Implications of Hierarchical N-Body Methods for Multiprocessor Architectures

JASWINDER PAL SINGH, JOHN L. HENNESSY, and ANOOP GUPTA
Stanford University

To design effective large-scale multiprocessors, designers need to understand the characteristics of the applications that will use the machines. Application characteristics of particular interest include the amount of communication relative to computation, the structure of the communication, and the local cache and memory requirements, as well as how these characteristics scale with larger problems and machines. One important class of applications is based on hierarchical N-body methods, which are used to solve a wide range of scientific and engineering problems efficiently. Important characteristics of these methods include the nonuniform and dynamically changing nature of the domains to which they are applied, and their use of long-range, irregular communication. This article examines the key architectural implications of representative applications that use the two dominant hierarchical N-body methods: the Barnes-Hut Method and the Fast Multipole Method.

We first show that exploiting temporal locality on accesses to communicated data is critical to obtaining good performance on these applications and then argue that coherent caches on shared-address-space machines exploit this locality both automatically and very effectively. Next, we examine the implications of scaling the applications to run on larger machines. We use scaling methods that reflect the concerns of the application scientist and find that this leads to different conclusions about how communication traffic and local cache and memory usage scale than scaling based only on data set size. In particular, we show that under the most realistic form of scaling, both the communication-to-computation ratio as well as the working-set size (and hence the ideal cache size per processor) grow slowly as larger problems are run on larger machines. Finally, we examine the effects of using the two dominant abstractions for interprocessor communication: a shared address space and explicit message passing between private address spaces. We show that the lack of an efficiently supported shared address space will substantially increase the programming complexity and performance overheads for these applications.

Categories and Subject Descriptors: B.0 [**Hardware**]: General; B.3 [**Hardware**]: Memory Structures; C.1.2 [**Computer Systems Organization**]: Multiple Data Stream Architectures; C.4 [**Computer Systems Organization**]: Performance of Systems; C.5.1 [**Computer System Implementation**]: Large and Medium Computers; D.0 [**Software**]: General; J.2 [**Computer Applications**]: Physical Sciences and Engineering

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.
Authors' addresses: Computer Systems Laboratory, Stanford University, Stanford, CA 94305; J. P. Singh is currently at Princeton University, Department of Computer Science, Room 423, 35 Olden Street, Princeton, NJ 08544; email: jps@cs.princeton.edu.

General Terms: Algorithms, Experimentation, Measurement, Performance

Additional Key Words and Phrases: Communication abstractions, locality, message passing, N-body methods, parallel applications, parallel computer architecture, scaling, shared address space, shared memory

1. INTRODUCTION

An understanding of workload behavior proved very useful in advancing the field of uniprocessor computer architecture. Analyses of instruction set usage led to the design of simpler and more-effective processors, and an understanding of temporal and spatial locality in programs allowed effective memory organizations to be designed. As we design machines with more and more processors, the performance impact of matching application characteristics to architectures becomes even greater, clearly suggesting that the architecture of large-scale multiprocessors should be driven by application characteristics as well.

The most important interactions between applications and architectures for multiprocessors are likely to be found at a different level than they were for uniprocessors. In particular, the difficulty in designing large-scale multiprocessors lies not in providing more aggregate computational power, but rather in providing the memory bandwidth and communication capabilities required to support the available computational power. Thus, insights into the communication needs and memory system requirements of applications are likely to be critical.

Four application characteristics emerge as being important to understand:

—What is the ratio of communication to computation, and how does it scale as larger problems are run on larger machines?

—What is the structure of the communication, and how can it best be supported by the architecture, both by communication mechanisms as well as by latency-hiding techniques?

—How do certain key hardware resources need to scale as processing nodes (and hence computational power) are added to the machine? For example, how should the size of processor caches or local memories scale in order to be effective but not unnecessarily large?

—Do the communication characteristics of the application lead to significantly different overheads—in programming complexity or execution time—under different programming models that an architecture might support, in particular, under the two dominant communication abstractions presented to a programmer today: a globally shared address space and private address spaces with explicit message passing between them?

Unfortunately, obtaining the desired information about parallel applications is not easy. There is neither an established base of representative applications nor an established methodology for writing parallel programs. And many of the interesting issues require a higher-level understanding of the applications than can be automatically extracted by profiling tools. Our only recourse, therefore, is to develop important classes of parallel applications carefully and extract our insights from them.

An important class of applications is those that use hierarchical N-body methods. These methods have wide applicability to problem domains that require high-performance computing; they solve large-scale problems efficiently and should be able to use large numbers of processors effectively; and they are not trivial to obtain good parallel performance on, but challenge parallel architectures in ways that are representative of important classes of applications.

Hierarchical N-body algorithms are a class of methods that satisfy all these criteria. In this article, we study the key architectural implications (listed above) of gravitational simulations that employ the two most-important hierarchical N-body algorithms: the Barnes-Hut method [Barnes and Hut 1986] and the Fast Multipole Method or FMM [Greengard and Rokhlin 1987].

We first show that exploiting temporal locality on accesses to communicated data is critical to obtaining good performance on hierarchical N-body applications, and argue that coherent caches in shared-address-space machines provide this locality both automatically and very effectively. Then, we examine the scaling of important execution characteristics as larger problems are run on larger machines. We find that, under the most appropriate model of scaling the applications, both the communication-to-computation ratio and the size of a processor's important working set (and hence the size of cache it needs to yield effective performance) increase slowly with the number of processors, while the per-processor main memory requirements in a shared address space decrease. Finally, we study the implications for communication abstractions. We find that the absence of a shared address space causes substantially increased algorithmic and programming complexity to manage communication in these applications—due to their highly nonuniform, long-range, and dynamically changing communication requirements—and that this complexity translates directly to significant runtime overheads in message-passing implementations. Because the Barnes-Hut method is simpler in structure and more widely used today, we make our main points through it first and later discuss the Fast Multipole Method either for support or as a point of comparison.

To motivate the study of hierarchical N-body methods, Section 2 discusses the insight behind them and their range of application in solving physical problems. The gravitational problem our applications solve and the sequential Barnes-Hut method are outlined in Section 3. Section 4 describes the available parallelism and the characteristics that make it challenging to exploit the parallelism effectively. The speedups obtained by using successful partitioning/scheduling techniques on an experimental high-performance shared-address-space multiprocessor are also presented in this section. Section 5 describes the simulated multiprocessor environment that we use for our experiments in the rest of the article. The importance of caching communicated data to obtain good performance is demonstrated in Section 6. Section

7 addresses the scaling of important application characteristics. Section 8 outlines the Fast Multipole Method and addresses for it all of the issues discussed earlier in the context of Barnes-Hut. Section 9 discusses the implications for communication abstractions. Finally, Section 10 summarizes the main conclusions of the article.

2. 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 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 propagate global information efficiently through the domain. Prominent among these algorithms are N-body methods, multigrid methods, multilevel preconditioners, adaptive mesh-refinement algorithms, and wavelet basis methods. Because these algorithms use fundamental physical insights to solve large-scale problems efficiently, and because they are naturally amenable to parallelization, it has been urged that parallel architectures be designed especially to support the communication needs of these algorithms [Chan 1990].

The class of applications we examine in this article, classical N-body simulations, studies the evolution of a system of particles (bodies) under the influences exerted on each particle by the whole ensemble.[1] The most time-consuming part of these simulations is the calculation of interparticle forces on potentials. If all pairwise forces are computed directly, this calculation has a time complexity that is $O(n^2)$ in the number of particles. Hierarchical, tree-based methods have recently been developed that reduce the complexity to $O(n \log n)$ [Barnes and Hut 1986] for general distributions, or even $O(n)$ for uniform distributions [Appel 1985; Greengard and Rokhlin 1987].

The particular form that the insight described above takes in N-body problems dates back to Isaac Newton in 1687: if the magnitude of interaction between particles falls off rapidly with distance (as it does in most physical interactions, such as gravitation or electrostatics with their $1/r^2$ force laws), then the effect of a large group of particles may be approximated by that of 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 of this insight—first used by Appel [1985]—implies that the farther away the particles, the larger the group that can be approximated by a single particle. Although Newton arrived at his powerful insight in the context of gravitation, hierarchical N-body techniques based on it have found increasing applicability in various problem domains. The domains include both classical N-body problems, in which the physical domain is actually

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

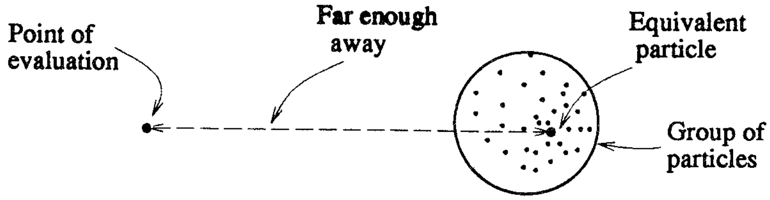


Fig. 1. Approximation of a group of particles by a single equivalent particle.

composed of bodies or particles, as well as others that can be formulated as N-body problems. Classical domains include astrophysics (gravitational force law), plasma physics (Coulombic), and molecular dynamics (Coulombic and others). Other domains include the vortex blob method in fluid dynamics [Chorin 1973], integral equations in boundary value problems [Greengard 1987], Cauchy integrals in numerical complex analysis, and—most recently—radiosity calculations in computer graphics [Hanrahan et al. 1991].

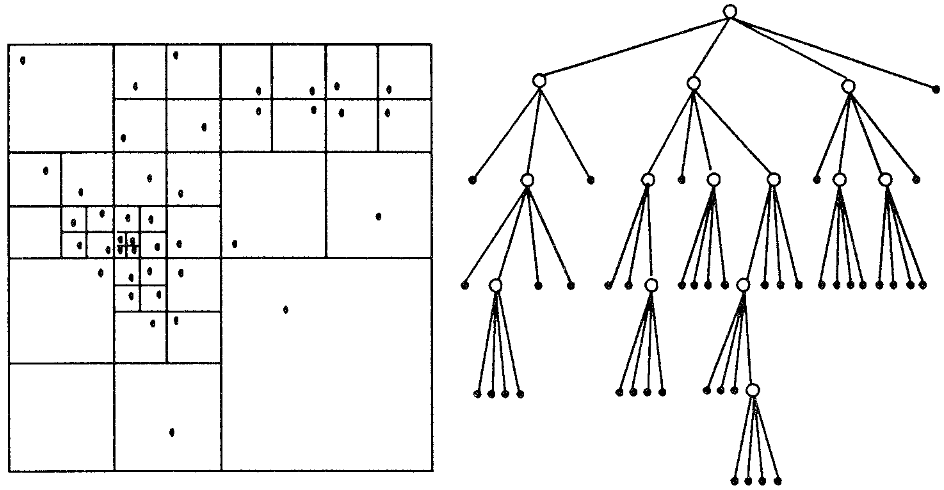
Particularly given the continued discovery of new domains of application, applications that use hierarchical N-body methods will clearly continue to be among the dominant users of high-performance computers. The two representative examples that we study in this article are galactic simulations from astrophysics. The first uses the Barnes-Hut method, and the second uses the Fast Multipole Method. The problem and the Barnes-Hut method used by the first application are described in the next section.

3. THE PROBLEM AND SOLUTION METHODS

We examine a classical N-body problem, which simulates the evolution of stars in a galaxy (or set of galaxies) under the influence of Newtonian gravitational attraction. The simulation proceeds over a large number of time-steps, every time-step computing the net force on every particle and updating its position and other attributes. Hierarchical N-body methods differ in the specific algorithms they use to compute interparticle interactions (forces).

All the hierarchical methods for classical problems first build a tree-structured, hierarchical representation of physical space, and then compute interactions by traversing this tree. The first of these methods was devised by Appel [1985]. However, his method is quite unstructured, which makes it difficult to program and to analyze for accuracy. The Barnes-Hut and Fast Multipole methods we study are better structured and hence more popular.

The tree that represents 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. The tree is therefore adaptive in that it extends to more levels in regions that have high particle densities. Since particles move and their distribution changes between time-steps, the tree is rebuilt in every time-step of the



(a) The Spatial Domain (b) Quadtree Representation

Fig 2. A two-dimensional particle distribution and the corresponding quadtree.

simulation. 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. Figure 2 shows a small two-dimensional example distribution and the corresponding quadtree. Let us now describe the Barnes-Hut method. The FMM will be discussed in Section 8.

3.1 The Barnes-Hut Method

The sequential Barnes-Hut algorithm divides every time-step into four phases: (1) computing the root cell dimensions and building the tree, (2) computing the cell centers of mass by an upward pass through the tree, (3) computing forces, and (4) updating the properties of the particles. A fifth, partitioning, phase is added in our parallel implementation. Let us look at the force computation phase in some detail, since it consumes more than 95% of the sequential execution time in typical programs.

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

$$\frac{l}{d} < \theta \tag{1}$$

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 accuracy parameter (θ is usually between 0.4 and 1.2). In this way, the tree traversal for a particle descends deeper in parts of the tree which represent space that is physically close to the particle, and groups other parts of the tree at a hierarchy of length scales. The time complexity of force calculation for all particles, and hence of the Barnes-Hut method, is $O(n \log n)$.

4. TAKING ADVANTAGE OF PARALLELISM

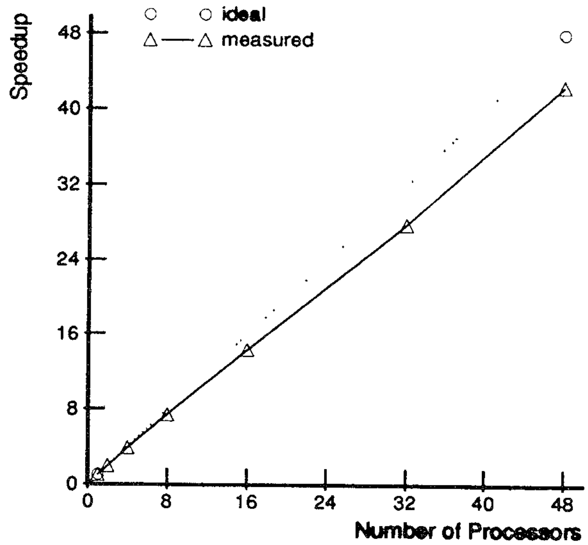
Until the discussion of communication abstractions in Section 9, our experiments in this article are conducted with parallel programs written for multiprocessors that support a shared address space. In both applications, we execute each of the phases within a time-step in parallel. In the Barnes-Hut application, the parallelism exploited in all phases is across particles, except in computing the cell centers of mass, where it is across cells.

Unlike many scientific applications that operate on uniform problem domains and use algorithms that require only localized communication[2] (see, for example, Singh and Hennessy [1992]), our N-body applications have several characteristics that make it challenging to obtain scalable parallel performance. Some of these characteristics have direct implications for architectural support, as we shall see. The characteristics include the following:

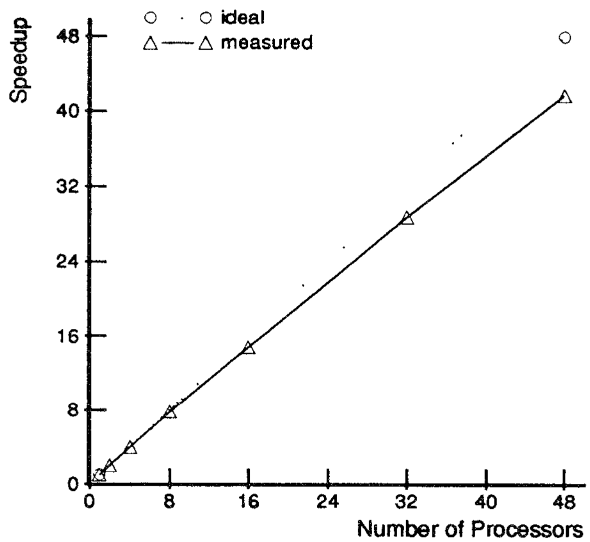
—The need for direct *long-range communication.*
—The *nonuniformity* of the physical domain, which has implications for both load balancing and communication.
—The *dynamically changing nature* of the particle distributions, which causes the work and communication distributions to change across time-steps.
—The fact that communication falls off with distance equally in all directions, which implies that a processor's partition should be spatially continguous and not biased in size toward any one direction, in order to minimize communication frequency and volume.
—The fact that different phases in a time-step have different distributions of work across particles/cells.

We have developed a successful partitioning scheme, called *costzones*, a description of which is deferred to Section 9. (Partitioning and scheduling issues for hierarchical N-body applications are discussed in detail in Singh et al. [1992a].) The scheme, which is inexpensive and hence invoked every time-step, assigns every processor a set of particles and cells which that processor is then responsible for in that time-step. To demonstrate that the scheme achieves the goals of load balance and data locality, Figure 3 shows the speedups obtained with both the Barnes-Hut and FMM applications on

[2] Even in applications that require long-range information propagation for the total solution, the information is often propagated indirectly—and inefficiently—via local communication. A classic example is the use of Jacobi or SOR iteration to solve elliptic partial differential equations.



(a) Barnes-Hut, 32K particles, theta = 0.7



(b) FMM: 32K particles, 24 terms

Fig. 3. Speedups on the Stanford DASH multiprocessor.

an experimental multiprocessor, the Stanford DASH multiprocessor, which has a multilevel memory hierarchy with highly nonuniform access costs [Lenoski et al. 1990]. The initial distribution in both cases is a pair of nonuniform Plummer model [Aarseth et al. 1974] galaxies that interact with each other. In these and all results we present, we run five time-steps of the galactic simulation, but ignore the first two time-steps to factor out cold-start

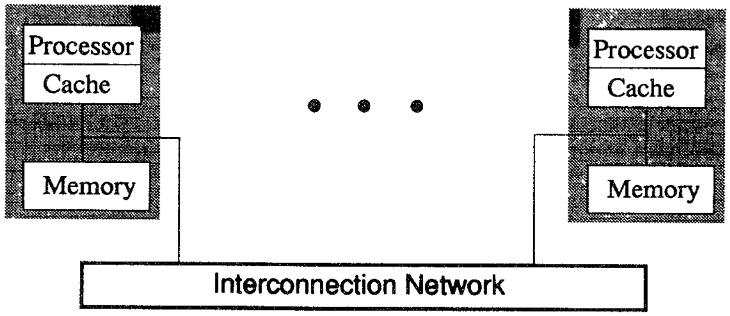


Fig. 4. The simulated multiprocessor architecture.

effects that would be negligible in real runs that execute for many hundreds of time-steps. Similar results were obtained with other distributions and reasonable problem configurations.

By understanding the structure of communication and data referencing in an application, we can determine the types of locality in an architecture that the application can and cannot exploit effectively. We can also study which types have the greatest impact on performance. Before we do this for Barnes-Hut, let us describe the multiprocessor environment we use to obtain these and the rest of our results in this article, and the kinds of locality that this multiprocessor affords.

5. THE SIMULATED MULTIPROCESSOR ENVIRONMENT

Real multiprocessors such as DASH have fixed configurations and are thus inappropriate for our studies in this article. We therefore use a multiprocessor simulator which is composed of two parts: the Tango event-driven reference generator (described in Goldschmidt and Davis [1990]) and a memory system simulator that feeds back into Tango. We simulate a very general shared-address-space architecture, consisting of a number of processing nodes connected together by some general interconnection network (see Figure 4). A processing node consists of a processor, a cache, and an equal fraction of the total physical (logically shared) memory on the machine. That is, memory can be directly referenced by any processor, but is physically distributed. A cache holds shared as well as private data, and shared data that are cached are kept coherent by a hardware distributed-directory mechanism [Lenoski et al. 1990]. This architecture affords locality at three levels of its memory hierarchy:

—*Cache locality*: This is the reuse of data that a processor brings into its cache (whether from its own local memory unit or from across the network), before they are replaced or invalidated. It includes the reuse of the particular data words that have been referenced before (temporal locality), as well as the prefetching afforded by multiword cache lines (spatial locality). Temporal locality is obtained by partitioning and scheduling the application appropriately, and spatial locality is obtained by also organizing data structures to take advantage of multiword cache lines.

—*Memory locality*: If references miss in the cache, one would like to have them satisfied in the local memory unit rather than communicate across the network. Memory locality can be exploited in two ways: (1) by distributing data across physical memory units—either statically or dynamically—so that data are allocated on the memory unit closest to the processor that accesses them most often and (2) by replicating data in main memory to exploit temporal locality—particularly when the hardware caches fail to do so for capacity or conflict reasons—thus using local memory as a software-controlled cache for communicated data. Both distribution and replication are typically done at the granularity of physical memory pages.

—*Network locality*: If references do have to go across the network, one would like them to go as close as possible to the issuing processor in the network topology. Network locality is obtained by mapping tasks to processors so that the communication topology in the application maps well onto the physical topology of the network.

Network locality is not a very significant issue in today's machines, since the time for a packet to get into 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 cache and/or memory locality are exploited effectively, and techniques to exploit network locality are usually orthogonal to those that exploit cache and memory locality. We therefore ignore network locality in this article.

While we vary latencies when necessary, the base latencies we assume for memory references are as follows. Cache hits cost a single cycle; read misses that are satisfied in the local memory unit stall the processor for 15 cycles; and read misses satisfied in some other memory unit stall the processor for 60 cycles. Write miss latencies can be hidden very effectively by hardware techniques [Gharachorloo et al. 1991], so that they rarely stall the processor in most applications; we therefore assume that local write misses cost a single cycle and remote write misses 3 cycles.

6. THE IMPORTANCE OF TEMPORAL LOCALITY ON COMMUNICATED DATA

In this section, we show that the reuse obtained by caching shared data (from local or remote memories) is the key form of locality in the Barnes-Hut application, and that this locality is exploited both automatically and very effectively by caches on shared-address-space machines. On the other hand, data distribution in main memory—to allocate the particle/cell data assigned to a processor in that processor's local memory—is both difficult to implement and not nearly so important. We shall demonstrate similar results for the FMM in Section 8.1.

We demonstrate the importance of temporal locality on communicated data first qualitatively, by describing the structure of communication in the application, and then quantitatively through simulation. In Section 9, we shall discuss how different architectural or programming models exploit temporal locality, and shall argue further for the advantages in this regard of shared-address-space architectures that cache shared data.

6.1 Structure of Communication

The interprocessor communication patterns in these applications are different in different phases of computation. The vast majority of time in each application, however, is spent in the force calculation phase. In the Barnes-Hut application, the force calculation for a particle/cell requires reading position and mass information from many other particles/cells that are potentially not in its processor's partition. However, the information thus shared is not modified during the force calculation phase, but only later on in the update phase of the time-step. The nonlocal data read to compute the force on one particle/cell can therefore be replicated (cached) and reused locally to compute the forces on other particles/cells in the processor's partition. Partitioning schemes that preserve contiguity of partitions in physical space, such as the ones we use, reduce the amount of data that a processor needs to replicate and enhance the degree of temporal locality.

Redistributing data in main memory as partitions change, on the other hand, is difficult in these applications. The applications share data at a fine granularity: that of individual particles or cells. The particles and cells that are in a processor's partition are close to one another in physical space, but not necessarily in the shared array data structures that hold the particle and cell data. This is not a problem for caches in hardware-coherent machines, since the unit of data transfer (the cache line size) is relatively small. However, because of the large granularity of the pages that are the units of data movement in main memory, redistributing data in main memory across time-steps requires rearranging the layout of particles and cells in the shared arrays.[3] Besides this complexity of implementation for the application programmer, data redistribution is also very expensive, both because of the overhead of rearranging data structures and because redistribution involves invoking the operating system. (For the same reason as for granularity, replicating at the page level in main memory causes fragmentation and unnecessary communication.)

6.2 Measurements and Discussion

To demonstrate the importance of exploiting temporal locality on communicated data, we examine the benefits obtained by caches in our simulated multiprocessor architecture (Section 5), as well the benefits obtained by dynamic data redistribution in main memory. Note that we only redistribute,

[3]The alternative of providing support in the multiprocessor for data migration in main memory at the granularity of program objects (particles/cells) has drawbacks too: the memory overhead associated with the software tables needed to maintain information about these object-sized units can be very large when the objects are small (see Chapter 9 as well).

and do not replicate, data in main memory. All replication, and hence temporal locality, on communicated data is handled by the caches.

Despite the difficulties and overheads associated with data redistribution, as described above, we give all possible (unrealistic) allowances to data redistribution in our experiments. Our simulator allows us to redistribute data at any granularity and at no cost whatsoever.

We run a fixed problem for an application on a fixed number of processors (16), but vary the per-processor fully associative cache size.[4] For each cache size, we run two versions of the application: one in which we do not distribute shared data explicitly at all, but allow pages of memory to be allocated in a round-robin interleaved manner among processing nodes, and the other in which shared data are redistributed every time-step (at no cost). Data that are private to a processor are allocated in that processor's local memory in both cases.

The execution times for the Barnes-Hut application, normalized to the execution time with infinite caches and appropriate dynamic data distribution, are shown in Figure 5. Figure 5(a) shows the normalized execution times for the reference latencies we used in our earlier experiments (see Section 5). The results clearly reveal the importance of temporal locality as obtained by caching. Even for these reference latencies, not having caches (actually, having 8-byte caches) degrades performance by about a factor of 3 over having large-enough caches. There is a sharp knee in the curve at a critical cache size (between 14KB and 16KB in this case). Caches larger than this critical size exploit temporal locality successfully, while smaller caches substantially degrade performance. The amount of private data that needs to be replicated in the caches is very small, so that the improvements shown in the figure (at least from the 4KB data point on) are mostly due to caching shared data. In Section 7.5, we shall show that the critical working sets and hence cache sizes needed for typical problems are relatively small and scale slowly with problem and machine size, so that there is no need for replication in main memory. Figure 5(a) also shows that, regardless of whether caches are or how large they are, even unrealistically free data redistribution helps performance only marginally.

The read latencies assumed in Figure 5(a) have a larger ratio between cache hits and cache misses that are satisfied in local memory (1 : 15 cycles) than between cache misses satisfied in local memory and cache misses satisfied in remote memory (15 : 60 cycles). If the latter ratio is made larger, the benefits of redistribution are increased. However, the benefits of caching nonlocal (communicated) data are increased much more. Figure 5(b) shows results for a remote read latency of 500 cycles, a remote-to-local ratio which is very large compared to those in shared-address-space machines that people are contemplating to build, and which therefore exaggerates the impact of data redistribution. Even in this case, the benefits of (free) data redistribution are only about a 10% increase in performance, whereas the difference

[4] The use of fully associative caches is justified in Section 7.5.1.

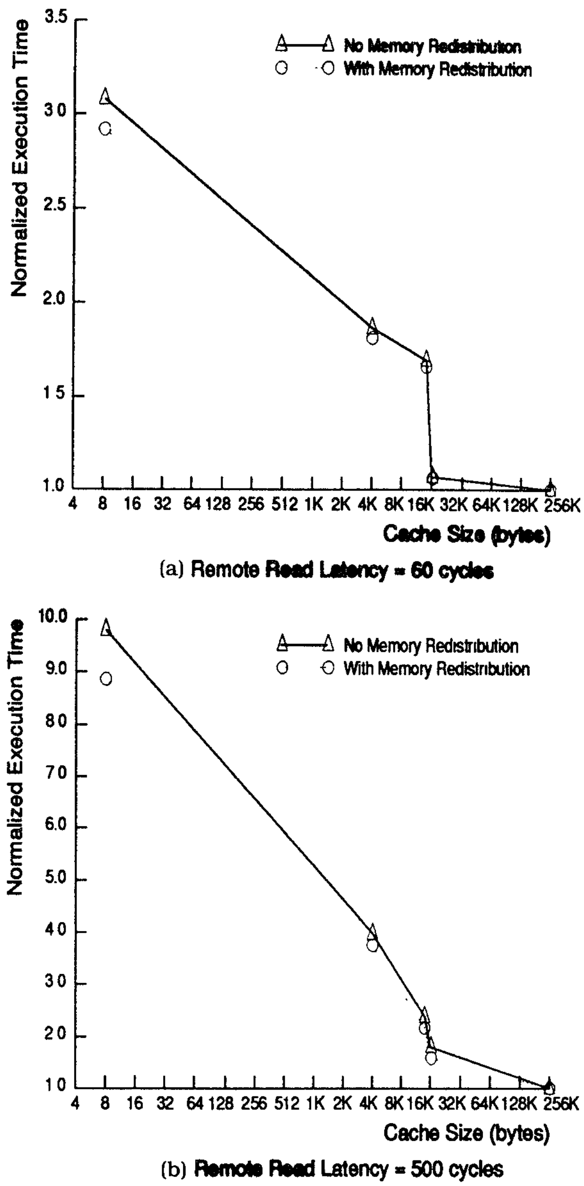


Fig. 5. Normalized execution time for the Barnes-Hut application: $n = 8192$, $\theta = 1.0$, $p = 16$.

between caching and not caching is over 500%. Similar results were obtained with other problem sizes and numbers of processors.

The reasons for the above results follow from the communication structure and granularity of the application, as described in Section 6.1. A key property that inhibits the benefits of data distribution (ignoring its cost and complexity) is that, regardless of whether caches are large enough or not, most of the shared references that miss in the cache are to parts of the tree that are in other processors' partitions. The cost of these misses is not alleviated by redistribution, but only by replicating and thus reusing communicated data.

7. IMPACT OF SCALING PROBLEM AND MACHINE SIZE

As larger and more-powerful machines become available, scientists will want to simulate larger physical systems. It is therefore very important for architects to understand how problems are likely to be scaled and the implications of scaling for the effectiveness and design of larger machines. Specifically, two important implications of scaling are: (1) how do the communication-to-computation ratios that a machine must support scale with the number of processors and (2) how do the sizes of a processor's important working sets, which determine the amount of cache or perhaps local memory it needs for ideal cost-effectiveness, scale? The answers to these questions are likely to vary across problem domains and applications. The way to study scaling, therefore, is through careful analyses of important applications or application classes. In this section, we perform such an analysis for the Barnes-Hut application. We assume that a multiprocessor is scaled by adding identical processors to it, each added processor having the same amount of main memory as the others already in the ensemble.

In another article [Singh et al. 1993], we discussed methodological issues in studying the scaling of parallel applications and architectures. We argued that the scaling of real applications is more complicated than the common practice in which people consider only the effects of scaling the input data set size on the computation and communication complexities of individual algorithms. Realistic scaling studies must incorporate considerations imposed by the applications that use these algorithms. In particular, other application parameters often need to be scaled along with the data set size to meet the application user's goals in running larger problems.

For example, in the large class of scientific applications that simulate physical phenomena, a primary goal of running larger problems is to reduce the errors introduced by various approximations made in the simulation. The approximations include the temporal and spatial discretization of the problem domain and the limited accuracies of the numerical algorithms used. Each of the approximations is usually represented by a different application parameter. The data set size, for instance, often represents the spatial resolution of the model, and increasing this size reduces the error contribution owing to spatial discretization. However, this is only one source of error, and reducing it will often call for the reduction of other errors as well in order to efficiently reduce overall simulation error.

While error is a dominant consideration in determining how to scale parameters in many scientific applications, the specific guiding principles for scaling parameters might be different in different domains. For example, if errors due to different sources combine in analytically predictable ways, it may make sense to scale in order to minimize the overall simulation error subject to a constraint on execution time or memory usage, or to minimize time or memory usage subject to a bound on simulation error. In gravitational N-body simulations, however, errors do not combine very predictably. We therefore use a scaling principle that is considered most realistic in practice for this as well as many other classes of scientific applications (personal communication, J. E. Barnes, May 1991) [Singh et al. 1993]:

—*All sources of error should be scaled so that their error contributions are about equal.*

This scaling principle addresses the question of *how* an application's parameters should be scaled relative to one another. It is independent of *how much* an application is scaled, that is, of the constraints (on execution time, memory usage, etc.) under which an application is scaled to use a larger machine. In our study, we examine the following three scaling models that are generally accepted in the literature for the constraints under which an application might be scaled to use more processing nodes (processors and memory).

—*constant problem size scaling* (CPS): Keeping all application parameters constant when more processors are used. (The problem size is specified by the particular input parameters that the application is run with.) The assumption is that the user simply wants to solve the same problem faster.

—*memory-constrained scaling* (MC): Scaling the problem so that the maximum amount of physical memory required on a single processing node remains constant. The assumption is that the user always wants to run the largest data set possible without overflowing any processor's memory.

—*time-constrained scaling* (TC): Scaling the problem so that the machine takes the same amount of absolute time to solve the problem as processors are added. Thus, the user wants to solve the largest possible problem in the fixed amount of time at hand.

Of course, no one model can be claimed to be the most realistic for all applications and all users. Users are in fact unlikely to follow any model very strictly. However, these models constitute simple and useful tools for an analysis of scaling.

Using the above scaling principle and models, we employ the following methodology—as in Singh et al. [1993]—to study scaling issues in our applications. We first understand the relationships between application parameters in terms of their error contributions, and use our equal-error principle to develop a rule for scaling the parameters relative to one another. Then, we examine how memory requirements and time complexity depend on different parameters. This allows us to understand how the parameter scaling rule interacts with different scaling models (constraints). Finally, we examine how the execution characteristics of interest scale with the number of processors under the appropriate rule and models.

7.1 Relative Scaling of Application Parameters

Discrete N-body simulations have several sources of error in their approximation of a continuous physical process. For the collisionless gravitational problems to which hierarchical N-body techniques are well suited, these sources include the following in the Barnes-Hut application: (1) Monte-Carlo sampling of phase-space to approximate the system by a finite number of bodies (spatial discretization), parametrized by the number of bodies n; (2) the use of a discrete nonzero time-step in integrating the equations of motion (temporal discretization), parametrized by the time-step duration Δt; (3) approximations made in force computation within a time-step, parametrized by the Barnes-Hut accuracy parameter θ; (4) force softening to render the force computation expression $F = Gm_1 m_2/(r^2 + \epsilon^2)$ nonsingular, parametrized by ϵ; and (5) roundoff due to finite word length in the computer. The last source is a static machine characteristic, and we ignore it. Studies in the astrophysics community have investigated the impact of the other parameters on simulation accuracy [Barnes and Hut 1989; Hernquist 1987]. While some of the error contributions are not always completely independent, the following rules emerge as being generally valid in interesting parameter ranges.

—n: The error from the increased relaxation rate due to Monte Carlo sampling scales as $1/\sqrt{n}$; thus, an increase in n by a factor of s leads to a decrease in simulation error by a factor of $\sqrt{s}$.

—Δt: The leap-frog method used to integrate the particle orbits has a global error of the order of Δt^2. Thus, a reduction in error by a factor of $\sqrt{s}$ (to match that due to an s-fold increase in n) requires a decrease in Δt by a factor of $\sqrt[4]{s}$, and hence that many more time-steps to simulate a fixed amount of physical time, which is usually held constant.

—θ: The results in Hernquist [1987] and Barnes and Hut [1989] demonstrate a scaling of the force calculation error proportional to θ^2 in the range of practical interest, for common distributions. (This is for the original Barnes-Hut algorithm—the one we use—which does not incorporate quadrupole corrections to the force approximation. If quadrupole corrections are incorporated, the error scales as θ^4.) Reducing the error by a factor of $\sqrt{s}$ thus requires a decrease in θ by a factor of $\sqrt[4]{s}$.

—ϵ: The error due to force softening can be ignored if the value of ϵ is smaller than the average interparticle spacing, as it usually is.

The rule for scaling problem parameters together that results is:

Barnes-Hut Scaling Rule: If n is scaled by a factor of s, then Δt and θ should each be scaled by a factor of $1/\sqrt[4]{s}$.

Let us now examine how each of the parameters separately impacts the serial computational complexity and storage requirements of the applications. Since we used the Barnes-Hut application as our primary example in the scaling methodology article [Singh et al. 1993], some of what follows can also be found in that article. However, we repeat it here for completeness.

7.2 Scaling of Memory Requirements and Computational Complexity

Memory. The main sources of memory requirement in the Barnes-Hut application are the data for particles and tree cells. The former require memory proportional to n. The latter depend on the spatial distribution as well as n, but are also found to be proportional to n for distributions of interest [Hernquist 1987]. The other two parameters we scale (θ and Δt) do not affect the data requirements of the application in a shared address space.

Time Complexity. The serial time complexity depends on n in an $O(n \log n)$ fashion for realistic ranges of θ. The dependence on θ is found to be roughly proportional to $1/\theta^2$ for fixed n [Hernquist 1987]. Finally, the complexity can be assumed to be proportional to the number of time-steps, that is, inversely proportional to Δt when a fixed amount of physical time is being simulated (as is usually the case). Thus, an increase in the input data size (n) by a factor of s from a base size n_0 leads to a storage requirement increase of the same factor, but a serial complexity increase of a factor of

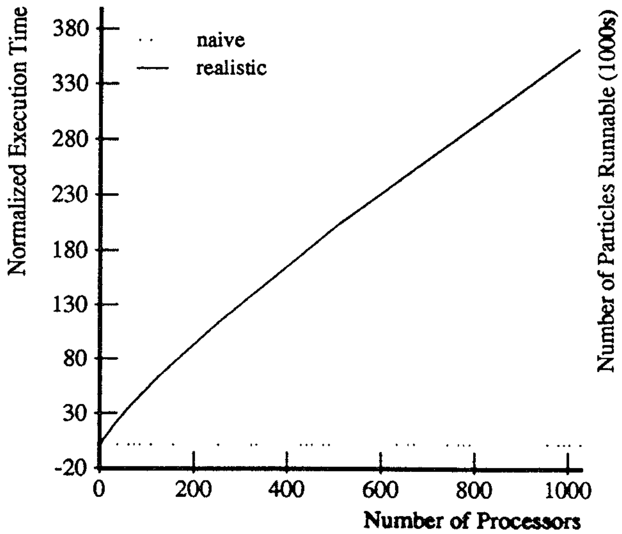
$$s * \left(1 + \frac{\log s}{\log n_0}\right) * \sqrt[4]{s} * \sqrt{s}$$

under our scaling rule, if we make the generally valid assumption—within useful parameter ranges—that the execution time scales independently with n, θ, and the number of time-steps. The first two terms in the above expression are due to an increase in n by a factor of s, the third due to an increase in the number of time-steps by a factor of $\sqrt[4]{s}$, and the last due to a decrease in θ by a factor of $\sqrt[4]{s}$.

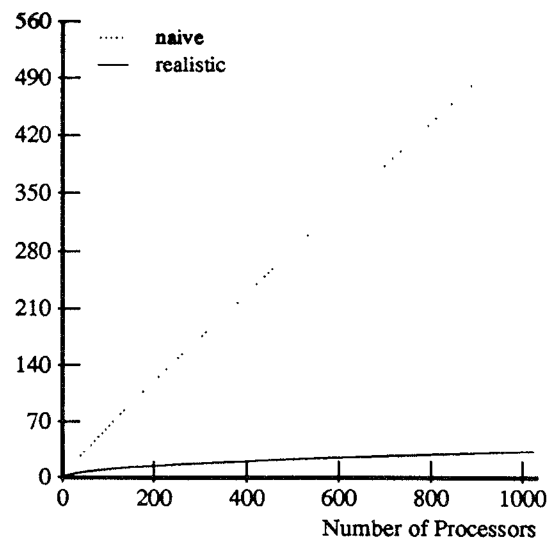
7.3 Impact on Memory-Constrained and Time-Constrained Scaling

The issue of scaling problem parameters is irrelevant to constant problem size scaling. Let us see how the above effects of scaling the parameters interact with the memory- and time-constrained scaling models. Specifically, we examine how the parallel execution time grows under memory-constrained scaling and how the data set size (or number of particles that can be simulated) grows under time-constrained scaling.

Execution Time Under Memory-Constrained Scaling. Under memory-constrained scaling, an increase in the number of processors by a factor of k gives us k times as much memory and allows us to run a problem with k times as many bodies (assuming a shared address space with no replication of data in main memory; see Section 9.6.2 for a discussion of how this changes for message passing). This is because increasing the number of processors does not affect the shared-data requirements, and the amount of per-processor private memory used is negligible in comparison. Under naive scalability analysis (scaling only the number of bodies n), the parallel execution time would increase only slightly—by a factor of $kn_0 \log(kn_0)/k * n_0 \log n_0$ or $(1 + \log k/\log n_0)$, where n_0 is the number of bodies run on the original, smaller machine—assuming perfect speedup with no communication or other overheads due to parallelism. Under our more realistic parameter scaling



(a) Memory Constrained



(b) Time Constrained

Fig. 6. Impact of realistic versus naive scaling.

rule, however, an additional factor of $\sqrt[4]{k} * \sqrt{k}$ or $k^{3/4}$ applies (see Section 7.2). This additional factor is much larger than that already yielded by the naive analysis, and in fact constitutes an unacceptable increase in execution time under memory-constrained scaling (see Figure 6(a)); scaling from one to 1024 processors, for example, would increase the parallel execution time by a factor of 300, even if perfect speedup were achieved. Moving from naive to realistic scaling turns the problem from perhaps being memory bound to being clearly computation time bound. *Time-constrained scaling is therefore likely to be more important than memory-constrained scaling in practice.*

Data Set Size Under Time-Constrained Scaling. If only the number of particles, n, were scaled under time-constrained scaling, the $O(n \log n)$ complexity of the algorithm suggests that n would scale almost linearly with the number of processors, i.e., almost as fast as under memory-constrained scaling. Because other parameters have to be scaled, however, *the number of particles that can be simulated in a fixed amount of time grows much more slowly* (Figure 6(b)). Under our equal-error scaling rule, for example, if s and k are the factors by which the number of runnable particles and the number of processors grow, respectively, then s is related to k by the following expression (assuming perfect speedup):

$$k = s^{7/4}\left(1 + \frac{\log s}{\log n_0}\right),$$

where n_0 is the number of particles run on the smaller machine. That is, s grows by a factor that is closer to $\sqrt{k}$ than to k. Thus, if a uniprocessor can run a problem with 16K particles in one day, then a machine with 1024 processors would run a problem with only about 700K particles in the same time, rather than the almost 10 million particles that scaling only the data set size would predict.

The realistic scaling of parameters has already led us to two different results than scaling only the data set size: first, that memory-constrained scaling is unrealistic since it causes computation time to grow too quickly, and second, that the number of particles that can be simulated in a fixed amount of time grows much slower with computational power than the $O(n)$ complexity in terms of data set size might suggest. Let us now examine how the inherent communication in the application—that is, the minimum communication-to-computation ratio that an architecture must sustain—scales with machine size under the different scaling models.

7.4 Scaling of Inherent Communication

The communication-to-computation ratio of a parallel application is an important determinant of how efficiently a multiprocessor can run the application. Beyond a threshold of easily sustainable communication, higher ratios lead to deteriorating performance. In many scientific applications, the primary determinant of the communication-to-computation ratio is the data set size per processor, i.e., the ratio is inversely related to n/p. Common examples are applications that partition a uniform grid into contiguous subgrids, one per processor, with a processor computing at all elements in its partition but communicating only at interpartition boundaries. Since memory-constrained scaling usually keeps the data set per processor constant in a shared address space, it keeps the communication-to-computation ratio constant, promising constant per-processor efficiency as larger machines run "proportionally larger" problems. Such results were shown in the Sandia experiments

[Gustafson et al. 1988]. However, we showed in Singh et al. [1993] that primarily because the data set size does not grow linearly with the number of processors under the most appropriate scaling model—time-constrained scaling—the communication-to-computation ratio becomes worse under this model. In this section, we study the communication-to-computation ratio for Barnes-Hut in more detail.

7.4.1 *Experimental Methodology.* Communication in a multiprocessor arises from two sources: the inherent communication required by the parallel program and communication that is an artifact of how the program's data requirements or referencing patterns interact with the particular memory system organization. On shared-address-space multiprocessors such as the ones we use, the artifacts include false sharing of large cache lines, replacement of communicated data in finite caches, and inappropriate allocation of memory on physically distributed nodes. Since we are interested in inherent communication behavior in this section, we would like to eliminate these artifacts. To virtually eliminate false sharing, we use a small cache line (8 bytes) in our simulated multiprocessor (Section 5). To eliminate replacements and the impact of memory allocation after the cold-start period, we simulate infinite per-processor caches. And to eliminate cold-start misses, which are negligible in realistic runs that last many hundreds of time-steps, we do not measure the first two time-steps of the computation.

We measure the communication-to-computation ratio as the average number of read misses per 1000 busy cycles per processor. In the multiprocessor model described above, read misses represent the fundamental communication required to solve the problem (in fact, they represent the minimum amount of inherent communication that would be required in any communication abstraction). Write references can also generate communication traffic in a real cache coherence protocol, but they still do not affect the trend for communication-to-computation ratios with infinite caches in these cases.

7.4.2 *Results.* Figure 7 shows how the communication-to-computation ratio in the Barnes-Hut application scales under time and memory-constrained scaling. "Naive" scaling in the figure refers to changing only the number of particles (data set size) without scaling other parameters. For all the curves in the figure, the base problem size on a single processor simulates 128 particles with $\theta = 1.0$ and $\Delta t = 0.025$. Since this base problem is very small (so we could simulate the scaled problems easily), we also examined the trends with other base problems sizes as well as by factoring out load imbalance; the trends were very similar. Clearly, the communication-to-computation ratio grows under realistic TC scaling.

Communication is very difficult to model accurately in this application, particularly for a nonuniform distribution. We therefore qualitatively explain how the communication-to-computation ratio depends on different parameters and use this explanation to understand the trends seen in the figure for the different scaling methods. The parameters that we use in our discussion are the granularity of the partitions (parametrized by the average number of particles, n_g, assigned to a processor in every time-step; $n_g = n/p$), the

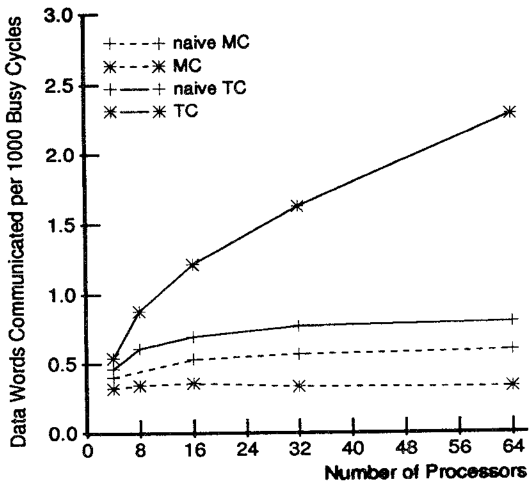


Fig. 7. Communication-to-computation ratio under TC and MC scaling.

spatial extent of the partitions (parametrized by $1/p$, the reciprocal of the number of processors), the force calculation accuracy parameter θ, and the time-step interval Δt. The spatial extent of a partition is independent of n since the input spatial dimensions of the galaxies—and in fact even the nature of the particle distribution—are independent of n: larger n correspond simply to higher-resolution samplings of the galaxies.

Granularity n_g. As the granularity, n_g, increases for fixed p, θ, and Δt, the amount of computation per processor scales as $O(n_g \log n)$. Communication scales more slowly, however, primarily because most of the communicated data are heavily reused by other particles in the partition. Also, most of the new data referenced by a particle due to an increase in n are relatively close to the particle and therefore quite likely to be in the particle's own partition. These are data from cells that are not far enough away to be approximated by their center of mass, but that earlier had only a single particle and now have more (and are hence opened).

Accuracy θ. The communication-to-computation ratio also improves with increased force calculation accuracy (decreased θ) for the same n, p, and Δt, for the same reasons of data reuse.

Processors p. For the same n_g, θ, and Δt, the amount of communication increases with p (i.e., under naive MC scaling), both because $\log n$ increases and because more of the data that a processor references are not in its (now spatially smaller since p is larger) partition. Computation in this case increases as only $O(\log n)$, and the result is a slow increase in the communication-to-computation ratio under naive MC scaling.

Time-Step Interval Δt. Finally, as Δt is refined, the spatial distribution changes more slowly across time-steps, which should reduce communication a little. These trends can be used to explain the results for different scaling models in Figure 7 as follows.

Under CPS scaling (not shown in the figure), both n_g and $1/p$ decrease rapidly, while other parameters are held constant. The communication-to-computation ratio therefore increases quickly. We have seen above that naive MC scaling (i.e., scaling p while keeping n_g, θ, and Δt fixed) is not enough to keep the ratio constant in this application, but results in a slow increase (unlike in the Sandia applications [Gustafson et al. 1988]). Keeping the ratio in check in fact takes realistic MC scaling, in which case θ and Δt also decrease while n_g still stays fixed, and the ratio actually falls a little with scaling. However, we have also seen that MC scaling is a highly unlikely method for this application in practice since execution time increases rapidly.

The results of greatest practical interest are those for TC scaling, in which case also the trends for naive and realistic scaling are different. Naive TC scaling is quite close to naive MC scaling in this case (with only a log n difference owing to the $O(n \log n)$ complexity), so that the communication-to-computation ratio in this case increases almost as slowly. Realistic TC scaling, however, causes n to grow much more slowly (see Section 7.3) and therefore n_g to decrease quite quickly as p increases. The communication-to-computation ratio increases more quickly than under naive TC scaling, though still not very quickly, even though the rate of increase is inhibited by the scaling of θ and Δt.

Under the most appropriate scaling model, then, the communication-to-computation ratio increases slowly in the Barnes-Hut application as larger problems are run on larger machines. However, when appropriate partitioning and scheduling techniques are used [Singh et al. 1992a], the ratio remains very small for realistic problems on today's machines. For example, a typical simulation today uses 64K particles with $\theta = 1.0$ and quadrupole moments (personal communication, J. E. Barnes, May 1991). When run for 512 time-steps, this problem takes about three days to run on a single processor of an SGI 4D/240. When run on a 64-processor machine, this problem has a communication-to-computation ratio of less than 1 double-precision word per 2000 instructions. The same problem running on 1024 processors has a ratio of 1 double-precision word per 170 instructions. The largest problems people run today are on the order of several million particles, and these are run on machines with fewer than 1024 processors (using quadrupole moments rather than just centers of mass to represent cells); for these problems, the ratios are on the order of a double-precision word per several thousand instructions. Since there is also abundant concurrency and no other major problem for parallel force computation, we can continue to expect good parallel performance for quite a while from the Barnes-Hut application as larger parallel machines are built. The problem likely to limit speedup first is the performance of the tree-building phase, as discussed in Singh et al. [1992a] and Holt and Singh [1995].

Let us now examine the scaling of the other important architectural parameter we study in this article: the size of a processor's working set.

7.5 Scaling of Working-Set Size

As discussed in Section 5, modern multiprocessors are built with hierarchical memory systems, in which smaller, faster levels of the hierarchy can serve as hardware- or software-managed caches on successively larger, slower levels. An important question facing architects of large-scale multiprocessors is how large the caches or memories at each level of the hierarchy should be for good cost performance. A part of the answer is determined by practical issues such as hardware cost or packaging constraints. A crucial part of the answer, however, should depend on how much capacity the applications likely to run on these machines require at different levels for effective performance and how these requirements scale as larger problems are run on larger machines. In this section, we address this question for our hierarchical N-body applications by examining the scaling of the different *working sets* in the applications.

The working-set model of program behavior [Denning 1968] is based on the temporal locality exhibited by the data-referencing patterns of programs. Under this model, a program has a set of data that it reuses substantially for a period of time, before moving on to other data. The shifts between one set of data and another may be abrupt or gradual. In either case, there is at most times a "working set" of data that a processor should be able to maintain in a fast level of the memory hierarchy, to use that level effectively. We have found that many scientific applications in fact have a hierarchy of well-defined working sets, the ability to contain each of which in a given level of the memory hierarchy has its own impact on performance.

Consider, for example, the first level of the memory hierarchy external to the processor: the first-level cache. We saw in Section 6 that a cache is a key performance-enhancing resource for N-body applications on shared-address-space multiprocessors. A cache is an expensive resource, however. For a given application, and in the absence of time-sharing, ideally the cache should be just large enough to hold a working set that (1) is of a size that can be expected to fit in a processor cache and (2) can substantially aid overall program performance. How the size of this working set scales therefore directly impacts the effectiveness of different cache sizes and determines how the cache size should be scaled on larger machines for optimal cost effectiveness. The working-set hierarchy also tells us which working sets can be expected to fit in which levels of the cache hierarchy.

In message-passing architectures, a processor's local memory is often used as a software-controlled cache for communicated data, in addition to holding the processor's own assigned data partition; that is, data are replicated in the memories of those processors that need to use them. In addition to the active working sets for caches, then, message-passing programs also have a "working set" of communicated data that helps determine the size and scaling of a processor's local memory. We focus on active working sets for caches here and leave a discussion of message-passing machines to Section 9.

7.5.1 *Experimental Methodology.* We measure working sets using the simulated architecture described in Section 5, again with a small 8-byte cache line size. As in Section 6, for the same problem size and number of processors, we vary the per-processor fully associative cache size and plot the execution time—normalized to that obtained with infinite caches—versus the cache size used. For every important well-defined working set, there should be a knee in the execution time versus cache size curve at the cache size corresponding to that working set. Cache sizes are varied at a fine granularity to identify working sets precisely.

Caches on shared-address-space machines suffer misses from four sources: (1) *communication*, both inherent and due to false sharing; (2) replacement of useful data due to limited cache *capacity*; (3) replacement due to *conflicts* within the cache caused by an associativity smaller than the cache size; and (4) *cold-start* effects. As in our earlier experiments, the cold-start period is ignored by starting the measurements after the second time-step. By using fully associative caches with LRU replacement, we omit conflict misses in the cache and include only communication and capacity misses. While fully associative caches are unrealistic in practice, they provide us with a clean measure of working-set size, untainted by the many low-level artifacts that can cause cache conflicts. We do, however, comment on results obtained with low-associativity caches as well. Once again, since the problem sizes we simulate are very small, we verified our scaling trends by examining some larger base problems as well as by factoring our load imbalance. Let us now examine the results for the two applications.

7.5.2 *Results.* Figure 8 shows the results for the Barnes-Hut application for a fixed problem size and number of processors. The ordinate is normalized so that 1.0 represents the execution time with infinite caches.

There are clearly three levels in the working-set hierarchy in this application. The first, or level-1 working set (lev1WS), is the amount of temporary storage used to compute an interaction between a particle and another particle/cell, and reused in successive interactions. Its size is very small, is independent of n, θ, or the number of processors p, and depends only on the kind of moment used to compute an interaction. The lev1WS is about 0.5KB in size when centers of mass are used, and 0.7KB when quadrupole moments are used. Having a cache large enough to hold the lev1WS reduces the read miss rate from 100% with no cache to about 20% in most cases we have simulated. While this is a large reduction, the miss rate is still not low enough for effective performance since most of the remaining misses are to nonlocal data (see Figure 8).

The second, or level-2 working set (lev2WS), is the most important working set in the application. It is very sharply defined and takes the normalized execution time from over 1.8 to less than 1.05 within a range of 2–4KB in cache size. For the small problem shown in Figure 8, the size of the lev2WS set is about 11.5KB. We shall discuss its scaling shortly.

Beyond the lev2WS, performance improves much more slowly with cache size until the cache size reaches the lev3WS. The size of this working set is roughly the maximum of (1) the amount of data in a processor's partition and (2) the amount of data a processor needs to compute the forces on all the

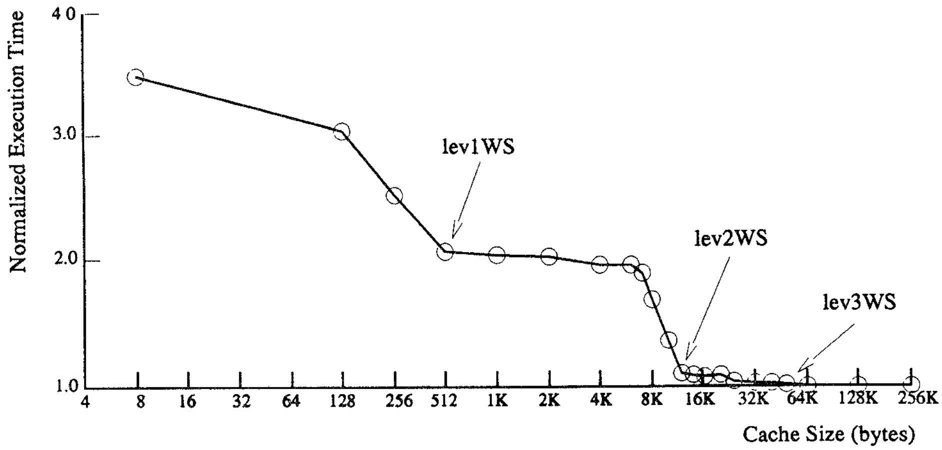


Fig. 8. Workings sets for the Barnes-Hut application: $n = 1024$, $\theta = 1.0$, $p = 4$.

particles in its partition. Thus, the lev3WS size decreases with increasing number of processors and increases with increasing n and decreasing θ. However, since the lev3WS marks the culmination of a slow decrease in miss rate, and since the capacity miss rate is already very small after the lev2WS is accommodated, it is not important to performance, and we do not consider it further. Let us look at the important level-2 working set more carefully (in the rest of the Barnes-Hut discussion, we refer to lev2WS as the important working set or simply the working set).

An interesting, initially somewhat surprising, observation about the important level-2 working set is that its size is independent of the number of processors used. That is, although a processor's partition of the data set diminishes rapidly under CPS scaling, the size of its important working set stays constant. From another perspective, although the communication-to-computation ratio with infinite caches increases substantially under CPS scaling, as we saw in Section 7.4, the cache size required to approximate an infinite cache stays the same. This result runs counter to what might be expected from many scientific applications, in which a problem domain is decomposed and in which a processor only needs data from its own subdomain and from the borders of its neighboring subdomains. In those cases, the working set per processor becomes smaller under CPS scaling. Let us now examine what determines the size of the important working set in the Barnes-Hut application and understand how it scales under different models.

To compute the force acting on it, a particle reads position data from other cells and particles in the tree, at lower levels close to it and at higher levels further away. Figure 9 shows the section of the tree that a particle references for force calculation. The *costzones* partitioning scheme we use ensures that both the partitions and the particle ordering within them are spatially contiguous (see Section 9.4.1), so that successive particles in a processor's partition reuse most of the previous particle's section of the tree. The amount

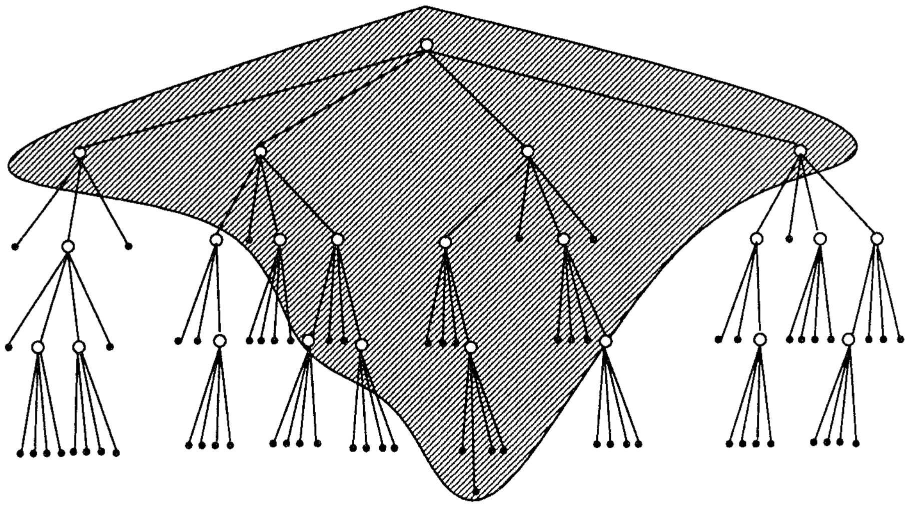


Fig. 9. Important working set for the Barnes-Hut force computation phase.

of displacement, mass, and moment data from particles/cells in such a section, which remains essentially the same in size and changes only very slowly with time in composition, is therefore a good measure of the important working set.

This amount of data, and hence the working-set size, is clearly independent of the number of processors. It is roughly proportional to $(1/\theta^2)\log n$ (see Section 7.2), with the constant of proportionality being a few kilobytes for our application. That is, it grows very slowly with n and much more quickly with θ. Under the equal-error scaling rule, if n is scaled by a factor of s, then θ should be scaled by $1/\sqrt[4]{s}$ when only centers of mass are used. Thus, the contribution of n to the scaling of working-set size is an *additive* term of $O(\log s)$, while that of θ is a *multiplicative* term of $O(\sqrt{s})$. (When quadrupole moments are used, the working-set size increases by about 60% over the same problem configuration with only centers of mass, and the contribution of scaling θ realistically is a multiplicative term of $O(\sqrt[4]{s})$ rather than $O(\sqrt{s})$.) The contribution of scaling the accuracy parameter is clearly much more significant than that of scaling only the data set size, which reinforces the importance of studying scaling under application considerations in order to reach the correct architectural conclusions.

Figure 10 shows simulation results for how the important working-set size scales with the number of processors under the equal-error scaling rule under all three scaling models, as well as under naive TC scaling. The problem sizes used are the same as those used to measure the scaling of the communication-to-computation ratio in Figure 7. As already mentioned, the working-set size remains constant under CPS scaling. Under naive TC scaling, n is increased quite quickly, but its contribution is only a very slow increase in working-set size. Under realistic TC scaling, which is the most appropriate

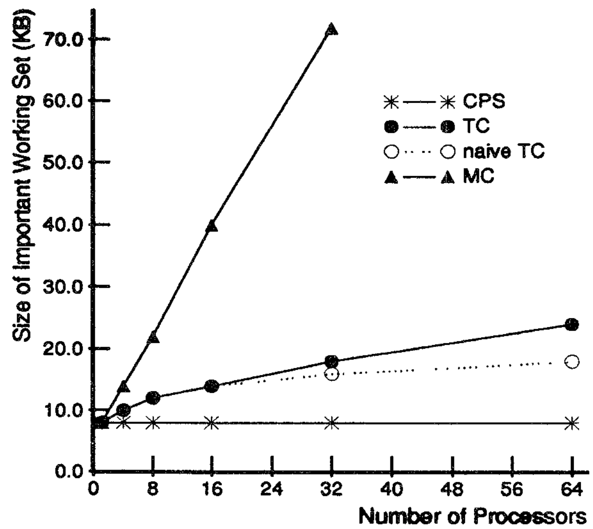


Fig. 10. Scaling of the important working set under all scaling models for the Barnes-Hut application.

method, n is increased less quickly. However, θ is also decreased. Since the impact of θ is much larger than that of n, the result is a faster increase in working-set size under realistic TC scaling than under naive TC scaling. Once again, this is particularly interesting since the important working-set size increases under realistic TC scaling despite the fact that the amount of data in each processor's partition actually decreases: the size of a processor's partition is clearly not a good indicator of its cache requirements in hierarchical N-body methods. Finally, under the impractical MC scaling, both n and θ are scaled very quickly, and the increase in working-set size is very rapid (though not so if only n is scaled in a memory-constrained way).

While the results in Figure 10 are for small simulated problems, as these scaling trends indicate, the actual values of the level-2 working sets for realistic problems today are quite small as well, and are likely to fit in caches on shared-address-space machines even as problems and machines are scaled in the foreseeable future. Let us look at some concrete examples, assuming that the level-2 working set scales as $(6/\theta^2)\log n$ kilobytes.

We begin with the typical problem discussed in Section 7.4.2 (64K particles with $\theta = 1.0$ and quadrupole moments) running on a 64-processor machine. The working-set size for this problem is about 32KB. Let us see what happens when we scale up to a 1024-processor machine and a million-processor machine.

Under MC scaling, a 1024-processor machine would run a problem with 1 million particles (near the large end of the numbers of particles that people can run on the most-powerful parallel machines today), and the million-processor machine would run a problem with a billion particles (inconceivable today). Under naive MC scaling, the working sets would grow to 40KB and

60KB, respectively. Under the inappropriate realistic MC scaling, the working set grows much faster, but would still be only 80KB with a million particles ($\theta = 0.71$) and under 300KB even with a billion particles ($\theta = 0.6$, octopole moments).[5]

Under the most appropriate realistic TC scaling, a 1024-processor machine would run only 256K particles ($\theta = 0.84$), and the important working-set size would be only 51KB. A million-processor machine would run about 32 million particles ($\theta = 0.6$, octopole moments), and the working-set size would still be under 250KB. As for associativity, experiments we have performed with up to 16K particles indicate that the cache size needed to hold the important working set increases by at most a factor of 2–3 in going from fully associative to even direct-mapped caches with the largest change being in going from two-way associativity to direct mapping.

Thus, although the important working set for this application grows slowly with problem size, it is inherently still well under 100KB for the largest problems people run today, and it is likely to stay small enough to fit in a cache even for problems whose solution is far beyond the realm of possibilities today.

The Fast Multipole Method exhibits results that are largely similar to those we have just seen for the Barnes-Hut method: in the importance of temporal locality as well as in scaling issues and characteristics. We address these issues for the FMM in the next section. Since we are interested in the adaptive FMM, and even a sequential three-dimensional adaptive code was not available at the time of this study, we study a two-dimensional FMM instead. The parallelization and the qualitative results extend to the three-dimensional case. The differences are that the constant factors are substantially larger in three dimensions (both in working sets and particularly in execution time, where the large constant factors have caused people to prefer using Barnes-Hut for most realistic problem sizes despite the superior asymptotic and error properties of the FMM), and the dependence of execution time on the force computation accuracy parameter ϵ is $\log^4\epsilon$ rather than $\log^2\epsilon$ (see below).

8. THE FAST MULTIPOLE METHOD

The Fast Multipole Method (FMM) comes in two flavors: uniform and adaptive. We use the adaptive FMM since it subsumes the uniform and is more useful in practice.

The FMM also uses a recursive decomposition of the computation domain into a tree structure and a similar strategy of approximating cells of the tree by equivalent single "particles" when they are far enough away. There are two key differences between it and the Barnes-Hut method. One is that while the Barnes-Hut method only computes particle-particle and particle-cell interactions, the FMM allows the direct computation of cell-cell interactions as

[5]As θ starts to shrink beyond about 0.6, people will tend to stop reducing θ and obtain higher force calculation accuracy by using higher-order moments instead.

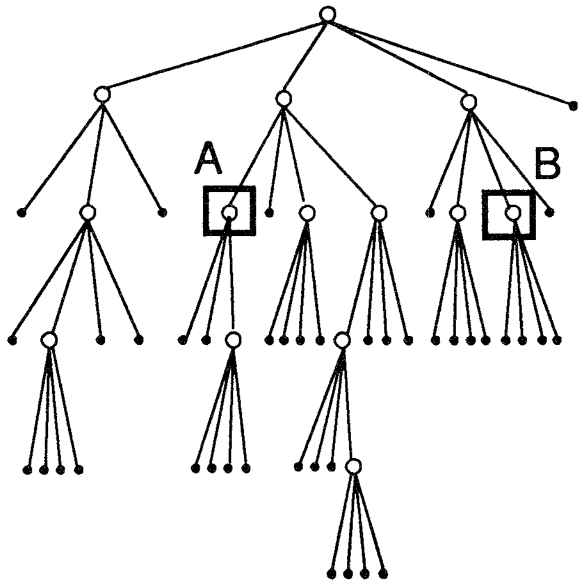


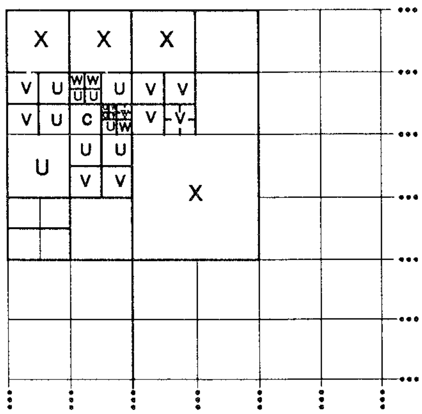
Fig. 11. The fast multipole method.

well. If an internal cell in the tree (cell A in Figure 11) is far enough away from another internal cell (cell B), then the interaction between these two cells is computed directly, and the effect of the interactions is propagated to the descendants of A and B. This reduces the computational complexity, since the individual particles or lower-level cells within cell A do not have to compute interactions separately with cell B, and vice versa. In fact, the use of cell-cell interactions makes the complexity of this algorithm $O(n)$, rather than $O(n \log n)$ as in Barnes-Hut, at least for uniform distributions [Greengard 1987; Singh 1993].[6]

The other key difference is that force calculation accuracy in the FMM is not determined by controlling which cells are considered far enough away with respect to a given cell. A cell, A, is simply considered far enough away (or "well separated") from another cell, B, if its separation from B is greater than the length of B. Cells that are far enough away are represented by higher-order series approximations (called multipole expansions) in the FMM than simply their centers of mass. In fact, the accuracy of the force computation algorithm is determined by the number of terms, m, retained in these series expansions.

The phases in a time-step of the FMM application are essentially the same as in Barnes-Hut: (1) building the tree, (2) computing multipole expansions of all cells about their geometric centers in an upward pass through the tree, (3)

[6]It is shown in Singh [1993] that the complexity of the FMM is not $O(n)$ for nonuniform distributions as originally believed.



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.

Fig. 12. Interaction lists for a cell in the adaptive FMM

computing forces (as described in the next paragraph), and (4) updating particle properties. Let us look at the force calculation phase in more detail.

For efficient force calculation, a tree cell b divides the rest of the computational domain into five lists of cells, each list containing cells that bear a certain spatial relationship to b. The four lists that b interacts with are shown in Figure 12; the fifth list (not maintained by the program) comprises cells that are not in the first four lists. 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 four lists. The hierarchical nature of the algorithm ensures that a cell b never directly computes interactions with cells that are well separated from b's parent (the cells in the fifth unlabeled list in Figure 12). Interactions with these distant cells are accounted for by b's ancestors at higher levels of the tree. Once the interactions for internal cells are computed, their effects are propagated down the tree until they reach the leaves, where they are evaluated at individual particles. Details of the different types of interactions, which are not very important for our purposes, can be found in Greengard [1987].

The efficiency of the FMM is improved by allowing cells at the lowest level of the tree to contain more than a single particle (we allow a maximum of 40 particles per leaf cell, as Greengard [1987] suggests and as works best in our implementations, unless otherwise mentioned). Thus, both the leaves and the internal nodes of the tree represent space cells in the FMM, and both maintain interaction lists as described above. The natural unit of parallelism in the FMM is a cell rather than a particle as in Barnes-Hut.

8.1 The Importance of Temporal Locality

The issues and results for temporal cache locality versus data distribution and locality in main memory are similar to those for Barnes-Hut. Figures 13(a) and 13(b) show the results for 60- and 500-cycle remote latencies, respectively. With the very large remote-to-local latency ratio (500-cycle

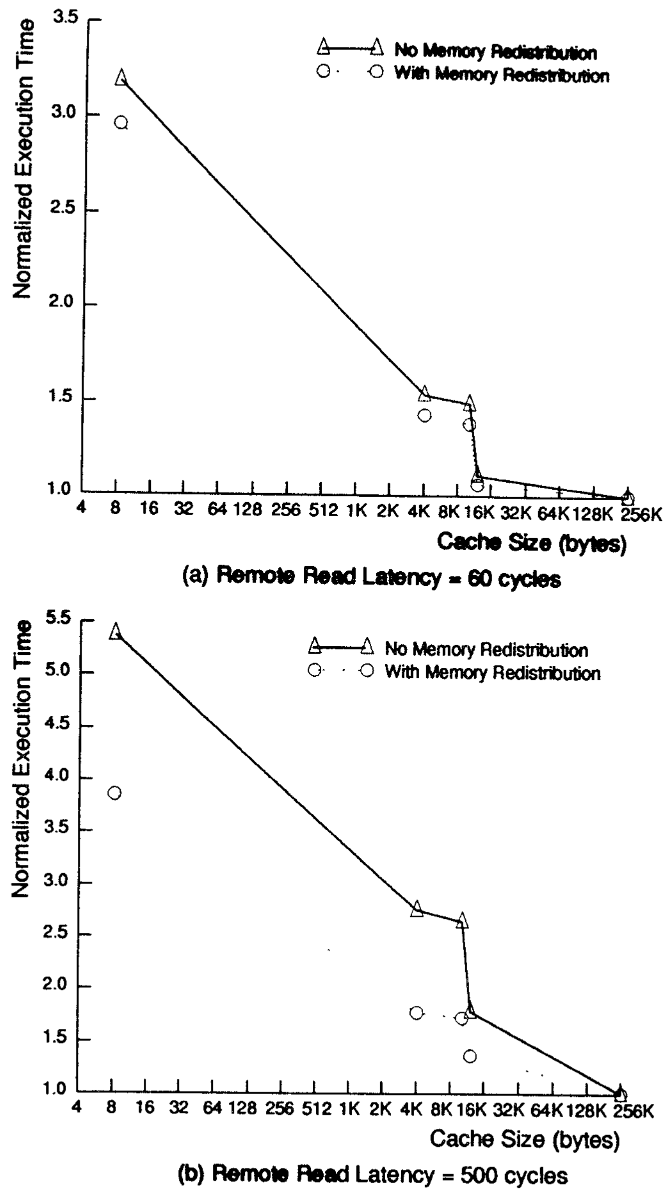


Fig. 13. Normalized execution time for the FMM application: $n = 4096$, $\epsilon = 10^{-5}$, $p = 16$.

remote latency), the impact of the idealized data distribution we use is quite significant in the FMM. This is because—compared to Barnes-Hut—relatively more of a processor's references to shared data are to data in its own partition rather than other partitions. However, we recall that our measurements do not consider the substantial difficulty and runtime overheads of redistribution. In reality, redistribution is not likely to help much even with these large latencies, and the temporal locality provided by caching communicated data remains the most effective way to obtain high performance. This property holds to a larger extent for the Barnes-Hut application than for the FMM.

8.2 Relative Scaling of Application Parameters

The FMM application also has the n, Δt, and ϵ parameters of the Barnes-Hut application. The only difference is in the force calculation accuracy parameter. In the FMM, force calculation accuracy is controlled by the number of terms, m, retained in the multipole expansions used to represent cells, rather than by a parameter like the Barnes-Hut θ that determines which cells are far enough away from a given point. Greengard [1987] shows that an error bound of ϵ in force calcualtion can be guaranteed by using a number of terms m equal to $\log \epsilon$. The equal-error scaling principle therefore yields the following parameter scaling rule for the FMM:

FMM Scaling Rule. If n is scaled by a factor of s, Δt must be scaled by a factor of $1/\sqrt[4]{s}$, and the number of expansion terms m must be increased by an additive term of $\log \sqrt{s}$.

8.3 Scaling of Memory Requirements and Computational Complexity

Memory. The number of bodies n is the major determinant of memory usage in the FMM as well. The contribution of increasing the number of terms, m, in the multipole expansions is negligible for practical purposes, particularly given how slowly m scales relative to n.

Time Complexity. The serial complexity scales as $O(m^2 * n/\Delta t)$. (As discussed in Singh [1993], at least one phase of the FMM has a complexity that is not quite $O(n)$ as claimed in Greengard [1987]. However, the $O(n)$ components dominate for the problem sizes we have run, and we assume an $O(n)$ complexity in this article.) Under the equal-error scaling rule, then, an increase in the input data set size (n) by a factor of s leads to a storage requirement increase of essentially the same factor s, but a serial complexity increase of a factor of

$$s * \sqrt[4]{s} * (\log \sqrt{s})^2.$$

8.4 Impact on Memory-Constrained and Time-Constrained Scaling

Execution Time Under Memory-Constrained Scaling. An increase in the number of processors, and hence memory, by a factor of k allows us to simulate roughly k times as many particles. Under naive scalability analysis, the $O(n)$ complexity suggests that the parallel execution time would not change at all. Under realistic scaling, however, the parallel execution time increases by a factor of $\sqrt[4]{k} * (\log \sqrt{k})^2$, assuming perfect speedup. If the base problem took 1 day on a single processor, the scaled problem on 1024 processors would take 187 days rather than the 1 day predicted by naive scaling analysis. This is slower than the increase in the Barnes-Hut case, but is still unacceptable.

Data Set Size Under Time-Constrained Scaling. Under naive time-constrained scaling, the $O(n)$ complexity of the algorithm would suggest that n would scale linearly with the number of processors, i.e., just as fast as under memory-constrained scaling. Under the equal-error scaling rule, however, the relationship between the factor s by which n grows and the factor k by which the number of processors (p) grows is given by:

$$k = s^{5/4}\left(1 + \frac{\log^2 s}{\log^2 e_0}\right),$$

where e_0 is the force calculation accuracy used on the smaller machine. Thus, if a problem with 16K particles and a force calculation accuracy of 10^{-8} takes one day to complete on a uniprocessor, then a machine with 1024 processors would run a problem with only about 3.8 million particles in the same time, rather than the 16 million particles predicted by scaling only data set size.

8.5 Scaling of Inherent Communication

Communication in the FMM can be quite easily modeled analytically for a uniform distribution (see Appendix A). The dominant terms in the computation and communication expressions for far-field interactions (those which use multipole expansions) using the tree can be shown to scale as $O(m^2 * n_g)$ and $O(m * \sqrt{n_g})$, respectively, where m is the number of terms used in the expansions.[7] For the near-field (direct) interactions, the dominant terms are $O(c^2 n_g)$ and $O(c\sqrt{n_g})$, respectively, where c is the number of particles in a leaf cell. The communication-to-computation ratio is therefore proportional to $1/(\alpha * n_g)$ in both cases, where α represents either m or c.

Under CPS scaling, α does not change while n_g decreases linearly with the number of processors p. The communication-to-computation ratio therefore grows roughly linearly with p. It takes the unrealistic MC scaling (naive, which is unrealistic, or realistic, which is impractical) to keep the communication-to-computation ratio roughly constant in the FMM as well.

Under realistic TC scaling, n_g decreases a little faster than as $1/k^{1/5}$ (since n increases a little slower than $k^{4/5}$; see Section 8.4). c remains constant, and m increases only very slowly (as $\log k^{4/5}$; see Section 8.2). The communication-to-computation ratio therefore increases slowly in the FMM application as well. However, even in this application, the absolute values of the ratio are very small for the problem and machine sizes that people use today or are likely to use in the foreseeable future. Hierarchical N-body methods are therefore likely to benefit substantially from large-scale parallelism.

[7]There is a $\log p$ term in the communication expression, as shown in Appendix A, but it is relatively unimportant until the number of processors becomes very large.

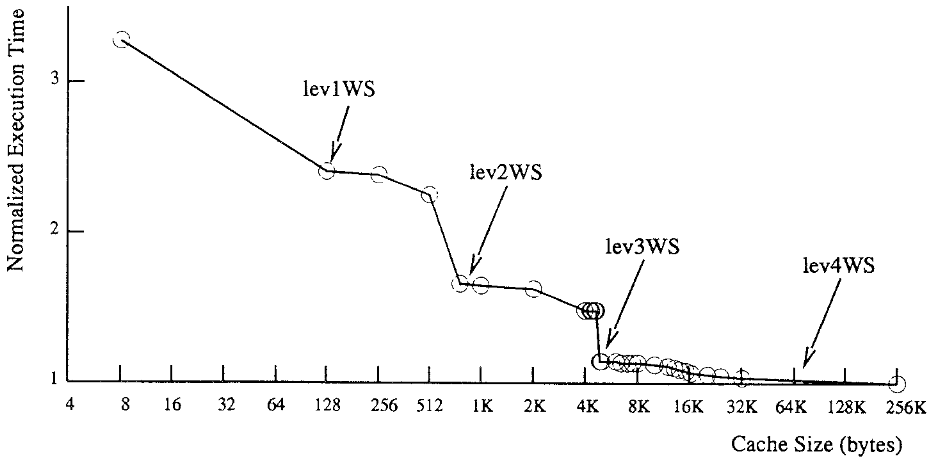


Fig. 14. Working sets for the FMM application: $n = 4096$, $\epsilon = 10^{-5}$, $p = 4$.

8.6 Scaling of Working-Set Size

The FMM application has four well-defined levels in its working-set hierarchy, as shown in Figure 14. The lev1WS and lev2WS represent the amount of temporary storage needed to compute single interactions of different types. The lev1WS is for interactions that do not involve multipole expansions, such as direct particle-particle interactions in the U lists, while the lev2WS is for interactions that involve the manipulation of expansions. Both these working sets are very small. The lev1WS is about 0.125KB, independent of n, p, or m. It brings the miss rate down to about 60%. The lev2WS is proportional to m (the number of terms in the expansions). More precisely, its size is about $0.625m$ kilobytes, independent of n or p. It brings the miss rate down from about 60% to about 20%, still not low enough for effective performance since most of the remaining misses are to nonlocal data in this application as well.

The most important working set is the lev3WS. Like the important working set in the Barnes-Hut application, it too is very sharply defined, taking the normalized execution time down from about 150% to about 110%. For the small problem shown in Figure 14, the size of the lev3WS set is about 4.75KB. We shall discuss its scaling shortly.

Once the lev3WS is crossed, performance improves much more slowly with cache size until the lev4WS is reached. The lev4WS set has the same characteristics as the lev3WS in the Barnes-Hut case, and we do not consider it further.

Let us examine the composition and scaling of the important level-3 working set. The dominant part of the application is the computation of list interactions, in which a processor computes the U, V, and W/X list interactions, as applicable, of all cells in its partition (see Figure 12). Because partitions are spatially contiguous, the lists for successive cells are likely to overlap substantially, providing much reuse of data. For simplicity of explanation, and because most of the time is spent in processing V list interactions, let us understand the scaling of the working set assuming that it contains only V list interaction data. That is, the working-set size is roughly proportional to the product of two factors: the number of cells in the average V list that are reused and the amount of data needed from each cell.

Both these factors, and hence the working-set size, are clearly independent of the number of processors p (as in Barnes-Hut). Interestingly enough, and unlike in the Barnes-Hut case, the working-set size is essentially independent of the number of particles, n, as well, and is therefore constant under all naive scaling models. Larger n does mean more cells in the system, but it does not affect either of the factors that contribute to the working-set size. The only parameter that the working-set size depends on is the number of terms m used in the multipole expansions, since the amount of data needed from each cell in a V list interaction is proportional to m. For the galactic distributions we have used, we find that the size of the level-3 working set can be expressed as $19 * 16 * m$ bytes. The 19 factor comes from the 18 cells (out of a maximum of 26) in a V list that are reused on average, plus the cell whose expansion is being written. The 16 factor comes from the 16 bytes of data that constitute each term in the expansion.

Under the equal-error scaling principle, if n is scaled by a factor of s, then m should be scaled by a factor of $\log \sqrt{s}$. Thus, while the important working-set size remains constant under the naive scaling models, it grows, albeit extremely slowly, under realistic TC or MC scaling. In this sense, the FMM highlights the importance of scaling parameters other than the data set size more dramatically than does Barnes-Hut, even though the growth of the working set is much slower in the FMM: whereas in Barnes-Hut the accuracy parameter caused the working-set size to scale more quickly than did the data set size, in the FMM the accuracy parameter (m) is the only application parameter that affects the size of the working-set, so that the difference between naive and realistic scaling is qualitative rather than only quantitative.

We now turn our attention to the last of the architectural implications discussed in this article. We examine whether certain communication characteristics in applications that use hierarchical N-body methods lead to significantly larger programming complexity or runtime overheads when implemented under one or other of the dominant communication abstractions in parallel computers.

9. IMPLICATIONS FOR COMMUNICATION ABSTRACTIONS

Parallel applications have two broad classes of data: logically *shared* data, whose values need to be accessed by more than one processor and therefore have to be communicated among processors, and *private* data, whose values are needed by only a single processor. Programming paradigms for parallel computers differ primarily in the abstractions they provide for managing the communication of logically shared data among processors. These differences have implications for ease of programming, for the structuring of communication, for performance, and for scalability. The two dominant abstractions today are implicit communication through a *shared address space* and explicit communication through *message passing* among private address spaces. The communications architecture of a multiprocessor exports a hardware communication abstraction or set of primitives to software, and a programming language built on top of these primitives exports a communication abstraction to the programmer. The abstractions at the two levels may be different, but are likely to be most efficient when they match. In this section, we argue that a shared address space, particularly with automatic and coherent caching of shared data, has substantial advantages in programming complexity over explicit message passing for applications with irregular, dynamically changing behavior, and that the complexity of managing communication in a message-passing abstraction is likely to translate into runtime disadvantages as well. A question that follows naturally is whether a shared address space and automatic caching should be supported in hardware, and we comment on this as well.

We begin by describing four important aspects of communication management and how the dominant programming paradigms differ in these aspects in general. Then, we examine how our hierarchical N-body applications in particular interact with the programming paradigms in the four aspects.

9.1 Communication Management in the Shared-Address-Space and Message-Passing Paradigms

The 4 important aspects of communication management are (1) naming and protection, (2) exploiting temporal locality on logically shared data, which includes both managing data replication and renaming as well as maintaining the coherence of replicated data, (3) the granularity and overhead of communication, and (4) synchronization.

Naming. When a process needs a datum that is logically shared, how does it reference that datum or find it in the machine? In the shared-address-space (henceforth abbreviated SAS) paradigm, the burden of finding data is placed on the hardware or system software [Li and Hudak 1989], so that any application process can directly reference any variable that is declared in the shared address space (we assume hardware support for this address translation or naming and present some recently published results for software-versus-hardware address translation in Section 9.10). In the message-passing paradigm, on the other hand, the burden of finding data is on the application programmer. A processor can directly reference only those data that are in its local address space (local memory).[8] To reference a datum, the application program must therefore first know or determine which processor's address space it resides in and send a message to that processor requesting the datum.

[8] We use the terms "local address space" and "local memory" interchangeably for message-passing machines in this article. That is, we ignore the fact that secondary (disk) storage may be used to provide a per-processor virtual address space that is larger than the local memory. We also ignore disk storage in shared-address-space machines.

Related to naming is the problem of protection, i.e., ensuring that a process does not access physical data to which it does not have the appropriate access rights. In hardware cache-coherent machines, protection is provided by standard hardware virtual memory mechanisms just as on uniprocessors, though on many machines software may have to be invoked when virtual-to-physical mappings change. In the message-passing abstraction, the lack of hardware address translation for nonlocal data makes protection the responsibility of software for all but operating environments (such as gang scheduling).

Temporal Locality. As we have seen, in a machine with a physically distributed memory system it is often advantageous for a processor to replicate data locally that are communicated from other processors and exploit temporal locality by reusing the local copies rather than recommunicating every time the data are needed.[9] There are two aspects to exploiting temporal locality on communicated data: replicating the data and maintaining the coherence of the replicated data. We discuss data replication first.

Replication. Four issues distinguish how data are replicated on different systems: (1) who is responsible for doing the replication, i.e., for making local copies of the data, (2) where in the memory hierarchy is the replication done, (3) at what granularity are data replicated, and (4) how is the replacement of replicated data managed?

In the message-passing paradigm, the only way to replicate communicated data is for the user to copy the data into a processor's private address space explicitly in the application program. Data are always replicated in main memory first: a processor's cache holds only those data that are allocated in the processor's local address space (typically implemented by its local memory). The granularity of replication is variable and depends entirely on the user. We shall discuss replacement a little later.

The SAS paradigm allows machines to provide system support for replicating communicated data automatically, without user intervention. The support may be provided in hardware or software, depending on the particular architecture and the level of the memory hierarchy at which data are being replicated. For example, some SAS architectures (such as the Stanford DASH [Lenoski et al. 1990] and the MIT Alewife [Agarwal et al. 1991]) automatically replicate communicated data in the processor caches under hardware control, even without replication in main memory. Replication is managed at the fixed, usually small, granularity of a cache line in these cases. Other systems provide automatic replication in main memory under system software control. For example, IVY [Li and Hudak 1989] and its descendants provide replication at the fixed relatively large granularity of physical memory pages. Page replication has two potential disadvantages: (1) the software management and large granularity of pages make page replication expensive

[9]Replication may be viewed as sometimes exploiting spatial locality, when the granularity of communication is larger than a single data word and no individual data word in that unit of communication. However, this spatial locality at the granularity of individual data words can be viewed as temporal locality or reuse at the larger granularity at which data are communicated.

and (2) the large granularity can lead to unnecessary traffic when the actual data sharing in the application is fine grained.[10] We assume SAS machines with hardware caching of communicated data in our discussion.

If the processor caches on these machines are not large enough to exploit temporal locality effectively, data may be replicated at subsequent levels of the memory hierarchy, with the performance benefits decreasing as access latencies to successive levels increase. These next levels may include software-managed main memory—as in DASH- or Alewife-like machines—in which case replication is done at the level of pages with the attendant disadvantages or implicitly by the program, or they may themselves be hardware-managed caches, as in cache-only memory architectures [Hagarsten et al. 1990]. As we have seen in Section 7.5, the working sets of our hierarchical N-body applications fit comfortably in even relatively small processor caches by today's standards. For these applications, therefore, we do not need any system support for replication beyond the processor caches to argue the advantages of SAS machines that cache communicated data.

Finally, let us look at replacement, which is necessitated by the finite capacity of caches and memories. How replacement is managed has implications for the amount of communicated data that needs to be maintained in the relevant level of the memory hierarchy. Hardware caches manage replacement automatically and dynamically with every reference, so that the cache needs only be as large as the active working set of the application at any time (see Section 7.5). While replacement can be managed just as dynamically by the user in the message-passing paradigm as well—for example, by maintaining a cache data structure in local memory and managing it like a hardware cache for communicated data—this is both difficult and expensive to do in the application program. In typical usage of message-passing machines, replacements are managed less dynamically. Communicated data are allowed to accumulate in local memory and are flushed out explicitly at certain points in the program, typically when it can be determined that they are not needed at least for some time.

Coherence. Replication implies maintaining multiple copies of a logically shared datum in different caches or local memories. If a modification is made to one of the copies, correctness demands that the owners of other copies that

[10] A proposed alternative to page replication is to handle replication in main memory at the granularity of logically defined, variable-sized program objects. This is not without problems either. If the granularity of the objects is small (as in our N-body applications, where an object is a particle or cell), the overhead of maintaining translation information about objects in software can be excessive. Also, it is rarely the case that an entire object needs to be replicated at a given time; usually, it is only a part of an object. For example, in the force computation phase in the Barnes-Hut application, only the position and mass information need to be replicated, not the velocity or acceleration. Even object replication can thus lead to unnecessary traffic and replication. And finally, although in some applications the natural objects from the data abstraction and object-oriented programming points of view are the same as the objects that one wants to communicate or replicate for locality, this is very often not the case, and the latter "objects" have to be defined differently.

are still interested in the datum be notified before they use that datum again. Preserving this correctness is called maintaining the coherence of the memory system.

In the message-passing abstraction, the burden of managing coherence is also placed on the application programmer, just like naming and replication. A processor's modifications to data do not propagate beyond the local node unless the user program explicitly sends the appropriate messages to other nodes.

In the SAS abstraction, on the other hand, the burden of coherence may be placed on hardware, system software, the application programmer, or any combination of these. The granularity at which coherence is maintained is usually the same as that at which data are communicated and replicated, although this is not necessary. For example, the DASH multiprocessor keeps cached data coherent in hardware (transparently to the user) at the fixed, fine granularity of a cache line. And the IVY system supports its main memory replication by providing coherence at the coarser page level in system software. Such coarse-grained replication/coherence systems, however, can incur a lot of unnecessary coherence traffic due to the so-called *false-sharing* problem. For example, detailed studies of coherence traffic in some parallel applications have found the ideal granularity for replication/coherence in SAS machines to be much smaller than a page (under 128 bytes) for the applications studied [Eggers and Katz 1989; Torrellas et al. 1990; Weber and Gupta 1989].

We examine message-passing machines on one hand and SAS machines with hardware-supported cache coherence on the other (we assume hardware coherence, although it could be an efficient combination of hardware and system software). We call the latter machines SAS-CC (for "cache-coherent") and assume that the mechanisms they use to maintain coherence are scalable [Gupta et al. 1990; James 1989; Scott 1991; Simoni and Horowitz 1991].

Overhead/Granularity of Communication. Finally, the structure and granularity of communication are usually very different in SAS-CC and message-passing machines. The SAS paradigm promotes a style of programming that is a natural extension of uniprocessor programming: data items are referenced (read or written) as and when needed, thus generating *receiver-initiated*, *fine-grained* implicit communication through loads and stores. Since SAS-CC machines manage translation, protection, and communication in hardware and at the fixed granularity of relatively small cache lines, they support this fine-grained communication efficiently. The fixed granularity of communication makes efficient buffer management easy in hardware.

In the message-passing paradigm, the programmer is responsible for composing the messages that processors send to one another. This has certain advantages in the flexibility to control the structure and granularity of communication (e.g., sender- or receiver-initiated, coarse- or fine-grained). The flexibility, however, comes at the cost of substantial overhead in sending and receiving messages: messages have to be packaged and unpacked in software, and the need to manage incoming messages (e.g., ensure the desired protection and data alignment, buffer messages until the destination process is ready for them, etc.) typically involves the copying of messages into and out of buffers and the expensive intervention of an operating system. Thus, although there is no natural predilection for coarse-grained messages in the message-passing paradigm itself, there is a tremendous push in this direction from the performance viewpoint—to amortize the fixed overhead of a message over the many words of data being sent—at least on current message-passing systems. Moving a large chunk of data from one processor to another is usually more efficient with coarse-grained bulk transfer messages than through individual loads and stores. Note that there is nothing in the existence of a shared address space that precludes the use of coarser-grained communication. In fact, bulk transfer support in SAS machines can utilize the shared address space for efficiency of end-to-end transfer. However, most existing SAS machines do not yet provide efficient support for bulk transfer, but rely on fine-grained load/store communication.)

New message-passing systems are being proposed and implemented that try to "get the operating system out of the way" for most messages and optimize message-sending efficiency for the common case. Fine-grained messages will continue to be substantially more expensive in these systems than in SAS-CC machines. However, the need for coalescing messages together will decrease as the overhead of sending and receiving messages decreases.

Synchronization. A potential advantage of the message-passing abstraction is that, since data communication is explicit, synchronization is implicit in the data communication itself, i.e., a send-receive pair implements a data transfer and a pairwise synchronization event. In a load/store shared address space, synchronization is conceptually separate from the implicit data communication and must be performed separately and implicitly. Message passing can thus have both conceptual and performance advantages in this regard, since a single message can implement both data transfer and synchronization. However, this must be traded off with the complexity of naming, which is involved even in synchronization in message passing (at least for naming the processes with which to synchronize).

9.1.1 *Summary: General Advantages and Disadvantages of the Two Paradigms.* Table I summarizes the differences between machines that support the two paradigms. The primary advantage of explicit message passing is the ease and efficiency of building scalable machines, since processing nodes require minimal hardware/software support for communication management. On the other hand, several research projects (for example, Agarwal et al. [1991], Hagarsten et al. [1990], Lenoski et al. [1990]) are demonstrating that the overheads of providing cache coherence are relatively small, and that cost-effective, scalable SAS-CC machines can indeed be built. In fact, the cost of the extra main memory which is needed on message-passing machines for explicit replication of operating system and application code and data often dominates the hardware cost of cache coherence, as we shall see.

Table I. Comparison of Communication Abstractions

Aspect	*Cache-Coherent Shared Address Space*	*Message Passing*
Naming	Hardware address translation	No hardware address translation
Communication Overhead	Efficient support for fixed-size, fine-grain communication	Inefficient for fine-grain efficient for coarse-grain
Replication and Coherence	Automatic, hardware-supported in caches	No hardware support
Synchronization	Separate explicit synchronization events	Implicit in explicit communication
Advantages	* Ease of programming * Performance?	* Hardware design and cost * Bulk data transfer

The primary disadvantage of the message-passing paradigm is programming complexity. In many scientific applications, the interprocessor communication patterns are naturally well structured and predictable, so that the complexity of managing communication in the user program is not very severe. (For example, see dense linear algebra computations and Singh and Hennessy [1992]). For the hierarchical N-body applications considered here, however, we will show that communication management adds substantial conceptual, programming, and runtime overheads in message-passing implementations. In fact, these overheads are observable in a range of applications (computer graphics, for example) which share certain characteristics (described in Section 9.3) of hierarchical N-body methods.

9.2 Axes of Comparison for an Application

Other than system cost, there are two axes along which we can compare the SAS and message-passing paradigms for a particular application: programming complexity and performance. By programming complexity we mean the conceptual design and programming effort required to obtain *effective* parallel performance, not just to get a parallel program to run correctly (a shared address space is clearly advantageous for the latter purpose). Comparisons along this axis are difficult to quantify but are instructive. Direct comparisons of performance, while easier to quantify, are often less useful since they are very specific to the platforms on which performance is evaluated. We therefore focus on a comparison of programming complexity, and point out cases in which the complexity translates directly to runtime overhead, such as extra work done by the program, extra memory requirements, or extra communication. We use the Barnes-Hut application first to make most of our points, since it is simpler than the FMM and since a good message-passing implementation exists for it. Then, we examine some additional issues raised by the FMM.

9.3 Problems for Message-Passing Implementations

The complexity of managing communication in message-passing implementations of hierarchical N-body applications arises from two sources:

—the *nonuniform* and *dynamic* nature of the domain (particle distribution), which implies that the partitions assigned to processors change with time to maintain load balancing and data locality, and

—the need for *long-range* communication which is *irregular* as a result of the nonuniformity.

This leads to the following difficulties for the various aspects of communication management in the message-passing paradigm.

Naming. The structure of the tree changes slowly but unpredictably across time-steps, as do the assignments of particles/cells to processors. Hence, the address spaces (processes) from which a processor has to get the nonlocal data it needs change unpredictably across time-steps. Particularly given the irregular access patterns to the tree, a naming problem arises: *how does a processor know where to find the data it needs*, without resorting to the highly undesirable recourse of having every processor build and store a local copy of the entire tree in every time-step? As we shall see, the application-level orchestration required to address this naming problem—which hardware takes care of in SAS-CC machines—adds substantial algorithmic and programming complexity in message-passing implementations and is the largest source of execution time overhead (much larger than communication itself).

Replication. Replication can cause problems too, particularly owing to how replacement is managed. The typical message-passing strategy of allowing communicated data to accumulate in local memory and flushing them out at certain points in the program does not work well here like it does in regular, predictable programs. The reuse patterns even within a computational phase are irregular and unpredictable—for example, reusing the tree data structure in the force computation phase. Thus, the only convenient points at which to flush communicated data in the application program are the boundaries between computational phases. Since the amount of nonlocal data read from the tree during the force computation phase is large, the memory overhead due to data replication can be large when replacement is managed in this typical message-passing style (common cases require more than five times as much memory per process for replication as for holding the inherent data set; see Section 9.6.2), even though the active working set needed by a processor is small, as we have seen for the SAS implementation. Replication requirements can be reduced in some naming schemes by emulating a hardware cache in software for nonlocal data, thus needing to replicate only the active working set, but this incurs the overhead of runtime software checks of the emulated cache and determining whether the data are local or remote.

Coherence. Two levels of coherence need to be provided in these applications. First, as in many scientific programs [Singh et al. 1992b], some phases of computation have the property that the shared data read in those phases are not modified in them. For example, the portions of particle/cell data that are read in the force computation phase of the Barnes-Hut application are not modified in that phase but only later in the update phase. Coherence does not need to be provided within these phases, but only at the boundaries between them.

Other phases, however, require finer-grained coherence within them as well. In building a globally shared tree as in our SAS implementation, for example, different processes read and modify the same parts of the logically shared tree in an unstructured way (see Singh et al. [1992a]). The patterns of this read/write sharing are unpredictable, so that maintaining the required coherence in the application program is difficult. Nonetheless, there is significant reuse of data even in this phase, so caching and automatic coherence are helpful.

The tree-building and center-of-mass or expansion computing phases are also the ones that need complex synchronization (mutual exclusion and point-to-point event synchronization on cells; global barriers between phases are convenient elsewhere). However, given the problem of naming processes that own tree cells, it is not clear that message passing has any advantage here over using shared flags and locks with no naming problem.

Communication Granularity. The natural data-referencing patterns in these applications lead to fine-grained communication. Successive references to logically shared data follow the links in the tree—and depend on the opening criterion or interaction lists—so that they do not access predictable, contiguous data in the application's data structures. Significant programmer effort is required to combine messages and hence increase the granularity of communication as needed for performance in message-passing implementations, as we shall see. It would be very difficult for a compiler to do this automatically, for example. Also, since data transfer itself is not a severe performance bottleneck (see Section 7.4), bulk data transfer is not too likely to help performance much anyway.

In the rest of this section, we examine in more detail how these communication management problems have been or can be addressed in actual message-passing implementations, both to obtain a deeper understanding of the complexities and to quantify some of their effects. Since effective techniques to solve the above communication management problems depend on how the computation and data are partitioned among processors, let us first look at two partitioning techniques that yield effective performance.

9.4 Partitioning Methods

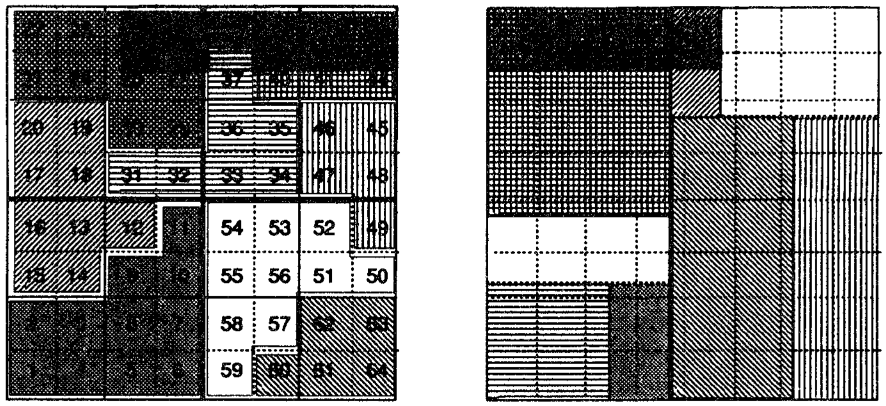
The goal in partitioning is to balance the workload across processors, and to provide locality of data reference by ensuring that the particles/cells assigned to a processor are close together in space. We describe two partitioning methods, initially in the context of the Barnes-Hut application: one (called *costzones*) is a simple technique that we have proposed for shared-address-space implementations, and the second (called *Orthogonal Recursive Bisec-*

tion or *ORB*) is a more complex technique used by Salmon [1990] in a message-passing Barnes-Hut implementation (which we shall describe), and which we have also implemented for shared-address-space machines. Details of the techniques can be found in Singh et al. [1992a].

9.4.1 *Costzones.* The Barnes-Hut algorithm already has a representation of the spatial distribution encoded in its tree data structure. In cost zones partitioning, 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. The cost of (or work associated with) every body is profiled in the previous time-step and stored with the body. A cell stores the sum of the work associated with all the bodies it contains. The total work in the system is divided among processors so that every processor has a contiguous, equal range or zone of work (for example, a total work of 1000 units would be split among 10 processes so that zone 1-100 units is assigned to the first process, zone 101-200 to the second, and so on). Which cost zone a body in the tree belongs to can be conceptually determined by the total cost of an in-order traversal of the tree up to that body. Processes partially traverse the tree in parallel, picking up the bodies (or usually entire large internal cells) that belong in their cost zone. The partitioning algorithm requires only a few lines of code, has negligible runtime overhead, and yields very good load balance [Singh et al. 1992a]. The partitions produced by the *costzones* scheme are not shaped regularly in space for nonuniform particle distributions, however (see Figure 15), which has implications for message-passing implementations as we shall see.

9.4.2 *Orthogonal Recursive Bisection* (*ORB*). The ORB technique [Fox 1988] obtains more-regular partitions (at least in Barnes-Hut; see Section 9.7) by directly partitioning space rather than the tree. The tree is not used in the partitioning process at all. The idea here is to divide space recursively into two subspaces with equal cost, until there is one subspace per process (see Figure 15). This introduces a new data structure: a separate binary ORB tree whose nodes represent the recursively subdivided subspaces and the processors associated with them, and whose leaves are the final spatial partitions. A parallel median finder is used to determine where to split the current subspace in the direction chosen for the split. The result is a set of regularly shaped partitions that are each contiguous in space. A complete description of the application of ORB to this problem can be found in Salmon [1990] and Singh et al. [1992a]. Besides introducing new data structures, ORB incurs substantially more runtime overhead and is more complex to implement and debug than the *costzones* tree-partitioning scheme we described above [Singh et al. 1992a].

In our SAS-CC implementations, we find the *costzones* scheme to perform better than ORB, particularly as the number of processors increases. The performance of the force calculation phase is almost equally good in the two schemes (*costzones* being a little better since it has slightly better load balancing and since it automatically orders the particles within a partition to



Costzones ORB

Fig. 15. Tree partitioning and spatial locality.

maximize temporal locality), but the cost of partitioning itself grows much more quickly with the number of processors in ORB [Singh et al. 1992a].

The only prior parallel implementation of a nonuniform hierarchical N-body method is a message-passing, galactic Barnes-Hut application by Salmon [1990]. His implementation uses ORB partitioning and yielded good parallel performance on a 512-processor NCUBE system. We shall call his approach the *locally essential trees* approach. In the rest of this section, we first briefly describe this approach, discuss how it handles the aspects of communication management described above, and compare its programming complexity with that of our shared-address-space Barnes-Hut implementations. Then, we examine the runtime overheads resulting from various sources in the locally essential trees approach. We argue that the programming and performance overheads are likely to be even greater for the FMM application than for Barnes-Hut. We comment briefly on possible improvements to Salmon's implementation of the approach, and discuss two other approaches with less well structured communication that might be used in the message-passing paradigm. These approaches are more flexible than locally essential trees and can reduce the replication requirements, but neither of them significantly alleviates the complexity of communication management or the performance overheads. Let us begin with the locally essential trees approach.

9.5 Salmon's Message-Passing Approach: Locally Essential Trees

9.5.1 *Concept.* This approach solves the naming problem by having the sender of data initiate and manage communication rather than the receiver. A new phase is introduced in every time-step, between the partitioning and force computation phases. In this phase, sender-managed communication brings to every processor all the particle/cell data that that processor will need to compute forces on the particles assigned to it. The part of the Barnes-Hut tree that these data comprise for a process is called the process'

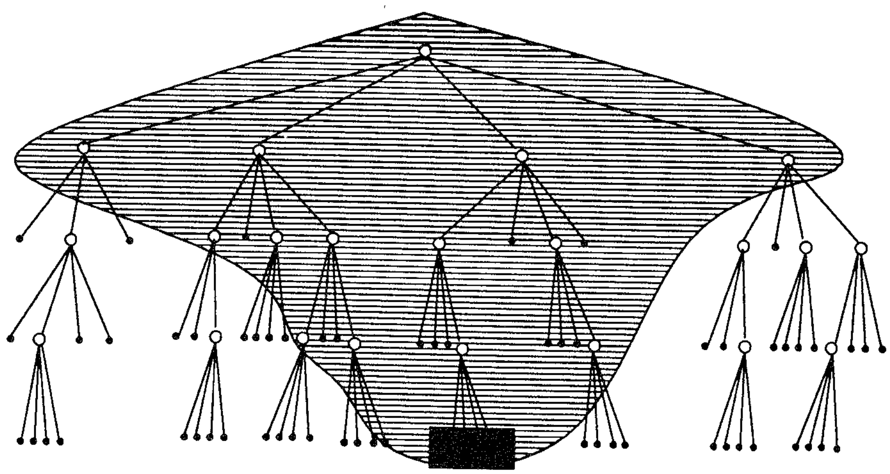


Fig. 16. A locally essential tree.

locally essential tree (see Figure 16, in which the shaded irregular part of the tree represents the locally essential tree of the process whose partition of particles is in the shaded rectangle). Clearly, the locally essential trees of different processes overlap substantially.

Once a process has its locally essential tree in its local memory, it can perform its force computation phase without any further interprocessor communication. Because the locally essential trees approach relies on a partitioning scheme (ORB) that provides physical locality, a process' locally essential tree is substantially smaller than the whole Barnes-Hut tree (see Figure 16). In fact, a complete Barnes-Hut tree that includes all particles is never constructed in this approach, but can be viewed as the union of all processes' locally essential trees. In contrast, our SAS approaches construct a single, complete shared tree, from which all processes directly reference the data they need during the force computation phase, implicitly communicating with other processes as necessary.

9.5.2 *Implementation.* To build locally essential trees, Salmon relies on two properties specific to the ORB partitioning scheme: the geometrically regular structure of the partitions it yields and its recursive, hierarchical nature. (As we shall see, the locally essential trees approach cannot easily be adapted for use with *costzones* partitioning, which is the most successful partitioning technique on a shared-address-space machine.)

The insight exploited is the following. Although a processor that needs a particle/cell C does not know which process' partition C is in, the process P that owns C can quite easily determine if C's children might be needed by some particle in another process' regularly shaped spatial subdomain S. Process P can do this by assuming that there is a particle at the point in S that is nearest to cell C (since the actual distribution of particles in S is yet unknown) and evaluating the Barnes-Hut opening criterion for that point

```
BuildTree (bodylist)
{
    Initialize locally essential tree to empty root cell, which represents entire domain
    Load bodies in bodylist into locally essential tree
    for (each bisection in ORB partitioning) {
        Traverse locally essential tree and queue data that may be necessary
            to some processor in the domain on other side of bisector
        Traverse locally essential tree and delete data that were temporarily
            received as a partner processor in a previous iteration but are
            no longer needed on this processor's side of current bisector
        Exchange queued data with "partner" processor
            on other side of bisector
        Merge received data into locally essential tree
    }
```

Fig. 17. Algorithm to build locally essential trees.

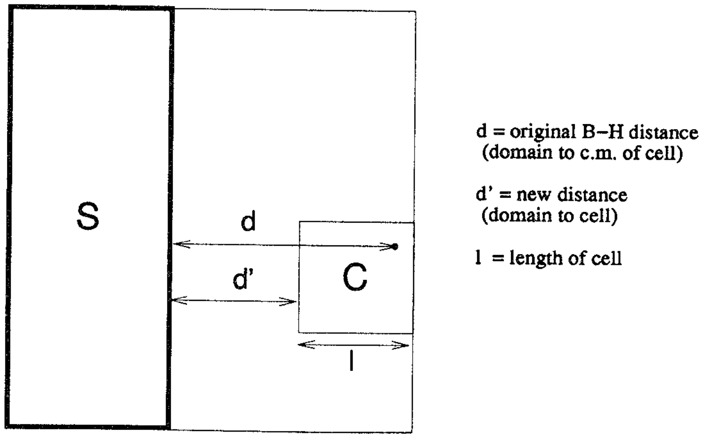


Fig. 18. Opening criterion for a subdomain and a cell in Salmon's method.

and the cell C. The algorithm to build locally essential trees is outlined in Figure 17. Details can be found in Salmon [1990].

Locally essential trees built in this way are conservatively large. The reason is that while the Barnes-Hut opening criterion is evaluated between a particle and the center of mass of a cell (see Section 3.1) the actual center of mass of cell C is not known at the time of the above opening-criterion evaluation: although C is in process P's locally essential tree, other processes might have particles in their partitions that are within cell C but that have not yet made their way into process P's locally essential tree. So, P must conservatively assume that C's center of mass is at the point within C that is closest to subdomain S, which can result in the opening criterion being satisfied and C's children being communicated when they are not really needed by particles in S for force computation (see Figure 18). This is a problem not only for the original Barnes-Hut criterion, but for any of the very useful opening criteria that require knowledge of the distribution of particles within a cell (personal communication, J. K. Salmon, 1991).

9.5.3 *Summary of Communication Management.* Communication in Salmon's locally essential trees method is initiated and managed by the sender of data rather than by the receiver. The naming problem is addressed by having the sender determine potential consumers of a datum and send it to them, rather than by having a processor that needs a datum determine where to find it. Replication is managed by explicitly copying logically shared data into different processors' locally essential trees (i.e., in local main memory). Replacement of communicated data is managed in typical message-passing style, so that a processor's local memory must be large enough to accommodate all the processor's local data as well as all the nonlocal data in its locally essential tree.

Coherence at phase boundaries is managed explicitly. For example, before processors need to read the new positions of nonlocal particles/cells in the force computation phase, they will already have deleted their old data and received the new data by explicit message passing when building locally essential trees. The finer-grained coherence needed in building a globally shared tree is not an issue *per se* in the locally essential trees approach, since a globally shared tree is not built at all. (Since the sum of the locally essential trees is much larger than the global shared tree, this approach can be seen as trading off redundancy for coherence.) However, a processor does have to maintain several checks and orderings between receiving, transmitting, and merging data when building locally essential trees. Some of the complications introduced by merging data into these trees are listed in Appendix B, and are very difficult to program and debug. Details of merging can be found in Salmon [1990], as can descriptions of several more-subtle complications in building locally essential trees that we have glossed over here.

Finally, the structure of communication is clearly large grained and sender initiated in Salmon's approach, rather than demand driven and fine grained as in our SAS implementations.

The design, programming, and debugging complexity of managing communication through building locally essential trees is clear, especially when compared with the fact that none of this has to be done in a shared-address-space implementation. The fact that Salmon's message-passing version took several times longer to implement than our shared-address-space version indicates the difference in complexity (personal communication, J. K. Salmon, 1991). The difference is also demonstrated by our experience with having groups of graduate students implement the application on SAS-CC and message-passing machines, in a ten-week parallel programming project course at Stanford University. The SAS versions were produced very quickly and yield very good speedups on the DASH multiprocessor. A message-passing version, however, is yet to be completed within the time allotted for the project (even though the starting point for the message-passing versions is a high-level description of the algorithms, which even removes most of the conceptual complexity). The complexity and characteristics of the message-passing implementation also have significant implications for runtime behavior, as we shall see in Section 9.6.

It is worth noting that the locally essential trees cannot reasonably be used with *costzones* partitioning, even though *costzones* is the best partitioning scheme for an SAS abstraction. There are two reasons for this. First, *costzones* partitioning requires the complete Barnes-Hut tree to have been already built before the partitioning is done. This complete tree is not built at all in the locally essential trees approach, and even the locally essential trees which are built require the partitioning to have been done already. Second, even if the complete tree were available, the *costzones* scheme produces irregularly shaped—albeit physically contiguous—partitions, and does not have the regular rectangular partitions of ORB partitioning on which the locally essential trees approach relies.

9.6 Runtime Implications of the Locally Essential Trees Approach

We divide the implications for runtime behavior into three parts: (1) execution time overheads, (2) the management of temporal locality and its implications for the problem sizes that can be run on a machine, and (3) communication volume.

9.6.1 *Execution Time Overheads.* Building locally essential trees incurs a lot of overhead in both computation and communication. Two tree traversals are required in each of the $\log_2 p$ iterations in the BuildTree algorithm (see Figure 17), each involving evaluations of the opening criterion at every visited node. Merging received data into a locally essential tree is also a high-overhead operation. In fact, Salmon finds that the vast majority of the runtime overhead in his parallel message-passing implementation is due to building the locally essential trees [Salmon 1990].

Salmon breaks down the overheads in his parallel implementation by their source. He finds that the largest source of overhead in the entire application is the extra work done in the parallel implementation. By far the dominant source of this extra-work overhead is associated with building the locally essential trees, not with doing the partitioning [Salmon 1990]. The next highest source of overhead is the load imbalance or waiting time at synchronization events, which is also found primarily in the phase of building locally essential trees. The overhead of communication itself (the time that a processor spends sending and receiving data) is comparably small. A representative example is a run with 100,000 particles on 512 processors, in which the complexity overhead of building locally essential trees is about five times the waiting-time overhead and a hundred times the communication overhead (a complete set of data for various problem configurations can be found in Salmon [1990]). In an SAS implementation, locally essential trees do not have to be built at all. And the synchronization and communication overheads involved in collaboratively building the shared Barnes-Hut tree are not very significant relative to total execution time for typical ratios of problem size to number of processors (cache miss rates of under 1% are typical, and the major source of performance loss is a load imbalance that is present, even in the locally essential trees message-passing approach). Thus, our SAS implementation does not suffer either of the overheads that most substantially lower the performance of the message-passing implementation.

9.6.2 *Temporal Locality and Its Implications.* We have already stated that the amount of replication needed in our SAS implementation is equal to the amount of data needed to compute forces on a single particle and that this replication is done only in the cache, whereas in the locally essential trees approach it is the amount of data needed to compute forces on *all* of a processor's particles, and that nonlocal data are replicated in main memory as well. Both Salmon and we have measured the memory overhead of replication in the locally essential trees approach. Memory overhead is defined as the ratio of the size of the locally essential tree to the size of the local tree constructed using only the processor's assigned particles. The latter quantity is a good approximation to the per-processor main memory requirements in as SAS-CC implementation.[11] Our measurements are for a version of Salmon's code that has been altered to match the mathematical functionality of our SAS code and that we have run on a 32-processor iPSC/2 computer.

For 200K particles running on 256 processors with a low force calculation accuracy, Salmon found the extra memory overhead to be a factor of five! Our results for a range of parameters are shown in Table II. The amount of replication with locally essential trees is clearly very large compared to the memory requirements in a single address space. The replication overhead grows quickly with the number of processors and with the force calculation accuracy (i.e., with decreasing θ), and decreases with increasing number of particles. In contrast, the amount of replication in caches needed in our SAS implementation is very small—much smaller than the partition size—even though it includes both local and nonlocal data.

The large amount of replication in main memory has implications for the size of a problem that is possible to fit in memory on a message-passing machine. On an SAS-CC machine, there is no need for replication in main memory (see Sections 6 and 7), so the number of particles that can be run grows linearly with the number of processors (and hence memory). On the other hand, although the amount of data in a processor's partition stays about the same if both the number of particles and number of processors are doubled, Table II shows that the size of a processor's locally essential tree increases (this can also be seen from the data presented in Salmon [1990]). Figure 19 shows the resulting dramatic difference in the number of particles that can be accommodated as the number of processors and amount of memory increase, using data from Salmon's thesis [Salmon 1990] for the locally essential trees approach. In fact, these results assume that only the number of particles is scaled with the number of processors. If the accuracy parameter θ is also scaled as discussed in Section 7, (1) the amount of replication needed is larger (see Table II) and (2) the number of particles that

[11] In fact, it is an overestimate, since the amount of data per cell needs to be a little larger in the locally essential trees case than in our SAS approach, and since many cells that exist in different processors' local trees in Salmon's approach are in fact shared in the SAS implementation

Table II. Barnes-Hut: Ratio of Locally Essential Tree Size to Local Tree Size in Salmon's Approach

Num.	n = 4k			n = 8k			n = 16k		
Procs	$\theta=0.6$	$\theta=0.8$	$\theta=1.0$	$\theta=0.6$	$\theta=0.8$	$\theta=1.0$	$\theta=0.6$	$\theta=0.8$	$\theta=1.0$
8	3.16	2.96	2.45	2.64	2.46	2.12	2.28	2.12	1.86
16	5.34	4.82	3.86	4.16	3.78	3.07	3.36	3.07	2.56
32	9.18	7.93	5.98	6.78	5.86	4.61	5.18	4.53	3.62

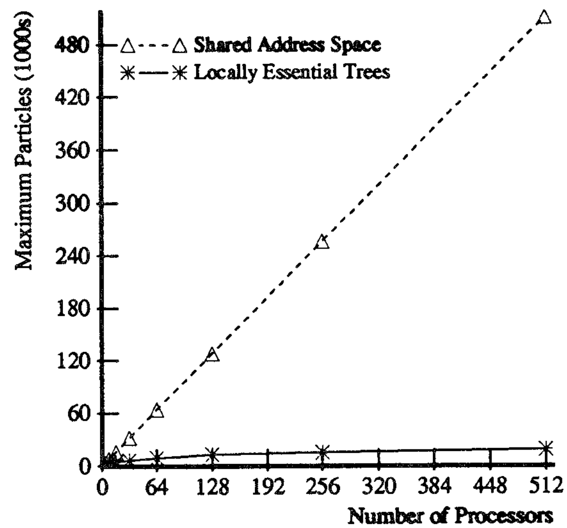


Fig. 19. Maximum number of particles that can be run using 100KB of memory per processor.

can be run grows even more slowly in the message-passing approach. Thus, even memory-constrained scaling does not allow the number of particles to increase linearly with the number of processing nodes, and the scalability of the locally essential trees approach is clearly limited.

Finally, we note that the consequences of a phase-structured replacement policy in message passing are even greater for partitioning schemes that do not incorporate physical locality but focus only on load balancing. Singh et al. [1992a] showed that such schemes can be quite successful on small to medium-scale SAS machines. These schemes, however, are very undesirable with phase-structured replacement on message-passing machines of any size. Because the particles assigned to a processor under these schemes are scattered all over the domain, every processor references almost the entire Barnes-Hut tree during force computation, and phase-structured replication severely limits problem size and scalability.

9.6.3 *Communication Volume.* While the number of data words communicated in the force calculation phase of the SAS approach is inherent to the Barnes-Hut algorithm, the amount of data communication in building locally essential trees is significantly larger. There are two reasons for this. First, a processor receives a lot of data in each iteration of the Buildtree algorithm (from its partner at that bisection) that it does not itself need but must pass on to another processor in a later iteration (see the earlier description of the Buildtree algorithm and Salmon [1990]). Our measurements over a range of problem parameters show that about 40% of the communication volume falls into this nonessential category. Second, the conservatism of the opening criterion used in building locally essential trees (as discussed in Section 9.5) often results in cell information being communicated that will not be used in the force computation, which uses the true (nonconservative) opening criterion.

It is possible to avoid the first source of extra communication. A processor can first use the ORB tree structure to determine all the processors that might need any data from its partition, and then send those data directly to all those processors, rather than sending data indirectly through corresponding partner processors at each node of the ORB tree. This also reduces the amount of merging and pruning of data that needs to be done in building locally essential trees. However, a fair amount of merging of received data remains, and the resulting communication is less structured and can cause more contention for network resources than in Salmon's implementation.

It may be argued that although there is extra communication in the ORB-based building of locally essential trees, the communication is coarse grained and well structured, which is better than the fine-grained, demand-driven communication in our SAS implementation. However, this is not a significant issue in these applications. We have seen that caching shared data keeps the communication-to-computation ratio very low. And the fine-grained communication that remains is handled efficiently, owing to the hardware address translation, protection, and buffer management discussed earlier. A change in communication management strategy is not likely to be necessary even on larger machines, particularly since the problem size also scales, so the communication-to-computation ratio does not increase very quickly (see Section 7.4).

9.7 Additional Complications for Locally Essential Trees in the FMM

The fact that the natural unit of parallelism in the FMM is a cell rather than a particle further complicates message-passing implementations using locally essential trees. A processor's locally essential tree in this case is the union of the four interaction lists of all cells in its partition (see Section 8) as well as all ancestors and children of its cells. The additional complications arise from two sources, both discussed in the context of partitioning in Singh et al. [1992a]. First, it is not only leaf cells that are partitioned in the FMM, but internal cells as well. Second, many cells will straddle the bisectors that ORB partitioning produces (unlike particles, which fall on one or the other side of a bisector). These border cells have to be partitioned separately for load balancing, which makes the resulting partitions irregular in the FMM. The resulting complications are:

(1) The cells that will exist in the current time-step are not known until locally essential trees are built. However, building locally essential trees requires the partitioning of cells (leaf and internal) to have been done. To solve this problem, internal cells might be partitioned as they are constructed during the building of locally essential trees. However, the cells are incomplete at this time; and their predicted costs will be inaccurate. In the Barnes-Hut application, this incompleteness of cells caused a more conservative opening criterion to be used; in the FMM, it would lead to poor load balancing as well.[12]
(2) The ownership of border cells (leaf or internal) has to be determined. The identities of processors that share border cells can be determined by an ORB tree traversal. However, determining which processor should be assigned which cells is more difficult.
(3) The fact that partition borders are irregular even with ORB implies that conservatively larger regular partitions must be defined to build locally essential trees.

Solving the above problems requires additional ORB tree traversals, each involving substantial computation and communication. Since we are unaware of a message-passing implementation of the adaptive FMM, we do not know how large the resulting overhead will be. Clearly, it will be significantly larger than in the Barnes-Hut case.

9.8 Summary of Drawbacks of the Locally Essential Trees Approach

In summary, then, the following are the main drawbacks of the locally essential trees approach for message passing, relative to our shared-address-space approach.

—Complexity of implementation and debugging.
—Performance overheads that result directly from this complexity.
—Restrictions on the number of particles that can be run and on the scaling of the number of particles with the number of processing nodes.
—The need for a modified cell opening criterion in Barnes-Hut and the general inability to use opening criteria that depend on the distribution of particles within a cell.
—The greater volume of communication.
—The inappropriateness of straightforward, load-balanced partitioning schemes even on small-scale multiprocessors.
—The fact that the approach cannot reasonably be used with *costzones* partitioning.

[12] A possible solution for partitioning internal cells traverses the ORB tree three times: first, to partition particles; second, to exchange particles in border cells and hence construct these cells completely; and third, to partition cells.

Finally, the locally essential trees approach relies on a property of classical N-body applications, which is that the need to examine an interaction between two entities (bodies or cells) depends only on the entities themselves and the distance between them, not on any other factors in the domain. More-complicated interactions exist in other applications that use a hierarchical N-body approach. In a hierarchical radiosity application that we have studied, for example, the need for subdividing a patch (opening a cell) is also determined by the extent to which two patches are directly visible from each other, which requires knowing what other patches lie between the two [Singh et al. 1992a]. Building locally essential trees as described in this section is not feasible for such applications.

9.9 Alternatives to Locally Essential Trees for Message Passing

The locally essential trees approach has the advantage that the sender of data initiates and manages communication, which makes it quite easy to increase communication granularity and impose structure on the communication. Despite the drawbacks discussed above, this approach may well be the best way to implement the Barnes-Hut application on a message-passing machine with high message overheads. Other message-passing approaches can be developed, however, in which the receiver initiates communication (as is natural to shared-address-space implementations). These approaches are more flexible for other opening criteria and other applications such as the FMM, and the second among them may be prefereable even for the Barnes-Hut application in the long run, particularly as message overheads become smaller.

9.9.1 *Virtual Pointers*. One approach is to have every processor hold a virtual copy of the tree, which includes actual data for the parts of the tree that are in its partition, and pointers to the owning processors of other parts. When a processor encounters such a "virtual pointer" in its traversal of the tree, it can either request the necessary data from the processor pointed to, or ask it to perform the required computation. Managing the virtual pointers and rebuilding the tree are complex tasks, however, since both the processors to which the virtual pointers point and even whether a given pointer is virtual or local change as the tree and partitioning change (see Appendix C). The overheads of this management may not be any better than those of building locally essential trees. However, the virtual pointers technique can be used just as well with *costzones* partitioning as with ORB.

9.9.2 *Hashing*. A better alternative is to distribute logically shared data statically among processing nodes. A simple hashing algorithm can be used to allow processors to determine where a given particle has its "home" and request it from that home when necessary.[13] This is very close to a shared-address-space style of programming, except that the programmer implements an application-specific shared address space. Since cells do not persist across time-steps, the hashing is a little more complicated for cells than for parti-

[13]Such an approach has been used in Warren and Salmon [1993] since this writing.

cles. A consistent scheme must be constructed so that all processors refer to a given cell by the same identifier. This approach avoids many of the flexibility disadvantages of the locally essential trees approach and allows the programmer to manage temporal locality in any manner desired, including by emulating a hardware cache in the application program. However, it has some disadvantages:

(1) The natural communication is demand driven and very fine grained. At least on current message-passing machines, individual data request or reply messages will have to be grouped together before sending them out. More significantly, substantial programming contortions will be required to hide the large latencies of messages by overlapping other computations with them.

(2) At every reference, the application program has to check if the datum being referenced is local, if it is nonlocal but has already been cached (replicated) locally, or if it has to be fetched from another processing node (in which case the translation and messaging have to be done). Such checks do not have to be made in the locally essential trees approach.

(3) There is now a need to provide coherence at a fine granularity, just as in the SAS implementation, when collaboratively building the single Barnes-Hut tree. This can be quite complicated and may motivate turning off caching for this phase.

(4) A key underpinning of this static data distribution approach is that good performance should not require redistribution of data as partitions change. That is, that a processor's partition of the tree should not have to reside in its local memory. If data redistribution is required, the tasks of naming and providing coherence become much more complicated. In Section 6, we showed that data distribution is not very important for the relatively small latencies encountered on SAS machines. On message-passing machines, the latencies for remote references are substantially larger. Having partitions be in local memory may therefore be more important, particularly in the FMM application (see Figure 13(b)).

Thus, while this SAS-like approach is more flexible than locally essential trees, it is not at all clear whether it is any easier to implement or has less runtime overhead. A recent implementation by Warren and Salmon [1993] indicates that the greatest benefit is in flexibility—to handle opening criteria and even FMM-like methods—and not in programming complexity or execution time. Nonetheless, we believe that it is a useful approach for supporting many classes of irregular, unpredictable applications which exhibit temporal locality on message-passing machines, particularly together with emulating a cache in software (see Singh et al. [1994] as well), and may behoove us to provide efficient support for in a language, runtime system, and compiler for these machines.

From the above discussion, it seems clear that providing system support for a shared address space—especially with coherent caching of shared data—is very advantageous for hierarchical N-body applications from the algorithm

Table III. Results for Software-Supported Shared Address Space (SAM) for Barnes-Hut (n = 25K, p = 32)

Machine	*Address Translation (Naming) Overhead (% of exec. time)*	*Speedup over 1 proc.*
Intel iPSC/860	19	10
Intel Paragon	17	10
Thinking Machines CM-5	20	16

design, programming, and runtime overhead points of view. The fact that individual communications in our shared-address-space implementations are fine grained is more than compensated for by two facts: that caching shared data is very effective at keeping communication-to-computation ratios low for N-body problems and that the hardware mechanisms for naming and cache coherence support fine-grained communication efficiently.

9.10 Is Hardware Support Necessary?

An interesting architectural question that remains is whether the cache-coherent shared address space should be supported in hardware. The alternative is to support it in a software layer on a message-passing machine. Several people have built such optimized software layers. Scales and Lam [1994] have implemented the Barnes-Hut application on one such optimized system (called SAM) that they have designed. Table III shows some of their results on three current message-passing machines for a 25K-particle problem. Thirty-two processors are used in all cases. For comparison, our SAS implementation yields a speedup of about 28 on 32 processors on the DASH machine for the same problem. While we do not claim that these results are decisive, it appears that hardware support at least for naming nonlocal data is very useful for applications with fine-grained, irregular referencing and communication patterns.

Before we conclude, let us note one other, more general disadvantage of the explicit message-passing paradigm. In any message-passing scheme, the burden is on the application programmer to get *all* the communication management (keeping track of data, coherence, etc.) exactly right to obtain a correct program. As we have seen, this is difficult for a programmer to do in nonuniform, dynamically changing applications that require long-range communication. In SAS implementations, on the other hand, the programmer's main concerns in communication management are to use an appropriate partitioning scheme that minimizes communication, and perhaps to manage data distribution explicitly (although the latter is not required in these applications). The programmer can in many cases afford to be somewhat lax about these issues, especially in complex but unimportant phases of a program, since the penalty for this laxity is not correctness but only a small amount of performance. Thus, even if some applications might benefit from mechanisms that are traditionally associated with message passing—such as bulk transfer of data—having an efficiently supported shared address space remains very useful, and these mechanisms should be built on top of an efficient shared-address-space machine.

10. CONCLUDING REMARKS

We have examined the properties of an important class of parallel applications—those that use hierarchical N-body methods—and how these properties impact the design of current and future parallel machines. We have shown the following results.

First, we examined the kinds of locality available in an architecture and how they interact with these applications. We found that the key form of locality is temporal locality on both local as well as communicated data. The degree of temporal locality on communicated data is high enough that excellent parallel performance is obtained, despite the irregular and dynamically changing data-referencing patterns of the applications. Hardware caches exploit this temporal locality very effectively, and there is no need to manage locality in physically distributed main memory, which would be quite difficult for these applications under program control. We have found this to be true of other classes of irregular applications as well, such as some in computer graphics [Singh et al. 1994].

Then, we examined the implications for scaling the applications to run larger problems on larger machines. We demonstrated the following results for scaling.

—Using scaling models that reflect the goals of the application scientist leads to different results about the scaling of important application characteristics than more-naive scaling models.

—Time-constrained scaling is more realistic than memory-constrained scaling for these and other scientific applications.

—The input data set size per processor, and the memory requirements in a shared address space, diminishes under time-constrained scaling.

—The communication-to-computation ratio grows slowly under time-constrained scaling.

—The size of the most important working set, which helps determine the size of cache that is desirable, grows under time-constrained scaling (and grows even more quickly under memory-constrained scaling).

—The number of particles that can be run on a message-passing machine does not scale linearly with the number of processors even under memory-constrained scaling (without substantial programming contortions and performance loss), although it does on a shared-address-space machine.

—The per-processor local memory requirements on message-passing machines are likely to decrease under time-constrained scaling.

Although our results show that the communication-to-computation ratio and the working-set size increase slowly under realistic scaling rules, both these important parameters are small enough with good partitioning schemes that we are likely to obtain very good parallel performance from hierarchical N-body applications for some time to come.

Finally, we examined the interaction between the communication properties of the applications and the use of a shared address space or explicit message passing between private address spaces as the method for interprocessor communication. We found that an efficiently supported shared address space (particularly with coherent replication) has substantial advantages in programming complexity over explicit message passing—owing to the irregular, dynamically changing communication needs—and that the complexity of message passing translates to substantial performance overheads as well. Our experiences lead us to believe that as people solve more and more complex and irregular problems on parallel machines—in order to model physical phenomena more accurately and efficiently—and as people run more-general workloads than numerical scientific applications [Singh et al. 1994], the advantages of a hardware-supported coherent shared address space over explicit message passing become substantial.

APPENDIX A. COMMUNICATION COMPLEXITY OF THE FMM

In this appendix, we derive the communication complexity of the FMM, assuming a uniform distribution of particles. Suppose that every processor's partition contains g-by-g leaf cells ($g = \sqrt{n/p * s}$, where n is the total number of particles, p the number of processors, and s the number of particles per leaf cell). If the root of the tree is level 0, then every processor can be assumed to own a subtree whose root is at level ($\log g - 1$) and which has g-by-g leaf cells at level ($\log n - 1$). Cells at levels 0 through ($\log g - 1$) are shared among processors. Let us first ignore the direct particle-particle computations at the leaf level and look only at the communication incurred when cells compute their interactions with cells in their interaction lists.

Every processor's partition (from level ($\log g - 1$) to the leaves) can be thought of as a pyramid of cells whose base is of size g-by-g. The volume of interaction-list communication due to the cells in this pyramid is proportional to the number of cells outside the pyramid that the cells in the pyramid interact with. Given the interaction patterns of the FMM, this amounts to the number of children of all cells on the sides of a ($g/2 + 2$)-by-($g/2 + 2$) pyramid (i.e., a pyramid that just surrounds the original, except for the leaf level). This number of cells is

$$
\begin{aligned}
&= (\text{internal cells on faces of pyramid} + \text{edge cells of pyramid} + \text{vertex cell}) \\
&\quad * 4 \text{ children} \\
&= \frac{g}{2}\left(1 + \frac{1}{2} + \frac{1}{4} + \cdots \log g \text{ terms}\right) * 4 + (\log g - 1) * 4 + 1 * 4 \\
&= 16(g + \log g) \\
&= 16\left(\sqrt{\frac{n}{p * s}} + \log \sqrt{\frac{n}{p * s}}\right)
\end{aligned}
$$

since $g = \sqrt{n/(p * s)}$.

For the cells in levels 0 through (log $g - 1$), the two extreme points are (1) a processor that owns all the cells from the root of the whole tree (level 0) to the root of its partition at level (log $g - 1$) and (2) a processor that owns no cells above level (log $g - 1$). In the former case, the processor owns (log p) cells in the levels above level (log $g - 1$), which communicate with approximately ($8 * \log p$) other cells outside the processor's partition. Thus, the total communication for a processor that owns the path from the root of the whole tree to the root of its partition at level (log $g - 1$) is

$$8m^* 2 * \left(\sqrt{\frac{n}{p * s}} + \log \sqrt{\frac{n}{p * s}} \right) + \log p$$

where m is the number of terms used in the multipole expansions. The total communication for a processor that does not own any cells above its partition at level (log $g - 1$) is of order

$$8m * 2 * \left(\sqrt{\frac{n}{p * s}} + \log \sqrt{\frac{n}{p * s}} \right).$$

For the direct particle-particle interactions that leaf cells compute with cells in their near-field, the number of nonlocal leaf cells with which these interactions are computed is the number of cells adjacent to the base of the pyramid, which is $(4g + 4)$ or $4 * (\sqrt{n/(p * s)} + 1)$. The processor communicates with half these cells, resulting in a communication of order $s * \sqrt{n/(p * s)}$, or $\sqrt{(n * s)/p}$.

APPENDIX B. COMPLICATIONS IN MERGING DATA INTO A LOCALLY ESSENTIAL TREE

Some examples of the complications involved in merging data into a locally essential tree are the following.

—A cell into which some piece of received data (a body or another cell) is going to be merged may itself be scheduled for transmission (in its original form) at the end of that iteration. For this reason, cells that are scheduled for transmission must be marked and, if they are to be modified in the merge phase, copied before modification.

—Data that are eligible for pruning may also be scheduled for transmission in the same iteration. These data cannot be deleted until sent. The deletion procedure must therefore check whether they have been marked for transmission and then mark them for the sending procedure to remove after transmission.

—The local tree may not yet be refined to the level at which an internal cell received from another processor must be inserted. The latter cell has a fixed position and level in its tree and must be inserted into a cell at its parent's level. The local tree may therefore have to be refined to that level first.

—The center-of-mass information for a cell must be checked for being up-to-date on two occasions and perhaps recomputed: when it is enqueued for transmission (since the structure of the cell might change between this time and when it is actually transmitted), and when the subtree below it is to be pruned.

—Even data that are not needed by a processor itself are merged into its tree and later pruned.

APPENDIX C. ORB WITH VIRTUAL POINTERS

In the virtual pointers approach, the Barnes-Hut tree is broken down into a set of subtrees, each being the largest subtree that is owned by a single processor. Every processor then holds its subtrees, the part of the tree (some "upper" nodes and edges) that is not owned by any single processor, and pointers to the owning processors of the other subtrees. These pointers give the scheme its name. Attaching the subtrees they point to to the virtual pointers (edges) on a processor would yield the entire Barnes-Hut tree. When a processor reaches a virtual pointer in the course of a tree traversal and needs information from the subtree underneath it, it sends a message to the owning processor as revealed by the pointer. For example, if a processor determines during the force computation traversal for a particle that a cell pointed to by a virtual pointer must be opened, it may simply send the owning processor a message with the particle information. The owning processor then computes the forces due to the subtree "underneath" that cell and send back the result. Of course, the structure of the tree, the partitions, and therefore the virtual points change across time-steps. Maintaining the virtual pointers to keep the tree up-to-date is still the programmer's task, as is catching references to virtual pointers and translating them to the appropriate messages. One mechanism to rebuild the tree with virtual pointers is as follows. The starting point is the tree structure with virtual pointers from the previous time-step, and the new partitions of particles for the new time-step computed using ORB. The tree-building process in this case includes the following steps:

```
Remove all particles (not internal cells or virtual
  pointers) from local tree.
Load particles in local partition into tree:
  if particle goes into virtual pointer, send
    to that processor to load (along with
    sender process id)
  if new cell is created, not part of owned subtree,
    figure out owner by some rule consistent across
    processors and send to owner (with sender id)
Traverse tree to
  compute c.m. of owned subtrees
  change ownership information of local subtrees
  enqueue new subtrees, those whose ownership has
    changed, and those that are no longer needed
Broadcast enqueued subtrees
Merge received subtrees into tree if owner, or
  set up virtual pointers
```

This method can use the original Barnes-Hut opening criterion just as our shared-memory implementation can. However, the communication in it is not as regular or block-structured as the method of locally essential trees with ORB partitioning. Like the shared-memory implementation, this method includes communication during the force computation phase, and the communication can be on-demand or larger grained. While it reduces replication needs relative to the locally essential trees method, the less-structured communication and the need for global communication of virtual pointers may hurt its relative performance.

ACKNOWLEDGMENTS

We would like to thank Joshua Barnes and Leslie Greengard for providing us with the sequential programs and for their patience in many electronic mail discussions. Dennis Roger, Eric Bruni, Chris Holt, and Maneesh Agrawala were responsible for a lot of the implementation of the parallel codes. John Salmon very kindly shared his parallel message-passing Barnes-Hut code with us, and Evan Torrie performed several experiments with it.

REFERENCES

AARSETH, S. J., HENON, M., AND WIELEN, R. 1974. *Astronomy Astrophysics 37*.

APPEL, A. A. 1985. An efficient program for many body simulation. *SIAM J. Sci. Stat. Comput. 6*, 85–93.

BARNES, J. E. AND HUT, P. 1989. Error analysis of a tree code. *Astrophysics J. Suppl.* 70, 389–417.

BARNES, J. E. AND HUT, P. 1986. A hierarchical $O(N \log N)$ force calculation algorithm. *Nature 324*, 4, 446–449.

CHAIKEN, D., KUBIATOWICZ, J., AND AGARWAL, A. 1991. LimitLESS directories: A scalable cache coherence scheme. In *4th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, 224–234.

CHAN, T. 1990. Hierarchical algorithms and architectures for parallel scientific computing. In *Proceedings of ACM Conference on Supercomputing*. ACM, New York.

CHORIN, A. J. 1973. Numerical study of slightly viscous flow. *J. Fluid Mech. 57*, 785–796.

DENNING, P. I. 1968. The working set model for program behavior. *Commun. ACM 11*, 5 (May), 323–333.

EGGERS, S. AND KATZ, R. 1989. The effect of sharing on the cache and bus performance of parallel programs. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems* (May). ACM, New York, 257–270.

FOX, G. C. 1988. A graphical approach to load balancing and sparse matrix vector multiplication on the hypercube. In *Numerical Algorithms for Modern Parallel Computer Architectures*. Springer-Verlag, New York, 37–62.

GHARACHORLOO, K., GUPTA, A., AND HENNESSY, J. 1991. Performance evaluation of memory consistency models for shared-memory multiprocessors. In *4th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, 245–257.

GOLDSCHMIDT, S. R. AND DAVIS, H. 1990. Tango introduction and tutorial. Tech. Rep. CSL-TR-90-410, Stanford Univ., Stanford, Calif.

GREENGARD, L. 1987. *The Rapid Evaluation of Potential Fields in Particle Systems*. ACM Press, New York.

GREENGARD, L. AND ROKHLIN, V. 1987. A fast algorithm for particle simulation. *J. Comput. Physics 73*, 325.

GUPTA, A., WEBER, W.-D., AND MOWRY, T. 1990. Reducing memory and traffic requirements for scalable directory-based cache coherence schemes. In *Proceedings of the 1990 International Conference on Parallel Processing*. 312–321.

GUSTAFSON, J. L., MONTRY, G. R., AND BRENNER, R. E. 1988. Development of parallel methods for a 1024-processor hypercube. *SIAM J. Sci. Stat. Comput. 9*, 4 (July), 532–533.

HAGARSTEN, E., HARIDI, S., AND WARREN, D. H. D. 1990. The cache coherence protocol of the data diffusion machine. In *Cache and Interconnect Architectures in Multiprocessors*. Kluwer Academic, Amsterdam.

HANRAHAN, P., SALZMAN, D., AND AUPPERLE, L. 1991. A rapid hierarchical radiosity algorithm. In *Proceedings of SIGGRAPH*. ACM, New York.

HERNQUIST, L. 1987. Performance characteristics of tree codes. *Astrophysics J. Supp. 64*, 715–734.

HOLT, C. AND SINGH, J. P. 1995. Hierarchical N-body methods on shared address space multiprocessors. In *Proceedings of the 7th SIAM Conference on Parallel Processing for Scientific Computing* (Feb.). SIAM, Philadelphia, Pa.

JAMES, D. V. 1989. P1596-SCI coherence overview. Tech. Rep. 27, SCI Committee, Mar.

LENOSKI, D., LAUDON, J., GHARACHORLOO, K., GUPTA, A., AND HENNESSY, J. 1990. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture* (May). 148–159.

LI, K. AND HUDAK, P. 1989. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst. 7*, 4, 321–359.

ROKHLIN, V. 1985. Rapid solution of integral equations of classical potential theory. *J. Comput. Phys. 60*, 187–207.

SALMON, J. K. 1990. Parallel hierarchical N-body methods. Ph.D. thesis, California Institute of Technology, Dec.

SCALES, D. AND LAM, M. 1994. A comparison of shared and distributed memory multiprocessors for performance and programmability. To appear as a Tech. Rep. of the Computer Systems Laboratory, Stanford University.

SCOTT, S. 1991. A cache coherence mechanism for scalable, shared-memory multiprocessors. In *Proceedings of the International Symposium on Shared Memory Multiprocessing* (Apr.). 49–59.

SIMONI, R. AND HOROWITZ, M. 1991. Dynamic pointer allocation for scalable cache coherence directories. In *Proceedings of the International Symposium on Shared Memory Multiprocessing* (Apr.). 72–81.

SINGH, J. P. 1993. Parallel hierarchical N-body methods and their implications for multiprocessors. Ph.D. thesis, Stanford Univ., Stanford, Calif.

SINGH, J. P. AND HENNESSY, J. L. 1992. Finding and exploiting parallelism in an ocean simulation program: Experiences, results, implications. *J. Parall. Distrib. Comput. 15*, 1 (May), 27–48.

SINGH, J. P., GUPTA, A., AND LEVOY, M. 1994. Parallel visualization algorithms: Performance and architectural implications. *IEEE Comput. 27*, 7 (July).

SINGH, J. P., HENNESSY, J. L., AND GUPTA, A. 1993. Scaling parallel programs for multiprocessors: Methodology and examples. *IEEE Comput. 26*, 7 (July), 42–50.

SINGH, J. P., HOLT, C., TOTSUKA, T., GUPTA, A., AND HENNESSY, J. L. 1992a. Load balancing and data locality in hierarchical *N*-body methods. *J. Parall. Distribut. Comput.* To appear. Preliminary version available as Stanford Univ. Tech. Report no. CSL-TR-92-505, Jan.

SINGH, J. P., WEBER, W.-D., AND GUPTA, A. 1992b. SPLASH: Stanford Parallel Applications for Shared memory. *Comput. Arch. News 20*, 1, 5–44. Also Stanford Univ. Tech. Rep. CSL-TR-92-526, (June).

TORRELLAS, J., LAM, M. S., AND HENNESSY, J. L. 1990. Measurement, analysis, and improvement of the cache behavior of shared data in cache coherent multiprocessors. Tech. Rep. CSL-TR-90-412, Stanford Univ., Stanford, Calif.

WARREN, M. S. AND SALMON, J. K. 1993. A parallel hashed oct-tree n-body algorithm. In *Proceedings of Supercomputing '93* (Nov.). 12–21.

WEBER, W.-D. AND GUPTA, A. 1989. Analysis of cache invalidation patterns in multiprocessors. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems* (Apr.). ACM, New York, 243–256.

Received September 1993; revised January 1994; accepted June 1994