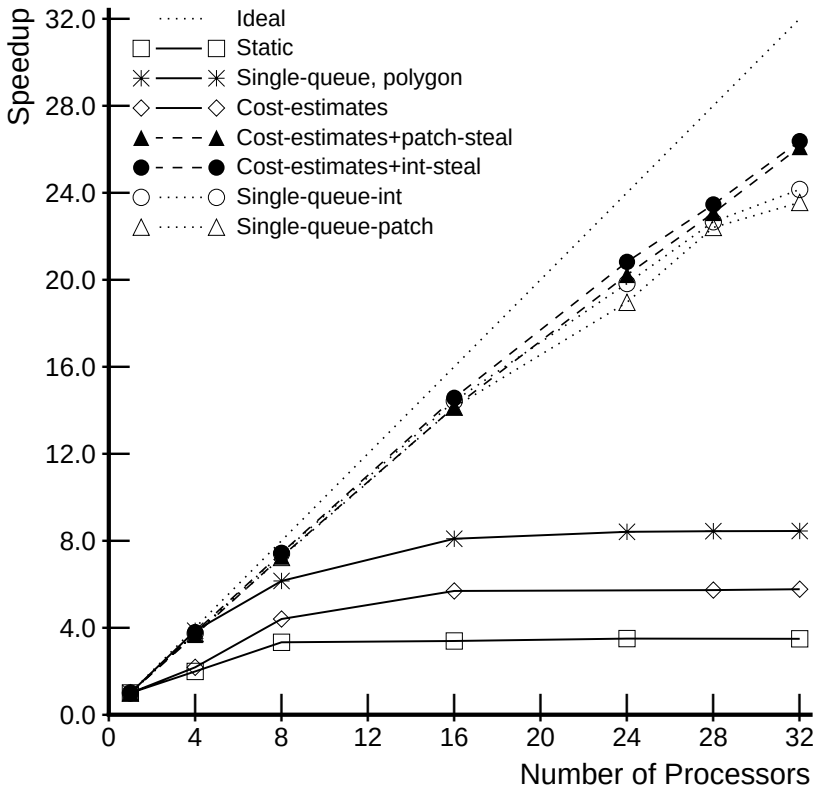


Figure 25: Speedups for the radiosity application on DASH.

in every iteration. Second, locality can be exploited across patch computations by processing patches in an appropriate order. Patches are refined depending on the errors computed in their interactions. Because sibling patches in a quadtree are physically adjacent and belong to the same polygon, interactions from these siblings to another patch (in another polygon) usually have similar amounts of error. Thus, if an interaction from one of the siblings causes the destination patch to be subdivided, interactions from the other siblings to the same patch are likely to cause subdivision as well. As a result, the sets of patches with which sibling nodes interact overlap substantially. For the input scenes we used, we found that of all the interactions at a given level of subdivision, 81% are made to the same patch from all four siblings, and 92% are made to the same patch from at least three of four siblings. This high coherence of the interactions can be exploited for locality by performing a breadth first traversal of the quadtree, so that sibling patches are processed consecutively.

The third form of locality can be exploited in visibility testing. Visibility testing involves shooting a finite number of rays between random points on the two interacting patches, and counting the number of these rays that are intercepted by other intervening patches. The visibility between the two interacting patches is the fraction of rays that are not intercepted. The BSP tree is used to find intercepting patches efficiently. A description of how the BSP tree is constructed and used can be found in Appendix A. Basically, a visibility calculation between two patches involves traversing a pruned inorder path in the BSP tree between the polygons that those patches belong to. Locality can therefore be exploited by trying to ensure that the same subset of BSP-tree nodes is traversed by a processor in successive visibility calculations. In Appendix A, we show that this locality is easily obtained by traversing the quadtree representing an input polygon depth-first rather than breadth-first.

Thus, while breadth-first traversal of the quadtrees is preferable for locality in interaction refinement, depth-first is preferable for locality in visibility testing. Since visibility testing and interaction refinement are

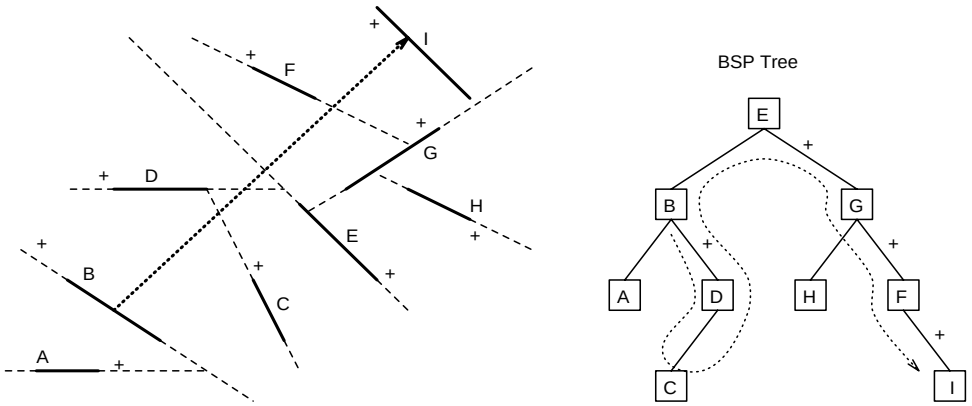


Figure 26: Visibility testing using a BSP tree.

performed during the same traversal of a quadtree, we have to choose one of the traversal schemes. We select depth-first traversal, because visibility testing consumes more than 90% of the sequential running time and because most of the cache misses occur in visibility testing. Fortunately, depth-first traversal preserves a significant amount of locality in interaction refinement as well. For example, leaf-level siblings are visited successively even under depth-first search, and these nodes are a large fraction (about three quarters) of the nodes in the system.

The *static* scheme we described earlier both exploits object locality across iterations and provides depth first traversal of the quadtrees. It therefore satisfies our locality criteria, and there is not much more a programmer can or needs to do about locality. The more important issue in this application is load balancing, and we examine that next.

10.2.2 Load Balancing

Since we would like to obtain load balancing without losing control over data locality if possible, and since the *static* scheme provides excellent data locality, we try to incorporate load balancing in the *static* scheme. Our first attempt was to use the same approach that was successful in the classical applications: using work profiles from one iteration to predict those in the next. This approach was unsuccessful for two reasons:

There are only a few (less than 10) iterations in this application—unlike the hundreds of time-steps in the gravitational case—and a disproportionately large fraction of the work is done in the first iteration. No profiled information is available for this iteration.

Unlike classical N-body problems, the domain being simulated in radiosity is not a physical system evolving slowly with time. Thus, the work associated with a patch in one iteration[19] is *not* a good indicator of the work associated with it in the next one. A patch has significant work associated with it in only those iterations in which many of the patch's interactions lead to subdivisions. The most work is associated with the first couple of iterations, since the patches have not been subdivided much already. After this, subdivisions happen unpredictably, depending on when current radiosity levels and form factors cause the value for an interaction to cross a certain threshold.

We experimented with several cost estimating and profiling functions. Since there is a general (nonmonotonic) decreasing trend in the cost of a patch across iterations, we even tried to use a history of profiled costs rather than the profiled cost in a single iteration. The best scheme that we could come up with is represented as *cost-estimates* in Figure 25. In it, we give exponentially decaying weights (1.0, 0.5, 0.25) to the costs in the

[19]Note that iterations in this application are a mathematical construct used to obtain convergence and have nothing with physical time, unlike the time-steps in the classical applications.

three previous iterations. While the performance of this scheme is better than that of the *static* scheme, it does not nearly solve the load balancing problem.

Our only resort for incorporating load balancing is through dynamic task-stealing. However, we must use finer-grained tasks than were used in the *single-queue-polygon* scheme. To reduce contention and preserve locality, we use a distributed task-queueing mechanism in which every processor has its own task queue. In every iteration of the radiosity algorithm, we first use one of the mechanisms discussed above—*static* or *cost-estimates*—to provide every task queue with an initial set of polygon-polygon interactions. If a processor subdivides a patch in the course of computing interactions, it inserts a task for each created patch into its own queue. A task for a patch includes all the interactions that that task must perform. It is created when all the initial interactions for the patch are determined, i.e. when its parent patch has been completely processed.

When a processor finds no more work in its task queue, it steals tasks from other processors' queues. The following mechanisms are used to preserve locality in the dynamic task-queueing system. Tasks that a processor creates are inserted at the head of its queue. A processor always dequeues tasks from the head of its own queue (which yields the desired depth-first search of the quadtrees). When a processor steals from another queue, however, it dequeues a task from the tail of that queue. This maximizes the likelihood of stealing a large-grained task and exploiting the locality within it.

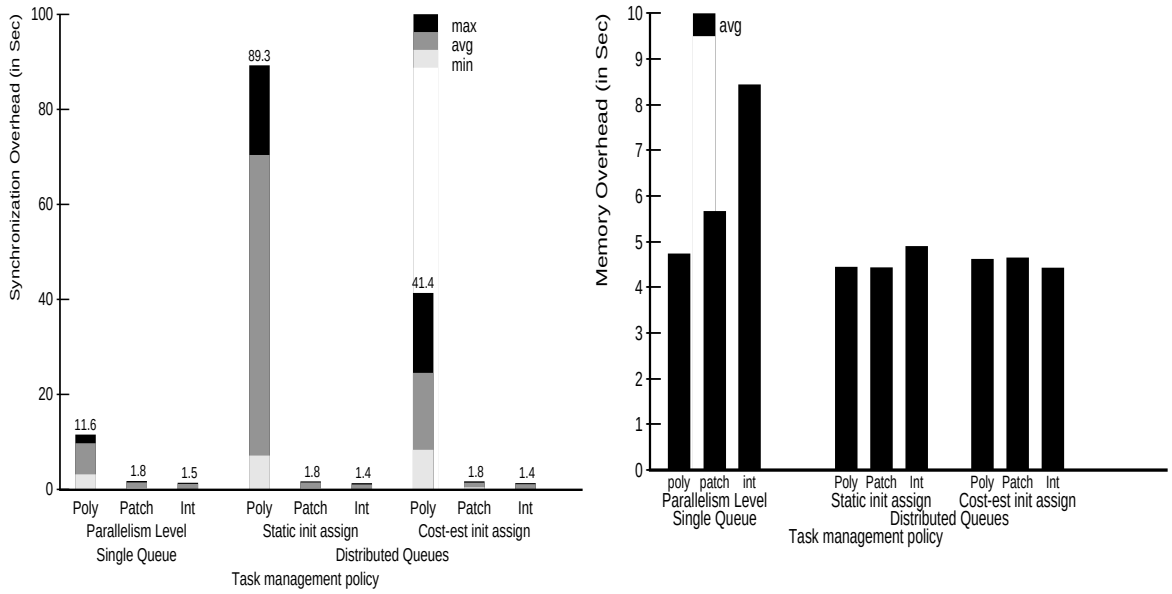


Figure 27: Synchronization and memory overhead for the radiosity application on DASH (= 16). Execution time for best case (rightmost bar) is 22 seconds.

The scheme that incorporates patch stealing is called *cost-estimates+patch-steal* (*C+PS*)[20]. Figure 25 shows that the incorporation of patch stealing makes a dramatic difference to parallel performance and provides very good speedups. Measurements obtained with the MTOOL performance debugger [12] on DASH (Figure 27) show that the techniques used to maintain locality while stealing cause the stealing to contribute no appreciable increase in memory referencing overhead for these numbers of processors (16, in the figures). Figure 27 shows the overheads due to time spent in the memory system and time spent waiting at synchronization events (such as the final barrier at the end of the radiosity computation in an iteration).

The granularity of tasks so far has been a patch and all its interactions. Since we find that there is still some small load imbalance at the end of an iteration, we examine the impact of using a smaller task granularity; that is, the finest level of parallelism described in Section 4.2.2. A task now is a single patch-patch interaction, so that the time-consuming visibility computation of a patch with different patches on its interaction list can be done in parallel. If an interaction subdivides one of the patches and thus spawns new interactions, then each

[20] *Static+patch-steal* shows similar results.

of the newly created interactions is a new task which is placed on the creating processor's task queue. This scheme is called *cost-estimates+int-steal* (*C+IS*). The performance debugger indicates that this scheme reduces load imbalance but increases memory system overhead. The net result is only a small improvement in speedup for this problem and number of processors.

Finally, to examine the impact of ignoring locality completely, we implemented schemes in which the patch and interaction tasks of *C+PS* and *C+IS* are inserted in and dequeued from the head of a single queue. These schemes are called *single-queue-patch* (*SQ-P*) and *single-queue-int* (*SQ-I*) in Figure 25. Their performance is not too much worse than that of *C+PS* and *C+IS*, since there is enough locality even within a patch-patch interaction for a machine of this size, and since contention for the single queue is also small compared to task granularity. This situation is likely to change for larger machines, however.

We also examined the performance of the different schemes with a room scene twice as large, and found the results to be quite similar (slightly larger overall speedups, and slightly smaller differences between the schemes that are load balanced). At least for moderately sized parallel machines, then, load balancing is a much more important performance issue than locality for this application, just as it was for the classical applications.

11 Summary and Concluding Remarks

We studied the process of obtaining effective speedups through parallelism on the major hierarchical N-body methods that have been proposed so far. Our study included the two most powerful methods for classical N-body problems—the Barnes-Hut algorithm and the Fast Multipole Method—using a galactic simulation as our example problem. In addition, we examined a hierarchical method for radiosity calculations in computer graphics, an application that uses the same basic approach as the other two but which has some very different characteristics. The focus of our study was on partitioning/scheduling techniques that simultaneously obtain load balancing and exploit data locality.

We found that all the applications can be made to yield excellent and scalable parallel performance. However, static or straightforward dynamic parallel implementations of the sort that a compiler or automatic scheduler might produce[21] do not scale. This is because the workload distribution and communication characteristics in the applications are nonuniform and change as the computation proceeds. The characteristics of classical N-body problems allow partitioning techniques based on a similar approach to be successful for the Barnes-Hut and Fast Multipole methods that are used to solve them, even though the techniques that are successful in the Barnes-Hut method have to be extended significantly for the FMM. Since several phases in these methods require similar operations to be performed on tree data structures, libraries of tree manipulation and tree/space partitioning techniques (which can accomodate different user-defined primitive functions) might be very helpful to a parallel programmer for classical N-body problems.

Although the radiosity application uses an algorithm based on hierarchical N-body methods, its characteristics are very different from those of the classical applications. As a result, partitioning and scheduling techniques that an automatic parallelizer might implement perform much more poorly in the radiosity application. Even the more sophisticated mechanisms, such as *costzones* and *ORB*, that were successful in providing load balancing and locality in the classical applications do not apply to the radiosity problem. Thus, although similar partitioning techniques succeed on the classical N-body applications, no single technique can be claimed to be successful for applications of the hierarchical N-body approach.

Despite our best efforts, we could not find an effective predictive mechanism that could provide load balancing without on-the-fly task-stealing in the radiosity application. Fortunately, however, there is enough locality within tasks, so that relatively simple stealing mechanisms do not compromise locality too much. The result is that the hierarchical radiosity application also yields good parallel performance that is likely to scale to large numbers of processors.

[21] If it knew the appropriate parallelism, which we believe would be impossible for today's compiler technology to find.

In another paper [28, 24], we show that the nonuniform, dynamically changing nature of problems that hierarchical N-body methods are applied to, together with their need for long-range communication in the physical domain, cause multiprocessors that support a shared address space as the communication abstraction to have substantial advantages in design and programming complexity over machines that support only explicit message passing among private address spaces. We also argue that this message-passing complexity translates directly to substantial performance overheads, which can be avoided with a shared address space.

Acknowledgements

We would like to thank Joshua Barnes for providing us with the sequential Barnes-Hut program. Dennis Roger and Eric Bruni developed an initial implementation of the parallel code as a class project at Stanford University. Maneesh Agrawala provided substantial help toward the implementation of ORB partitioning. We would also like to thank John Salmon, Rohit Chandra and Kourosh Gharachorloo for various discussions. This work was supported by DARPA under Contract No. N00039-91-C-0138. Anoop Gupta is also supported by a Presidential Young Investigator Award, with matching grants from Ford, Sumitomo, Tandem and TRW.

References

[1] S.J. Aarseth, M. Henon, and R. Wielen. *Astronomy and Astrophysics*, 37, 1974.

[2] Andrew A. Appel. An efficient program for many body simulation. *SIAM Journal of Scientific and Statistical Computing*, 6:85–93, 1985.

[3] Joshua E. Barnes and Piet Hut. A hierarchical O(N log N) force calculation algorithm. *Nature*, 324(4):446–449, 1986.

[4] Richard P. Brent. *Algorithms for Minimization Without Derivatives*. Prentice-Hall, 1973.

[5] Tony Chan. Hierarchical algorithms and architectures for parallel scientific computing. In *Proceedings of ACM Conference on Supercomputing*, May 1990.

[6] A. J. Chorin. Numerical study of slightly viscous flow. *Journal of Fluid Mechanics*, 57:785–796, 1973.

[7] K. Chua, A. Leonard, F. Pepin, and G. Winckelmans. Robust vortex methods for three-dimensional incompressible flows. In *Proceedings of Symposium on Recent Advances in Computational Fluid Dynamics*, 1988.

[8] M.F. Cohen and D.P. Greenberg. The hemi-cube: A radiosity solution for complex environments. In *Proceedings of SIGGRAPH*, 1985.

[9] William Press et al. *Numerical Recipes in C*. Cambridge University Press, 1988.

[10] Geoffrey C. Fox. *Numerical Algorithms for Modern Parallel Computer Architectures*, chapter A Graphical Approach to Load Balancing and Sparse Matrix Vector Multiplication on the Hypercube, pages 37–62. Springer-Verlag, 1988.

[11] Henry Fuchs, Gregory D. Abram, and Eric D. Grant. Near real-time shaded display of rigid objects. In *Proceedings of SIGGRAPH*, 1983.

[12] Aaron J. Goldberg and John L. Hennessy. Performance debugging shared memory multiprocessor programs with MTOOL. In *Proceedings of Supercomputing '91*, pages 481–490, November 1991.

[13] Stephen R. Goldschmidt and Helen Davis. Tango introduction and tutorial. Technical Report CSL-TR-90-410, Stanford University, 1990.

[14] Leslie Greengard. *The Rapid Evaluation of Potential Fields in Particle Systems*. ACM Press, 1987.

[15] Leslie Greengard and Vladimir Rokhlin. A fast algorithm for particle simulation. *Journal of Computational Physics*, 73(325), 1987.

[16] P. Hanrahan, D. Salzman, and L. Aupperle. A rapid hierarchical radiosity algorithm. In *Proceedings of SIGGRAPH*, 1991.

[17] Lars Hernquist. Hierarchical N-body methods. *Computer Physics Communications*, 48:107–115, 1988.

[18] J.G. Jernigan and D.H. Porter. A tree code with logarithmic reduction of force terms, hierarchical regularization of all variables and explicit accuracy controls. *Astrophysics Journal Supplement*, page 871, 1989.

[19] Jacob Katzenelson. Computational structure of the N-body problem. *SIAM Journal of Scientific and Statistical Computing*, 10(4):787–815, 1989.

[20] Dan Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148–159, May 1990.

[21] Vladimir Rokhlin. Rapid solution of integral equations of classical potential theory. *Journal of Computational Physics*, 60:187–207, 1985.

[22] Ed Rothberg, Jaswinder Pal Singh, and Anoop Gupta. Working sets, cache sizes and node granularity issues for large-scale multiprocessors. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993. To appear.

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

[24] Jaswinder Pal Singh. *Parallel Hierarchical N-body Methods and their Implications for Multiprocessors*. PhD thesis, Stanford University, February 1993.

[25] Jaswinder Pal Singh and John L. Hennessy. *High Performance Computing II*, chapter Data Locality and Memory System Performance in the Parallel Simulation of Ocean Eddy Currents, pages 43–58. North-Holland, 1991. Also Stanford University Tech. Report No. CSL-TR-91-490.

[26] Jaswinder Pal Singh and John L. Hennessy. Parallelism, locality and scaling in a molecular dynamics simulation. To appear as Stanford University Technical Report, 1992.

[27] Jaswinder Pal Singh, John L. Hennessy, and Anoop Gupta. Scaling parallel programs for multiprocessors: Methodology and examples. *IEEE Computer*. To appear. Also available as Stanford Univeristy Tech. Report no. CSL-TR-92-541, 1992.

[28] Jaswinder Pal Singh, John L. Hennessy, and Anoop Gupta. Implications of hierarchical N-body techniques for multiprocessor architecture. Technical Report CSL-TR-92-506, Stanford University, 1992.

[29] Feng Zhao. An O(n) algorithm for three-dimensional N-body simulations. Technical Report 995, MIT Artificial Intelligence Laboratory, 1987.

Appendix

A Visibility Testing Using a BSP Tree in the Radiosity Application

In the radiosity application, the interaction between patches can be occluded by an intervening polygon. For example, in Figure 28(a) polygon *E* occludes polygons *C* and *H*. It is therefore necessary to compute the visibility between patches . To compute visibility, a fixed number of rays is shot between random points on the interacting patches, to see how many of these are intercepted by other polygons. The visibility is computed as the fraction of rays that reach the destination patch.

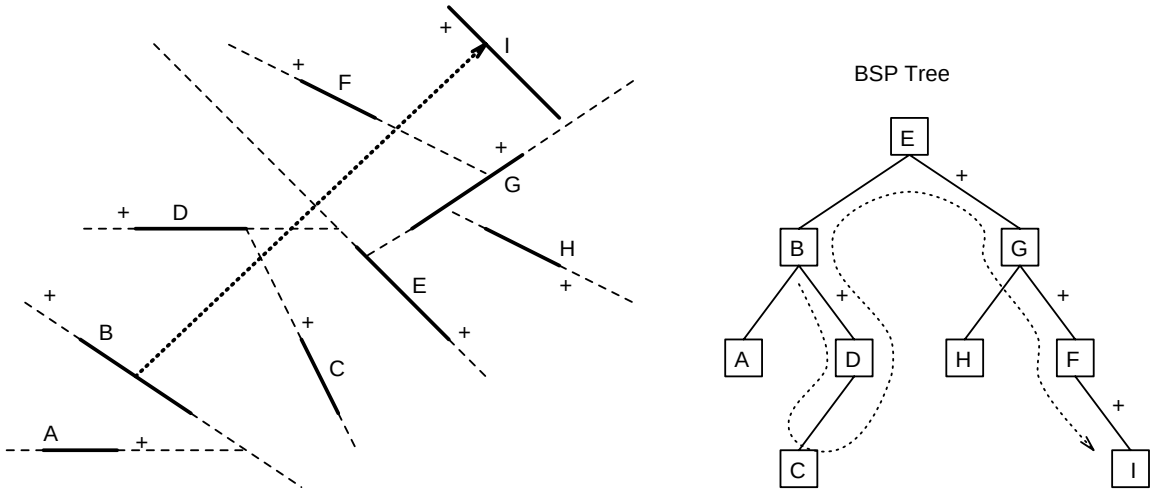


Figure 28: Visibility testing using a BSP tree.

In a naive visibility testing scheme, all other polygons must be tested for interception for every ray shot. The binary space partitioning (BSP) tree constructed at the beginning of the application allows the number of polygons tested to be greatly reduced. Every node in the BSP tree (Figure 28) is an input polygon. The root node separates the entire space into two halfspaces, separated by the supporting plane of the polygon at the root. Polygons in the left subtree of the root are those that are in its "left" halfspace, while polygons in the right subtree are in the "right" halfspace. Recursively, every node subdivides the subspace it inherits from its parent in a similar way.

For every ray fired in visibility testing, the BSP tree is partially traversed from the source to the destination patch in an inorder fashion, in that the order of traversal is (one subtree, the node, other subtree). An any node, whether the left or right subtree of the node is examined first depends on which subtree represents the subspace of that node's polygon facing the ray's source. If the supporting plane of a node (polygon) does not intersect the ray, it follows that the ray cannot be intercepted by any polygon in the subtree of that node which faces away from the ray's source. This fact can be used to prune the number of nodes examined in a traversal. Different rays may thus have different nodes pruned in their traversal, and examine different nodes.

Figure 26 shows a BSP tree traversal example, using a two dimensional analog (i.e., the visibility of line segments) for clarity. In this example, a ray is shot from segment *B* to segment *I*. Those line segments that lie between *B* and *I* are visited by a partial inorder traversal of the BSP tree shown by a dotted line. For example, segment *A* is not visited since *A* exists in the opposite half-space of *B* from the direction in which the ray travels, and segment *H* is not visited since it is in the opposite half-space of a segment (*G*) whose supporting line does not intersect the ray.

When an interaction is refined, visibility testing is performed for the newly created interactions. Because the new interactions are between subpatches of the old patches, the corridor through which the rays that are fired in visibility testing travel only becomes narrower. Thus, the subset of BSP tree nodes that are visited during

visibility testing for the new interactions is the same as the subset visited for the parent patches. This suggests that depth-first traversal of a quadtree is advantageous for obtaining locality in visibility testing.

B Three Dimensional Costzones with Nonuniform Numbering

As mentioned in Section 8.3.2, extending costzones to yield contiguous partitions in three dimensions requires us to use more child orderings than the four orderings used in two dimensions. We can restrict the possible orderings to traversals where all four cells in a given plane of a cube are traversed before moving to the adjacent plane, without losing any generality. In our examples, we show these planes as the back and the front plane of a cubical octree cell. When we move from one plane to another, we always move to the immediately adjacent cell in the other plane.

Given these conditions, there are

two planes to start in,

four possible cells to start at in that plane,

two directions (either clockwise or counterclockwise) to traverse the starting plane in, and

two directions to traverse the other plane in.

This yields a total of 32 different possible orderings for the children of a given cell. All 32 of these orderings are used when constructing the octree.

B.1 Description of Orderings

An ordering of children cells of a given cell is denoted by a six letter acronym, which describes how the ordering traverses the children cells. The six letters are the following:

Letter 1 (B/F): Denotes the starting plane, either Back or Front.

Letter 2 (R/L/U/D): Denotes the ordering within a plane. The letters stand for Right, Left, Up and Down. They can be thought of as the direction moved in from the initial cell in the current plane (front or back).

Letter 3 (C/A): Denotes a clockwise or anticlockwise direction of traversal in the starting plane. Together with Letter 2, this uniquely describes the ordering of the four cells in a two dimensional plane.

Letter 4 (F/B): Denotes the ending plane, and is therefore the opposite of Letter 1.

Letter 5 (R/L/U/D) and Letter 6 (C/A): Denote the ordering of cells in the ending plane, just as letters 2 and 3 describe the ordering in the starting plane.

Figure 29 shows an example of the ordering BRC-FUC. Three dimensional cubes are “flattened out” into two-dimensional planes. Each cube is represented by two adjacent large squares, each with four smaller squares inside. The large square on the left is always the back plane, the large square on the right is always the front plane, and the smaller squares inside correspond to the four cells in each plane. The starting cell in a plane is indicated by a circle, and the ending cell by a square.

B.2 Determining Child Orderings from Parent Orderings

The above configurations describe the ordering of the children cells of a particular parent (say). However, we also need to determine the ordering of the children of these children cells. As in the two-dimensional case, the ordering of a child ’s children depends on the ordering of ’s siblings and on which child of its parent

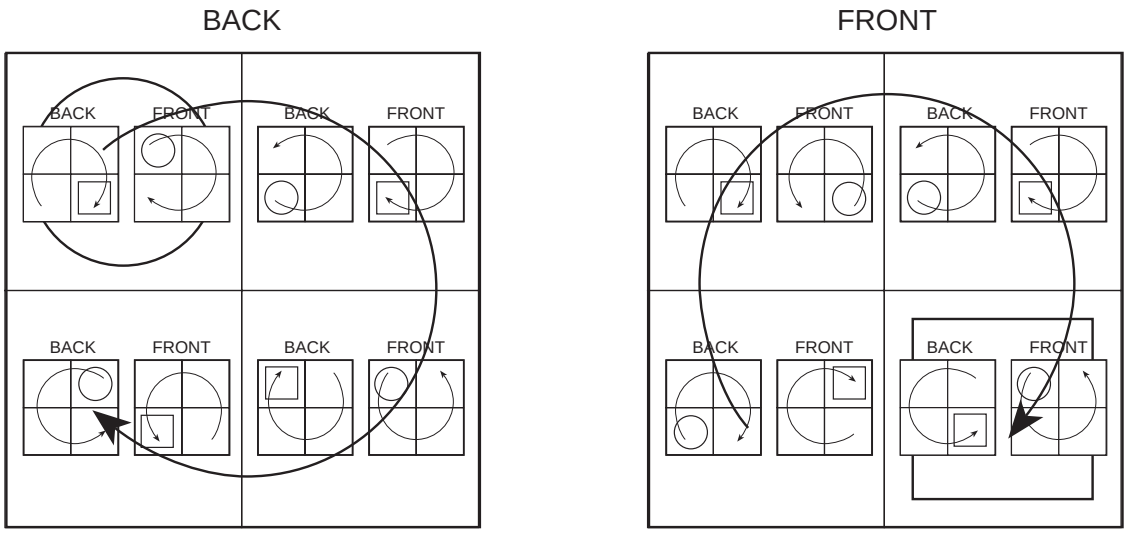


Figure 29: Example ordering, BRC-FUC, in the new 3-dimensional *costzones*.

( ) cell is. Figure 29 shows the ordering for the children of the cells ordered BRC-FUC (i.e. the children of 's children).

Since we start in the back plane of , we should exit the back plane from the front plane of the last child cell encountered in 's back plane. Also, since we always move from back plane to back plane of children or from front plane to front plane when moving between children in the same plane of , we must start in the front plane of the first child of that we encounter (so that we end up in the front plane of the last child encountered in the back plane of , and are positioned correctly to exit the back plane into the front plane of). Similarly, if we were to start in the front plane of , we would start in the back plane of the first child traversed in this plane.

Finally, to ensure physical locality in the partitions, we always start in the same corner of the appropriate child plane as that of the parent plane. For example, since the traversal of 's children starts in the top left hand corner of the back plane of , the traversal of the first child's children starts in the top left hand corner of the front plane of that first child.

A list of the 32 orderings that result from the above process can be found in [24].

C Parallel Tree Construction

This appendix sheds a little more light on the new algorithm for parallel tree construction described in Section 8.4.3, in which every processor builds a local tree out of the particles in its partition, and then merges its local tree into the single global tree.

Figure 30: Skeleton of new algorithm to build a global tree.

The high-level algorithm is shown in Figure 30. The building of the local tree is handled by the `InsertParticlesInTree` procedure which, given a particle list and a root cell, constructs a tree with those particles. The root cell of this tree represents the whole computational domain, not just a domain large enough to hold the local particles. Since only one processor is building the local tree, no synchronization is needed. To

improve efficiency, a processor remembers where in its tree it inserted its last particle, and starts traversing the tree from there rather than from the root for the next particle.

The `MergeLocalTree` procedure merges the local tree into the global tree. This procedure assumes that the roots of the local and global trees are identical (each representing the entire computational domain). This means that a cell in one tree represents the same subspace as the corresponding cell in another tree. This fact allows `MergeLocalTree` to make merging decisions based only on the types (internal, leaf or empty) of the local and global cells being merged.

Figure 31: Algorithm for merging a local tree into a global tree.

Pseudocode for `MergeLocalTree` is shown in Figure 31. It is first called with `local_cell` being the root of the local tree, `global_cell` being the root of the global tree, and `global_parent` being `NULL`. *MergeLocalTree* is a recursive algorithm that compares the types of the two cells it is called with (one from the local tree and one from the global tree) and takes an appropriate action based on these types. There are six relevant combinations of cell types:

1. *local cell is internal, global cell is empty*: The local cell is inserted into the tree.
2. *local cell is internal, global cell is a leaf*: The global leaf is removed from the tree. Then the particle (or particles, in the FMM) at that global leaf is inserted into the subtree whose root is the local cell. The parent of the global cell is relocked, and the local cell is inserted into the global tree.
3. *local cell is internal, global cell is internal*: A spatial equivalent to the local cell already exists in the global tree, so nothing is done with the local cell. The merge algorithm is recursively called on each of the local cell's children and their counterparts in the global tree.
4. *local cell is a leaf, global cell is empty*: Same as Case 1.
5. *local cell is a leaf, global cell is a leaf*: Same as Case 2.
6. *local cell is a leaf, global cell is internal*: The local cell is subdivided, pushing the particle(s) in it one level deeper in the local tree. Since the local cell is now internal, Case 3 applies.

Some notes about the algorithm. First, since `InsertCellInTree` and `RemoveCellFromTree` modify the global tree, they each lock the global parent before doing anything to the tree. Only these functions execute any locks. Second, since all processors try to merge as soon as they finish constructing their local trees, there may be some contention for the locks. That is, a processor may try to insert or remove a cell from the tree, only to find that someone else has done so already. For this reason, both these functions return whether they were successful or not. If a function was successful, then the processor that executed it can proceed. If not, then the processor has to get the new global cell from the global tree, and restart the merge that it was doing. The number of times a processor needs to restart is expected to be small: typically, it has to restart only once regardless of its current location in the tree.

The problem of contention at the higher levels of the tree is greatly alleviated by this new tree-construction algorithm. When the first processor tries to merge its local tree into the global tree, it finds the global root to be empty. As per case 1 above, it sets the global root to be its root. The processor's local tree has now become the global tree in one fell swoop, and contention for locking any one high-level cell is likely to be reduced. Since large subtrees are merged in a single operation, rather than single particles, the amount of locking required (and hence both the overhead of locking as well as the contention) is greatly reduced as well.

The reduction in locking overhead and contention in the new algorithm comes at a cost in extra work. There is some extra work done in first loading particles into local trees and then merging the local

trees, rather than loading particles directly into the global tree (as in the old tree building algorithm). When the partitioning incorporates physical locality, this extra work overhead is small and the reduction in locking overhead is substantial, since large subtrees are merged in a single operation. Of course, if the partitioning does not incorporate physical locality, this new tree-building algorithm has no advantages over the old one, and the extra work overhead it introduces will make it perform significantly worse.