Parallelization Requirements for Heirarchically Structured Applications on Coarse-Grained Parallel Computers

(*Preliminary Version/Under Revision*)

Sanjay Goil [1]
School of CIS &
Northeast Parallel Architectures Center
Syracuse University
Syracuse, NY 13244
email: sgoil@top.cis.syr.edu

Sanjay Ranka [2]
School of CISE
University of Florida
Gainesville, FL 32611
ranka@cis.ufl.edu

[1]The work of this author was supported in part by NASA under subcontract #1057L0013-94 issued by the LANL. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

[2]The work of this author was supported in part by NSF under ASC-9213821 and AFMC and ARPA under contract #F19628-94-C-0057. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

Abstract

In this paper we address a class of problems consisting of highly structured computations on data sets that are described by hierarchical data structures. These are often represented as *tree* structures to optimize data storage requirements and perform efficient queries for data access. Specifically, applications that are dynamic and perform many iterations on data are of interest to us, since the requirements for data evolve over time and require modifying data structures incrementally. The computational relationship between the subdomains is known only at runtime, and may change between computation phases. Parallelization of such applications requires efficient distributed data management and has received some attention recently. We study the computational structure of a few irregular applications falling in this category. This helps us to discuss the needs of choosing an appropriate data structure and the issues of data partitioning, load balancing and communication requirements for these applications. Most recent efforts have been application specific and solutions are not portable across applications or parallel computing platforms. We wish to characterize requirements for primitives and runtime software support needed to parallelize irregular applications.

In this paper we present a detailed study of the computational structure of the following eight hierarchical applications: *N-body simulation, Molecular Dynamics, Hierarchical Radiosity, Volume Rendering and Ray Tracing, Spatial databases, Adaptive meshes and, Image Compression.*

This allows us to identify requirements for parallelization across a broad spectrum of applications. Each of these applications uses a tree data structure. We enumerate tree algorithms that are used for solving these applications and discuss their parallelization. Next, we present an outline of common primitives that are required for these applications. Requirements for a runtime support system are presented for obtaining effective parallel solutions using the above primitives on coarse-grained distributed memory machines.

1 Introduction

Most previous research on coarse-grained MIMD machines has concentrated on parallelizing scientific High-level languages like Fortran-D [17], Vienna-Fortran [70] which support parallel operations on uniform arrays. Efficient parallelizations of dynamic and irregular problems have received much attention only recently. Runtime support for irregular structures is available in PARTI [45]. These approaches take advantage of static data accesses and communication patterns. They provide little in terms of efficient solutions for dynamically changing data distribution and communication patterns.

We study a class of problems that consists of highly structured computations on sets of subdomains that are coupled hierarchically. The computational relationship between the subdomain is known only at runtime, and may change between computation phases. Parallelization of these applications on distributed memory machines require exploitation of the hierarchical nature both for load balance as well as maintaining locality to reduce communication. The following applications have been investigated, N-body simulation, Molecular Dynamics, Hierarchical Radiosity, Volume Rendering and Ray Tracing in Computer Graphics, Adaptive meshes, Databases and Image Compression. These applications are usually efficiently represented and manipulated by using sparse data structures such as graphs, trees, and lists in sequential algorithms to reduce data storage sizes as well as to gain asymptotic performance.

The communication networks and software available on coarse-grained machines make local accesses at least an order of magnitude faster then nonlocal accesses. This is further accentuated by high latency costs of communication software on distributed-memory machines. Effective parallelization of these applications on coarse-grained MIMD machines requires careful attention for the following reasons:

- The amount of work done by the parallel algorithm should be within a small constant factor of the amount of work done by the sequential algorithm, since the number of processors used in practice is limited to from ten to a thousand. Parallel algorithms, which may be theoretically optimal, have limited use if the constants involved are large.
- The off-processor accesses generated by these applications are highly unstructured and may have many hot spots.
- For many applications, the data structures used have inherent locality of access and/or change incrementally. Exploitation of this information is necessary for efficient use of the various levels of memory hierarchy present in these architectures (register, caches, local accesses, nonlocal accesses, etc.). This requires fast methods for partitioning, repartitioning, replication, and migration of data.
- Static scheduling, dynamic scheduling, and load balancing are required to reduce processor

idle time, and synchronization is required to achieve program correctness.

Our goal is to study the scalability of these applications on coarse-grained machines in a relatively architecture-independent fashion. We discuss the structure of the above five applications in detail and present the data partitioning, communication and load balancing primitives that are required for their efficient parallelization.

The motivation for this work is from the recent efforts on parallelizing N-body applications on parallel machines, and extending their use to applications in computer graphics. Most implementations have been very specific to the problem and most often architecture dependent. The various phases in the implementation of the algorithm require immaculate control on the computation and communication functions. Our goal here is to develop an architecture independent infrastructure for parallelization of treecodes and apply it to challenging visualization applications such as ray-traced volume rendering [34], Ray Tracing and Hierarchical Radiosity [3], Spatial databases, Molecular Dynamics, Adaptive Meshes and Image Compression. Singh [57] discusses the parallelization needs of some graphics applications on a shared memory machine and elaborates the difficulties of parallelization on distributed memory machines.

The rest of the paper is described as follows. Section 2 discusses the distributed memory model of computation. Section 3 presents a survey of eight hierarchical applications and their computational structure. These include N-body simulation, Molecular Dynamics, Volume Rendering, Ray tracing, Radiosity, Databases, Adaptive meshes and Image Compression. Primitives for the applications for distributed memory machines are presented in section 4. Section 5 elaborates the use of primitives and requirements for a software system for hierarchical applications.

2 Distributed-Memory Machines

A distributed-memory machine consists of a set of processors linked by interconnection networks. Each processor has its own memory that is directly accessible only by itself. Data exchange and global operations among processors are accomplished through message passing or appropriate hardware support [29, 16].

The parallel-processing literature abounds with parallel algorithms designed for processors connected through special interconnection networks such as hypercubes, meshes, rings, or toroids. Available commercial architectures (IBM SP series, CM-5, nCUBE, and Intel Paragon) have subsets of the following properties:

1. *Distance.* With new routing techniques such as wormhole routing and randomized routing (as seen on the CM-5) [32, 29, 13, 43], the distance between communicating processors is less of a determining factor on the amount of time required for communication.

2. *Latency.* The start-up time for sending a message is an order of magnitude more than the cost of transmitting a few bytes of information. The latency of nonlocal access can be significantly reduced by using active messages [63, 9, 55]. Further, hardware support for accessing data from nonlocal memory (or moving pages into local memory) can provide a reduction in the effective latency [25, 14, 1].

3. *Node Contention.* A node can receive only one message (or a limited number of messages) at a time.

4. *Link Contention.* If two message paths have common links, the time required for their transmission may be affected. This effect is limited due to the use of virtual channels and because link bandwidth is much larger than node-interface bandwidths. Theoretical models typically assume only one virtual channel per link.

5. *Cross Section Bandwidth.* For machines which have an underlying mesh architecture (like Intel Paragon), the cross-section bandwidth may become a bottleneck.

Hardware support for cache coherence in machines (e.g., DASH, KSR) can significantly reduce programmers' efforts to maintain coherence of replicated data, and context switching can allow for multiple threads at low overheads [1]. We would concentrate on machine models without hardware support for shared memory and multi-threading. However, most of the techniques developed are of general applicability. One of our goals is to develop architecture-independent primitives that can be implemented on a wide variety of distributed-memory machines.

The specific interconnection network for which primitives would initially be developed for, assume that it can support arbitrary permutations (i.e., at a given time each node can send and receive one message from an arbitrary processor). This two-level memory model distinguishes elements needed for computation as being local or nonlocal to a processor. The logP [37] model and the postal model [46] are theoretical models, based on the above philosophy, for coarse-grained machines. Due to larger link bandwidths, as compared to node interface bandwidths, many networks have behavior close to this model (e.g., the IBM SP Series and the CM-5). Later specific networks such as hypercubes and meshes would be studied. A large number of commercial and research architectures use these interconnection networks.

3 A Survey of Hierarchical Applications

In this section, we describe the structure of several applications and describe the inherent parallelism available in them. We discuss the main efforts in parallelizing each of the application that has been reported in the literature.

3.1 N-Body methods

Computational methods to track the motions of bodies that interact with each other have been subjects of study for a long time in the areas of astrophysics, semiconductor device simulation, molecular dynamics and plasma physics. The N-body problem computes the state (position and velocity) of N bodies at a given time $t > 0$, given an initial state at $t = 0$. The most common approach is to iteratively calculate the solution by calculating all forces over a sequence of small time steps. Within each timestep the instantaneous acceleration is approximated by the instantaneous acceleration at the beginning of the time step, which is done by directly summing the force induced by each of the other $N - 1$ bodies. This method is conceptually simple and vectorizes well but its $\Theta(N^2)$ arithmetic complexity rules it out for large-scale simulations involving millions of bodies.

Many physical systems exhibit a large range of scales in their information requirements, in both space and time. A point in physical domain requires progressively less information at a lesser frequency from parts of the domain that are farther away from it. Applying this fundamental insight Appel [2] and Barnes and Hut [4] were the first to propose faster N-Body algorithms. Appel's method requires $O(N)$ steps but has a bigger constant attached to it, whereas the Barnes-Hut method is $O(NlogN)$ and is used in practice. N-body simulations using adaptive tree data structures are referred to as *treecodes*. Parallel implementation of the Barnes-Hut method has been fairly recent and is reported in Salmon and Warren [60, 61].

In an astrophysical N-body simulation, force interactions between celestial bodies are calculated according to the laws of Newtonian physics. Accelerations induce by forces due to the other $N - 1$ bodies are calculated as

$$\frac{d^2\vec{x}_i}{dt^2} = \sum_{j \neq i} \vec{a}_{ij} = \sum_{j \neq i} -\frac{Gm_j\vec{d}_{ij}}{|d_{ij}|^3}, \qquad \vec{d}ij = \vec{x}_i - \vec{x}_j.$$

This formulation leads to a $O(N^2)$ algorithm. Several approximate methods have been used to reduce the overall time and allow larger simulations to be done. The approximation that reduces the interactions using treecodes is stated as [60]

$$\sum_j \frac{Gm_j\vec{d}_{ij}}{|d_{ij}|^3} \approx \frac{GM\vec{d}_{i,cm}}{d^3{}_{i,cm}} + ...$$

where $\vec{d}_{i,cm} = \vec{x}_i - \vec{x}_{cm}$ is the vector from $\vec{x}_i$ to the center-of-mass of the particles that are summed in the left hand side in the above expression. Quadropole, octopole and further terms in the multipole expansion can be included for better approximations.

In the Fast multipole method presented by Greengard and Rokhlin [22] particles are organized into cells and then cell-cell interactions are computed prior to the force calculation step. Once this has been determined, the force on a single particle can be obtained in a time independent of N,

resulting in a *O(N)* scaling. The interactions here are more complex and the hidden constants in the notation are not very clear. Each cluster is characterized by a *multipole expansion* computed by traversing the tree in an upward phase. This is followed by a downward phase to combine multipole expansions and to propagate them to the leaves. At the end of the downward phase each leaf has data to compute the force induced by bodies in the *far field*, which is the area outside of this leaf and its neighbors.

The Barnes-Hut algorithm begins by constructing a tree, inserting the bodies into the cluster hierarchy one at a time. First an octree partition of the three-dimensional box (a region in space) is computed enclosing the set of bodies. The partition is computed recursively by dividing the original box into eight octants of equal volume until each undivided box contains exactly one body. Figure 1 is an example of recursive partition in two dimensions, the corresponding quadtree, which is called the Barnes-Hut (BH) tree. To minimize the number of interactions, each body computes interactions with the largest clusters for which the approximation can be applied. The bodies are added to the tree one at a time. The *i*th body is added into the BH-tree with *i - 1* bodies, the newly inserted body descending down the tree until it reaches a box of which it is the sole occupant. If the body reaches a leaf, the leaf is subdivided until each of the two bodies is in its own box.

Each internal node of the BH-tree represents a cluster. Once the BH-tree has been built, the centers-of-mass of the internal nodes are computed by traversing the tree bottom-up. Once that is done, each cluster represents the bodies in its region for interaction. For computing accelerations each body traverses the tree in depth-first manner starting at the root. For any internal node sufficiently far away, the effect of the subtree on the body is approximated by a two-body interaction between the body and a point mass located at the center-of-mass of the tree node. The tree traversal continues, but the subtree is bypassed. When the traversal reaches a leaf, a direct two body interaction is computed. The set of nodes which contribute to the acceleration on a body are called the *essential nodes* for the body. Each body has a distinct set of essential nodes which changes with time. Two important criterion help in approximating the force fields. Firstly, for a body far away from the cluster, the effect of the cluster can be approximated by its center of mass rather than by each individual interaction with each body in the cluster. Secondly, the depth-first traversal ensures that each body interacts only with the *largest* clusters for which the approximation is valid. Once accelerations on each body are known, the new positions and velocities are computed. The entire process is repeated for the desired number of time steps. The Barnes-Hut algorithm is shown in Figure 2.

For any particle p the force can be approximated by starting at the root cell of the tree. Let l is the length of the cell currently being processed and D the distance of p from the cell's center-of-mass. If $l \ / \ D < \theta$, where θ is a fixed accuracy parameter ~ 1, then the interaction between this cell and p is included in the total being accumulated. Otherwise, the cell is resolved into eight subcells, each one being recursively examined.

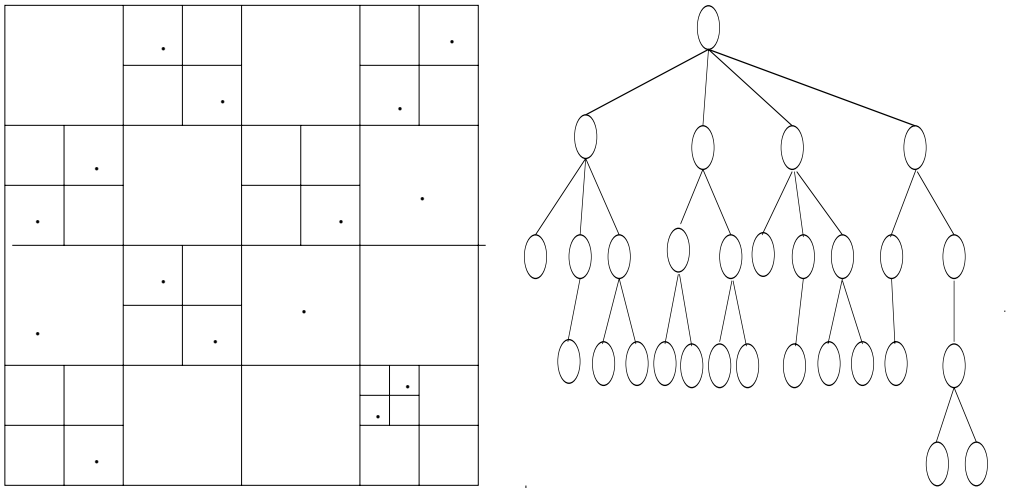

Figure 1: Creation of a BH-tree.

For each time step:

Step 1. Build the BH-tree.

Step 2. Compute centers-of-mass bottom-up.

For each body

Step 3. Start a depth-first traversal of the tree, truncating the search at a internal node where the approximation is applicable.

Step 4. Update the contribution of the node to the acceleration of the body

Step 5. Update the velocity and position of each body.

Figure 2: Barnes-Hut Algorithm

In this paper, we limit ourselves to the Barnes-Hut algorithm due to its popularity as well as its hierarchical nature. We provide a brief summary of the different issues required for its vectorization and parallelization on distributed memory machined with and without hardware support for shared memory.

3.1.1 Vectorization of Tree Traversals

Barnes [5] introduced a procedure to reduce the number of tree searches by establishing interaction lists for *groups* of particles which are close together in space. This shifts the bulk of the computation from tree descents to force summation, by minimizing the required number of tree traversals. A major regularity present in all hierarchical algorithms is that the representation of the gravitational field used to calculate the force on particle p is very similar to the representation used to compute the force on the nearby particle q. The expense of constructing the field expansion is thus shared between the particles within the cell. If there are too many particles in the cell then the cost of evaluating the local field by direct summation dominates and the cost of the multipole expansion will dominate if there are too few particles per cell. This is the strategy used in the Fast Multipole Method (FMM) which can be used to vectorize the Barnes-Hut algorithm. If n is a node on the interaction list L_p of p, and the distance between n and p is much greater than the distance between p and q, then most likely $n \in L_q$. An interaction list L_c is constructed which is guaranteed to satisfy the usual Barnes-Hut tolerance condition $l/D < \theta$ everywhere within the small cell c containing p and q and a few other particles. By reusing the same interaction list L_c for particles p and q to calculate force, the number of tree descents can be reduced by a factor equal to the number of cells in the cell c.

Makino [38] succeeded in vectorizing all aspects of the hierarchical tree algorithm by performing tree descents for many particles simultaneously, vectorizing loops over particles. A further gain can be realized by combining the Makino and Barnes procedures and performing tree searches for many *groups* simultaneously.

A minor disadvantage of both of these approaches is that they do not preserve the basic structure of the scalar version of the hierarchical tree method. The interaction lists are established for many particles simultaneously. This can be a hindrance if the force must be computed particle by particle, as is the case if each particle has a unique time step. The tolerance criterion is simultaneously applied to all relevant cells at a given level. Those cells at the current level satisfying the accuracy requirements are added to the list of interactions. The remainder are subdivided and their descendants are placed on the list of cells to be visited on the next level further down. Hernquist [23] proposed an algorithm to process the BH-tree *level* by *level* rather than *node* by *node* as in the scalar implementation. Vectorization is achieved by looping over all relevant nodes at the same level, simultaneously. We will restrict ourselves to the basic hierarchical approach for calculating forces and accelerations while discussing parallelization requirements. We discuss

efforts on shared memory machines briefly and move on to discuss solutions on distributed memory machines mainly because these machines are more readily available today commercially.

3.1.2 Parallelization on Shared memory machines

A parallel implementation of the Barnes-Hut algorithm on the DASH [14], a cache coherent shared memory multiprocessor shared memory machine, has been reported in [57] . The data partitioning is done using *costzones* [57] or by using Orthogonal Recursive Bisection (ORB). The data is globally shared among the processors in shared memory. The tree construction is parallelized by letting processors insert their particles into the shared tree concurrently. Whenever a processor has to modify the tree, either by putting a particle in a cell or by subdividing a cell, it must first obtain a lock on that cell to ensure that no other processor tries to modify it at the same time. As the number of processors increase, the overhead of executing these locks and unlocks becomes substantial. The ORB partition allocates every processor contiguous partition of space. This effectively divides the tree into distinct sections and assign a processor to each section, which means that there is little contention for locks after the first few levels of the tree are built, since processors will construct their own disjoint sections without much interference. Much of the contention is due to many processors simultaneously trying to update the upper levels of the tree.

In another approach outlined in [57], every processor builds its own version of the tree using only its own particles. These individual trees are then merged together into a single tree used in the rest of the computation. The number of times a processor has to obtain a lock on the global tree goes down as entire subtrees are usually merged into the global tree. There is a tradeoff when merging entire subtrees, since not much concurrency is available in the merging phase. However, results in [57] show that this method outperforms the previous one. This is essentially the approach one would take on a distributed memory by building local trees and then getting a global representation by combining them appropriately to store global information. The tree building and center-of-mass computation phases require both interprocessor communication and synchronization. The force calculation for a particle requires the communication of position and mass information from other particles and cells, but this is not modified during the force calculation phase. Forces on different particles can be computed in parallel without synchronization. Each processor will read the data it needs from the global tree, a single copy of which is shared among all processors. The work for a particle in the update phase is entirely local to that particle and requires neither communication nor synchronization.

3.1.3 Parallel Implementation on Distributed Memory Machines

On a distributed memory machine the data is distributed across processors. A data distribution that gives more or less equal work to all processors in desirable to obtain higher efficiencies. The

work metric used in the N-body simulation is the calculation of forces for each body. Hence, just the count of the number of bodies on a processor is not enough, their distribution in space also needs to be taken into account. This determines the force calculations that will be performed for each body and ideally we would like that to be perfectly load balanced. During the computation phase if data for force calculation is not available locally it must be fetched from the processor that has it. Data should be assigned to processors such that most data for computation should be available locally. Typically communication for fetching off-processor data is much more expensive than a local read. Data partitioning must preserve data locality to reduce communication requirements. An inappropriate data mapping can increase communication costs and degrade overall performance by adding to the overhead. A good partitioning would take both load balancing and data locality into account.

A distinguishing feature of the BH-tree is that it evolves continuously due to ongoing computation. It is a dynamic data structure and data must dynamically be updated to adapt to the evolving system. A static data mapping may not distribute data evenly after a period of time. The data mapping can either be done again at each time step or adjusted incrementally to reflect the changes in the system. Also, as bodies move and the distribution of bodies in space changes, the work associated with calculating forces can also change leading to differential load on processors. The mapping of bodies to processors must be adjusted to ensure load balance. Thus the BH-tree is adaptive to dynamic and irregular distribution of bodies. Also, the movement of bodies requires dynamic data mapping and distributed data management.

The communication pattern is irregular and dynamic. Since the BH-tree is distributed there is a need for off-processor data during computation. The data mapping can change from one step to another, the communication for such data will also change. So, one cannot calculate a static data pattern and optimize communication for it. It is unpredictable at compile time and hard to optimize. So finding a good communication schedule at the first iteration to be reused at later times does not work. A high-performance code can be developed by addressing the following issues

- Mapping of the BH-tree must change adaptively as the simulation proceeds.
- The set of tree nodes essential to a body can only be found by traversing the distributed tree. The cost of doing this with a dynamic tree can be prohibitive.
- The load balancing must take into account the work each body has to do rather than just the number of bodies on each processor.

In the following subsections, we describe the important approaches for data partitioning, tree construction, accessing non local data and tree traversals studied in the literature. See Table 1 for a summary.

For each time step:

Step 1. Partition the bodies to processors using ORB

On each processor :

Step 2. Obtain the *locally essential tree* for this processor

For each body in this processor

Step 3. Start a depth-first traversal of the tree truncating the search at a internal node where the approximation is applicable.

Step 4. Update the contribution of the node to the acceleration of the body

5. Update the velocity and position of each body belonging to this processor.

Figure 3: Parallel implementation of N-body algorithm using adaptive trees (Warren and Salmon, 1992)

	Phase	Implementation
1.	Tree Construction	Distributed Adaptive Trees (Warren and Salmon 1992) Hashed Octree (HOT) (Warren and Salmon 1993) Octree in shared memory (Singh, Hennessey and Gupta 1992)
2.	Data Partitioning	ORB tree for bodies. (Warren and Salmon 1992) Spatial coordinates to keys (Warren and Salmon 1993) Costzones (Singh, Hennessey and Gupta 1992)
3.	Tree Traversal	Latency hiding tree traversal (Warren and Salmon 1992)
4.	Locally essential data (Receiver-oriented) (Sender-oriented)	 Gather essential data for force computation (Warren and Salmon 1992) Send essential data to processor needing it (Liu P. 1994)
5.	Incremental Updates (Sender-oriented)	 Incremental Tree Updates (Liu P. 1994) Incremental updates of locally essential data (Liu P. 1994)

Table 1: Parallel approaches to N-body treecodes

3.1.4 Data Partitioning

The distributed tree representation is kept at each processor. Each processor owns portions of data, the amount of which is guided by the workload associated with it. Every body, on a processor, can only see a fraction of the complete tree in a distributed memory scenario. The distant parts are seen only at a coarse level of detail, while the nearby sections are seen all the way down to the leaves, observing that nearby bodies see similar trees. This means that the upper levels of the tree, have nodes that do not contain data but contain pointers to processors where the data can be found. Data partitioning can be done using orthogonal recursive bisection (ORB), where space is recursively divided in two, and half the processors are assigned to each domain until there is one processor associated with each rectangular domain. A multidimensional binary k-d tree, call it the ORB tree, can be used by alternating the dimensions of the split. A copy of the ORB tree is stored on every processor. Each internal node of the ORB tree represents a bisector plane and the domain it bisects, and each leaf is a processor domain. The ORB decomposition of space among processors allocates points in space to processors. This is useful for *sender-directed* communication of essential data used in relocating bodies which cross processor boundaries and for building the global BH tree. A owner of data can calculate which processors, if any, require its data by looking at the partitioning boundaries. This is in contrast with *receiver-directed* communication, where a data request is sent out and the appropriate processor(s) will fulfill it. ORB preserves data locality well and the cost of incremental load-balancing is negligible [36].

The load distribution changes only slowly across iterations and the ORB can be adjusted with minimum changes to balance the load again. The incremental update begins with each processor computing the total number of interactions used to update the state of the local bodies. A tree reduction yields the number of operations for the subset of processors corresponding to each internal node. A node is overloaded if its weight exceeds the average weight of the nodes at that level by some fixed quantity. A top-down search on ORB tree marks those internal nodes which are not overloaded but one of their children is overloaded. This node is called the *initiator*. Only the processors within the corresponding subtree participate in balancing the load for the region of space associated with the initiator. Since the subtrees for different initiators are disjoint, the non-overlapping regions can be balanced in parallel. The bodies of the overloaded child have to be moved to the non-overloaded child at each step. A new bisector plane is computed so that the right amount of workload can be shifted to the under-loaded child. This is done by determining the weight within the old plane and any given plane to find the correct bisecting plane by a binary search. The workload within the parallelepiped is computed by traversing the local BH-tree [36].

Another approach of partitioning data is by mapping the bodies by converting the locality information in terms of a one dimensional key. Each possible cell is identified with a key and by performing simple bit arithmetic on a key, keys for daughter or parent cells are determined. The translation of keys into memory locations where cell data is stored is achieved via hash table

For each processor DO in parallel

Step 1. Construct one dimensional keys for each body interleaving its spatial coordinates

Step 2. Sort the keys in an increasing order

Step 3. Divide the list onto processors weighted by the amount of work corresponding to each body

On each processor

Step 4. Construct the tree using the bodies local to the processor.

Step 5. Make copies of branches on every processor to complete the full representation of the tree

Step 6. Gather locally essential data needed for the bodies on this processor

For each body on the processor

Step 7. Traverse the tree representation in the local memory

Figure 4: Hashed Oct Tree (HOT) implementation (Warren and Salmon,1993)

lookup. This scheme provides a uniform addressing mechanism to retrieve data which is in another processor. This representation of data is called the *Hashed Octree* method [61]. The key is defined as a result of a map of d floating point numbers, body coordinates in d-dimensional space, into a single set of bits. The floating point numbers are converted into integers and then bits of the d integers are interleaved into a single key. This is identical to Morton ordering (also called Z or N ordering). This function maps each body in the system to a unique key. A hash table is used to map the key to the memory location holding this data. A hashing function maps the k-bit key to the h-bit long address. Collisions in the hash table can be resolved by chaining. During tree traversals, daughter nodes are found by shifting the parent key left by d bits and the result is OR'ed to daughter numbers 0 to $2^h - 1$. The key provides immediate *O(1)* access to any object of the tree. Access to data can be generalized to a global accessing scheme implementable on a message passing architecture. By taking advantage of the properties of mapping spatial coordinates to keys, a sorted list of one-dimensional body key coordinates are divided into equal pieces, weighted by the amount of work corresponding to each body. The work for each body is readily approximated by counting the number of interactions the body was involved in on the previous timestep. Using Morton ordered decomposition a processor domain can span one of the spatial discontinuity. *Peano-Hilbert* ordering can be used for domain decomposition, which does not contain any spatial discontinuity, but is harder to describe by body coordinates.

The data distribution described above, leads to a mapping with irregular boundaries. Incremental modification of the tree structures, the HOT tree and the locally essential tree, becomes increasingly difficult in such a case. Since boundaries are no longer uniform, the movement of bodies from one partition to another cannot be ascertained purely by looking at spatial coordinates. Hence, a sender initiated protocol is not suitable for such a scheme.

3.1.5 Tree Construction

Once the bodies are allocated to a processor using the ORB data partitioning, the adaptive BH-tree is constructed by building local trees on every processor using the local bodies only. A local tree is built with respect to the entire spatial domain and not its own processor domain. It represents the local view of the distribution of bodies within a processor domain. The local trees might not be structurally consistent with respect to each other. Local trees are made structurally consistent by adjusting the levels of all leaf nodes which are split by ORB bisector planes. For a tree with N bodies, each leaf in the BH-tree can contain upto L bodies, $L << N$, to adapt to the fast multipole expansion. This accelerates force calculations by reducing the number of tree traversals, but makes level adjustment more tricky.

Updating the BH-tree incrementally by adjusting the levels after each insertion/deletion within the localtree is described in [36]. Consider a body λ in level ℓ of a local tree consisting of n levels ($n >= l$). λ may actually be in a deeper level $\ell' > \ell$ in the global tree. The level of the cell needs to be adjusted down to ℓ' for the owners on the path from ℓ' to ℓ to receive contributions from λ. Only bodies that are not in the correct levels need to be adjusted. If the processor domain covers the entire leaf, consisting of atmost L bodies, the bodies within the leaf do not require adjustment. Otherwise the leaf's domain spans multiple processors. For each such leaf, ρ, define *covering processors* $\mathcal{CP}(\rho)$ as those processors whose domains overlap with ρ. The level of a body then is determined only by its covering processors. Initially the level information of a body λ contains its split leaf and the local bodies in it. The correct level information is found by refining this information further. Each body λ receives level information from each member of $\mathcal{CP}(\rho)$. Then each processor chooses the deeper leaf as the new level.

Once level adjustment is done, each processor computes the center-of-mass and multipole moments of its local tree. Next, each processor sends its contribution to an internal node to the owner of the node. The transmitted nodes are combined by the receiving processors completing the construction of the global BH-tree. At this stage each processor contains a tree with nodes containing bodies it owns and some empty nodes at upper levels to identify all the bodies on other processors. The owner of a tree node is the processor that contains its geometric center. An owner of a node keeps track of the forces and the acceleration of the bodies in it. This mapping function can be easily constructed by different processors consistently using the ORB tree. A local tree node has correct node information if it is completely covered by one processor domain.

During a time step a body may move into another processor domain and has to be moved from one local tree to another. Also, the body may remain in the same processor but move into a new tree node. If there is no body in the leaf then the leaf has to be deleted. If the body joins a leaf with already L bodies inside, it must be divided until no more than L bodies are in any new leaf.

3.1.6 Accessing Locally *Essential* Non Local Data

The ORB has a recursive structure, consisting of $log_2 P$ levels and $P-1$ spatial bisectors to divide spatial data into P partitions. For each processor a top-down BH-tree traversal collects the data that is needed for calculating accelerations for the bodies it owns. This is referred to as *locally essential* data, some of which might be owned by other processors. This data can be acquired in two ways. In the first method, a processors sends a request for data to the owning processor. The owners of data then fulfill the request by sending the requested data. This can either be demand driven, that is, essential data is requested on a need basis during force computation, or a communication phase before the start of force calculations can accumulate all the essential data. This scheme has the advantage that data partitioning schemes that preserve locality, but result in uneven boundaries, can be used. The disadvantages are that the overheads of fine-grained communications are high, which have been addressed by using multi-threaded tree traversals at the expense of considerable complexity [61].

The second method uses a one way message transfer by doing away with requests for data. Each processor exploits the ORB partitioning information, to calculate the processors which require its local data as essential data. A *sender-directed* communication mechanism is used in which every processor identifies its own exportable data, and then exchanges that data with a processor in the complimentary partition on the other side of the bisector. After $log_2 P$ exchanges every processor is in possession of its locally essential tree. . For calculating locally essential trees, the owner of a tree node sends information to a *influence ring*, the possible positions of bodies that the tree node is essential to. Let x be a tree node and let y be its parent. Let B_x be the region within which the approximation cannot be applied on x. B_y is similarly defined for y. The bodies outside B_y should apply approximation on y instead. The bodies within B_x cannot apply approximation on y either. Thus x is essential to only to those bodies within the annular region $B_y - B_x$. The owner of a tree node sends information only to processors whose domains overlap with the influence ring. The destination set is calculated from the influence ring and the ORB tree.

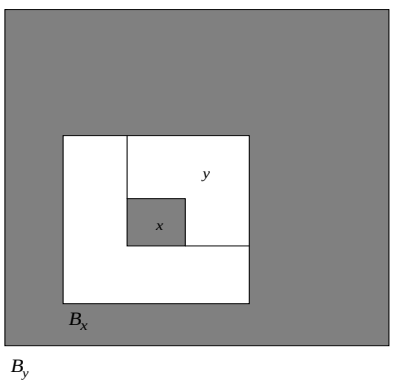

Figure 5: Influence ring of x

3.1.7 Tree Traversal

Tree traversals are needed in the force calculation phase. By calculating forces on bodies in groups the cost of tree traversal can be amortized over several bodies. The key point here being, nearby bodies on a node will see similar tree structures and will traverse similar paths [5]. A multipole acceptability criteria (MAC) is used to approximate the far-field forces. This determines the depth of the tree traversals for each body or a group of bodies, if that is the case. An additional multipole acceptability criteria (MAC) is the evaluation of the force at any point in a processor's domain. This can be controlled by the distance from the edge of the cell to the boundary of the rectangular domain. The positions of the bodies can be updated by traversing the locally essential trees. Nodes which do not have global information can safely be skipped since that cannot be essential data. There is no communication required in this phase. By calculating accelerations on groups of bodies the vector units in machines can be utilized effectively. There is a reduction in time spent in traversing the tree although the number of calculations increase [5]. A further reduction in tree traversal can be obtained by caching essential nodes. The key observation is that the set of essential nodes for two distinct groups close together in space are likely to have many elements in common, so the cache can be utilized effectively.

A *walk-list* of cell nodes is maintained, which on the first pass contains only the root cell. Each daughter cell of the input walk list nodes is tested against the MAC. If it passes then the corresponding cell data is placed on the *interaction list*. If a daughter cell fails the MAC, it is placed on the output walk list. After the entire input list is processed the output walk list is copied to the walk list and the process iterates. The process continues till there are nodes in the walk list. After this the calculations in the interaction list are processed.

The advantage of this new method is offset by losing the advantage of sender-directed communication. It is difficult for the BH node to compute the set of processors where its data is essential. This is due to the processor domains having complicated shapes because partitioning is now done according to the bodies position in the BH-tree. Also, the force computation stage is slowed down by fine-grained communication. The startup cost for a large number of small sized messages is high on current architectures. This is overcome by using multiple threads to pipeline tree traversals and to update accelerations. If a body does not obtain an essential node in its local tree it initiates the communication to get the data and continues with the tree traversal on some other part of the BH-tree. The communication throughput is increased by packing requests/data to/from the same processor into longer messages, at the expense of added complexity in code.

The MAC used above is difficult to calculate because the parallel algorithm requires identification of locally essential data before the tree traversal begins. With a data-dependent MAC it is difficult to determine before hand which non-local cells are required before the traversal begins. A different approach that does not require building the locally essential trees was proposed by Salmon [61]. It provides a mechanism to retrieve non-local data as it is needed during the tree traversal.

For each processor DO in parallel:
    Step 1. Build local BH-trees.
    For every time step do:
        Step 2. Construct the BH-tree representation
            Step 2a. Adjust node levels
            Step 2b. Compute partial node values on local trees
            Step 2c. Combine partial node values at owning processors
        Step 3. Owners send essential data
        Step 4. Calculate accelerations
        Step 5. Update velocities and positions of bodies
        Step 6. Update local BH-trees incrementally
        If workload is not balanced
                Step 7. update the ORB incrementally

Figure 6: A generic implementation of the N-body algorithm using incremental data structures

Calculating accelerations is the most time consuming step. Building locally essential trees lets each processor obtain the data it requires for calculating accelerations on the bodies owned by the processor.

3.1.8 Summary

We summarize the phases of computation and communication for the N-body simulation and global operations and collective communication requirements that it needs on a distributed memory MIMD machine.

- *Data partitioning:* Data partitioning can be done using Orthogonal Recursive Bisection. This ensures regular spatial boundaries which can be used for incremental tree updates. However, it is not so good in preserving spatial locality. A space filling approach, like the Morton ordering or the Peano-Hilbert ordering provide good data locality properties. They lead to uneven processor boundaries and incremental aspects become hard to implement.
- *Tree construction:* A tree data structure is employed to apply the multipole approximations for force calculations. A globally consistent tree is maintained by each processor for the bodies it owns, and pointers to locate bodies that are owned by other processors. A octree (in 3D) called the BH-tree is maintained. A *many-to-many communication* phase is needed to make this tree structurally consistent.
- *Incremental tree updates:* As the simulation proceeds and the new positions of bodies are evaluated, the ORB partitioning and the BH-tree need to be updated to reflect the changes.

This may involve movement of bodies across processor boundaries. In one case a tree needs *delete* a body and in the other an *insert* operation is needed. The BH-tree needs to be kept balanced for efficient tree traversals. The ORB tree partitions also need to be adjusted incrementally, otherwise the ORB tree would need to be constructed from scratch at every iteration.

- *Gathering essential data:* As the BH-tree is traversed top-down, bodies in some nodes are not available in the local memory. If such a case arises, the body data needs to be fetched from the appropriate processor, information of which is available at the node. This can either be done during the force calculation phase or in a pre-fetching phase, where all data is gathered beforehand and plugged appropriately in the local BH-tree. However, if the MAC is data-dependent, as in the case of [61], then essential data cannot be ascertained before force calculations begin. This phase requires several *many-to-many* communication steps.

- *Tree walking:* A node is opened if it fails the MAC criteria. Then each of the daughter cells is traversed recursively. This process starts from the root of the tree. It is interrupted when a node is reached which does not have data in local memory, but has pointers to other processors. Multiple threads have been used to overlap tree walking with communication delays [61].

- *Load balancing:* This issue can be addressed by adjusting the ORB partitioners incrementally. Otherwise, the ORB tree needs to be constructed from scratch at every iteration. Some amount of load-imbalance can be tolerated depending on the cost of such an operation. Space filling curves can be remapped to maintain load balance.

3.2 Molecular Dynamics

Molecular Dynamics is a widely used technique for the studying liquids, solids, complex molecular systems in Chemistry, Biology, Statistical Physics and Materials Science. Molecular Dynamics simulates the local and global motion of atoms, molecules or some larger unit, by integrating Newtonian equations for a system of N particles. It is a computationally intensive problem and parallel computers have been used to simulate larger and more realistic systems.

Let us consider a system with N particles represented by a collection of positions and velocities $\vec{x_1}, \vec{x_2}, ..., \vec{x_N}$ and $\vec{v_1}, \vec{v_2}, ..., \vec{v_N}$ respectively. Let r_{ij} be the distance between particle i and particle j. The total energy of the system is given by $E = \sum_i \sum_j U(r_{ij})$, where $U(r_{ij})$ denotes the inter-particle potential between particles i and j. Lennard-Jones potential is the most widely used potential in modelling liquid behavior, and is given by the equation

$$U(r_{ij}) = 4\epsilon[(\frac{\sigma}{r_{ij}})^{12} - (\frac{\sigma}{r_{ij}})^{6}],$$

where σ is the length at which the potential crosses zero and ϵ defines the energy scale. A cutoff distance r_c can be defined beyond which the potential is very small and can be taken to be effectively zero. The force is given by the gradient of the potential, $\vec{F_{ij}} = -\nabla U(r_{ij})$ and $F_{ij} = -F_{ji}$. From the forces the acceleration is calculated by using the equation $\vec{a_{ij}} = \frac{F_{ij}}{m}\vec{r_{ij}}$. By integrating the equations of motion , the new set of coordinates and velocities can be found for time $t + \delta t$ from the values at time t.

The computational tasks in molecular dynamics are to find the interacting neighbors of a particle (particles at a distance $\leq r_c$), compute the forces and integrate the equations of motion and this cycle continues for the period of the simulation.

In a system of N particles, each particle interacts with $\frac{4}{3}\pi {r_c}^3 \rho$ particles, where r_c is the cutoff distance and ρ is the average particle density. For each particle, a search for this neighbors is done which is a $O(N^2)$. For a rapidly decaying potential, like the Lennard-Jones, the interations greater than a distance r_c do not contribute and can be ignored. This reduces the search such that for a particle i the search would be done in the neighborhood $\aleph_i = j||r_i - r_j| < r_c$. The physical domain is divided into rectangular domains, called cells, of size $\sim r_c$ so that the search can be restricted to the neighboring cells.

N-body methods have been applied to molecular dynamics by using a hierarchical spatial octree to represent data. A list of interacting neighbors is maintained for each particle. This method is called the *Verlet neighbor table method* and a list of interacting pairs for which the interaction forces have to be computed is used to save computation.

The overhead of constructing the table is significant and should be amortized by making use of the table foe several time steps. The table can be reused if no pair of particles originally apart at a distance r_s comes closer than r_c, where $r_s = r_c + \delta_s$, δ_s being a safety distance. The table can be constructed by using a "cell" method in $O(N)$ time for N particles.

3.2.1 Parallelization on distributed memory machines

CHARMM [10] is a program which calculates empirical energy functions to model macro-molecular systems. Data decomposition and force decomposition methods are employed for parallelization of CHARMM, a molecular dynamics software, on MIMD machines in [51]. A regular force decomposition algorithm partitions atoms evenly over processors. However, the non-bonded interaction list is distributed unevenly and results in severe load imbalance. The nature of force interactions is irregular and hence any parallelization needs to take that in account. A hierarchical nature is embedded in the way force computations are performed. Bonded interactions occur only between atoms in close proximity to each other and non-bonded interactions are excluded beyond a certain cutoff range. To preserve locality and reduce communication it is reasonable to assign atoms that are close to each other on the same processor. The amount of computation associated with an atom

depends on the number of atoms with which it interacts.

A parallel cell method and a parallel verlet neighbor table method is described in [39] Processors must synchronize after the completion of the distributed force calculation before any processor can begin the update of the particle coordinates. This requires the loads to be uniformly balanced to reduce processor idle-time. Force calculations can proceed without synchronization on a MIMD platform. For calculating forces, each pair of interacting particles must have their mutual force calculated by bringing together the particles on a single CPU. Each processor is mapped a rectangular volume of space. Particles in the cells of this region are allocated to processors. Each particle is owned by some processor, which is responible for its force calculations. The particles at the boundary region of the processor may be needed by the neigboring processors. An extended space can be defined for each processor (as shown in Figure 7). Once the neighbor table is constructed for an iteration, a time step consists of a communication step, a force compuation step and an integration of the equations of motion. The communication step is for the particles at the node boundaries to be sent to the neighboring processors. Another communication step is needed for updating the neighbor table to reflect the movement of particles.

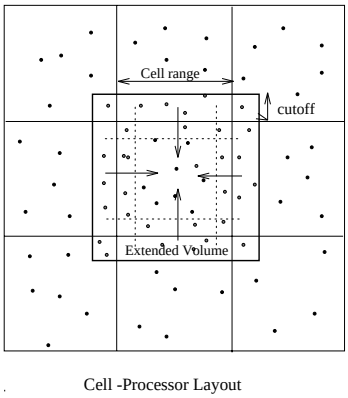

Figure 7: Cell processor layout showing the extended volume for particle interactions

3.2.2 Summary

The issues in parallelization of this application for our framework of hierarchical applications are summarized in this section.

- *Data Partitioning:* Equal work should be allocated to all processors. The measure of work is the number of interactions by all the particles on a processor. Particles or cells can be mapped to processors. The amount of work per cell is distribution dependent and hence data partitioning is an issue.
- *Tree construction:*Orthogonal recursive bisection can be used to construct a k-d tree. Particles are spatially distributed and we need to create the neighbor interaction list, which can be done

by tree traversals. Since the interactions are limited to neighboring cells, the tree traversals will be localized in a region, to access parent and sibling nodes.

- *Incremental Updates:* The neighbor table needs to be updated as particles change position as the simulation progresses. Particles are deleted from and inserted into the appropriate partitions. Doing this incrementally is more efficient than constructing the neigbor list from scratch.

- *Queries:* The queries are of the type where a nighbor list for all particles at a distance r_c need to be identified. These queries are spherical and can be approximated by using a rectangle query, leading to some duplicated effort. Communication is generated to acquire the appropriate data.

3.3 Volume Rendering

Volume rendering is a technique to visualize three dimensional volume data by projecting it onto a two dimensional plane.. It is used in diverse fields like medical imaging, modeling physical phenomenon and molecular structures. Most of these require generating multiple views of the volumes at different orientations from the viewer. Due to requirements for it to perform in real-time, parallel algorithms have been proposed and implemented to accelerate volume rendering.

Volume rendering is a technique for visualizing sampled scalar or vector fields of three spatial dimensions without fitting geometric primitives to the data. Images are generated by computing 2-D projections of a semitransparent volume, where the color and opacity at each point are derived from data using local operators. Since all voxels (volume elements) participate in the generation of each image, rendering time grows linearly with the size of the data set. The principal advantages of these techniques over others are their superior image quality and the ability to generate images without explicitly defining surface geometry. This gives the images a degree of visual realism.

Rays are cast from every pixel in the image plane into the volume and the data is resampled at regular intervals along each ray. At each sample point, the eight data values are trilinearly interpolated to provide a value and gradient that corresponds to the sample's location. Interpolation os necessary since the volume slice samplings may not coincide with the point the ray is driven through. This value is then classified to give equivalent color and opacity values. The color is shaded by calculating the dot product of the local gradient with each light source which is composited to the ray. Data volumes with contiguous subregions of voxels classified as having zero opacity values, these do not contribute to the final image and their resampling is unnecessary.

The algorithm described in [33] assumes a scalar-valued array forming a cube that is N voxels on a side. Voxels are indexed by a vector $\mathbf{i} = (i, j, k)$ where $i, j, k = 1, ..., N$, and the value of voxel $\mathbf{i}$ is denoted by $f(\mathbf{i})$. Using local operators, a scalar or vector color $C(\mathbf{i})$ and an opacity $\alpha(\mathbf{i})$ are derived

for each voxel. Parallel rays are then traced into the data from an observer position. It is assumed that the image is a square measuring P pixels on a side and that one ray is cast per pixel. Pixels, and hence, rays are indexed by a vector $\mathbf{u} = (u, v)$ where $u, v = 1, ..., P$. For each ray, a vector of colors and opacities is computed by resampling the data at W evenly spaced locations along the ray and by trilinearly interpolating from the colors and opacities in the eight voxels surrounding each sample location. Samples are indexed by a vector $\mathbf{U} = (u, v, w)$ where (u, v) identifies the ray and $w = 1, ..., W$ corresponds to the distance along the ray with $w = 1$ being closest to the eye. The color and opacity of sample $\mathbf{U}$ are denoted $C(\mathbf{U})$ and $\alpha(\mathbf{U})$, respectively. Finally, a fully opaque background is draped behind the dataset, and the resampled colors and opacities are composited with each other and with the background to yield a color for the ray. This color is denoted by $C(\mathbf{u})$. Working front to back color and opacity are composited at each sample location *under* the ray. Specifically, the color $C_{out}(\mathbf{u}; \mathbf{U})$ and opacity $\alpha_{out}(\mathbf{u}; \mathbf{U})$ of ray $\mathbf{u}$ after processing sample $\mathbf{U}$ are related to the color $C_z(\mathbf{u}; \mathbf{U})$ and opacity $\alpha_{in}(\mathbf{u}; \mathbf{U})$ of the ray before processing the sample and color $C(\mathbf{U})$ and opacity $\alpha(\mathbf{U})$ of the sample by the transparency formula

$$C_{out}(\mathbf{u}; \mathbf{U})\alpha_{out}(\mathbf{u}; \mathbf{U}) = C_{in}(\mathbf{u}; \mathbf{U})\alpha_{in}(\mathbf{u}; \mathbf{U}) + C(\mathbf{U})\alpha(\mathbf{U})(1 - \alpha_{in}(\mathbf{u}; \mathbf{U}))$$

and

$$\alpha_{out}(\mathbf{u}; \mathbf{U}) = \alpha_{in}(\mathbf{u}; \mathbf{U}) + \alpha(\mathbf{U})(1 - \alpha_{in}(\mathbf{u}; \mathbf{U}))$$

Many datasets contain coherent regions of empty voxels. A voxel is defined as empty if its opacity is zero. These do not change the opacity of the ray and need not need be rendered. An optimization to improve performance is to ignore empty voxels while rendering. Methods for encoding coherence in volume data include octree hierarchical spatial enumeration, polygonal representation of bounding surfaces and octree representation of bounding surfaces. The second optimization is based on the observation that, once a ray has struck an opaque object or has progressed a sufficient distance through a semitransparent object, opacity accumulates to a level where the color of the ray stabilizes and ray tracing can be terminated. Adaptive termination is implemented by stopping each ray when its opacity reaches a user-selected threshold level.

In this section we will concentrate on methods that use hierarchical spatial enumeration. An octree is used to represent voxel data which helps in skipping empty regions of the data set by appropriate tree traversals[34]. For a dataset measuring N voxels on a side where $N = 2^M + 1$ for some integer M, the hierarchical spatial enumeration can be represented by a pyramid of $M + 1$ binary volumes. Volumes in this pyramid are indexed by a level number m where $m = 0, ..., M$, and the volume at level m is denoted by V_m. Volume V_0 measures $N - 1$ cells to a side, volume V_1 measures $(N - 1)/2$ cells on a side, and so on upto volume V_m which is a single cell. Cells are indexed by a level number m and a vector $\mathbf{i} = (i,j,k)$ where $i, j, k = 1, ..., N - 1$, and the value contained in cell $\mathbf{i}$ on level m is denoted $V_m(\mathbf{i})$. The ray-tracing, resampling and compositing steps now use this pyramidal data structure.

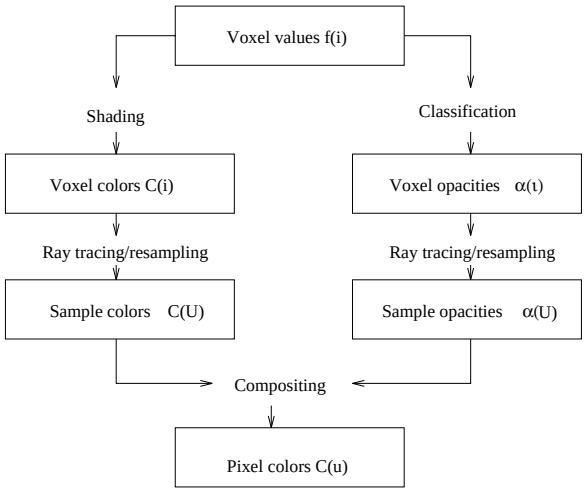

Figure 8: Overview of volume-rendering algorithm

For each ray, the point where the ray enters the single cell at the top level is calculated. The pyramid is then traversed in the following manner: After entering a cell, its value is tested. If it contains a zero, we advance along the ray to the next cell on the same level. If the parent of the new cell differs from the parent of the old cell, we move up to the parent of the new cell. This is done since if the parent of the new cell is unoccupied we can advance the ray further on the next iteration than if we had remained at the lower level. If, however, the cell being tested contains a one, we move down one level, entering whichever cell encloses our current location. At the lowest level, samples are drawn at evenly spaced locations along that portion of the ray falling within the cell, resample the data at these sample locations, and composite the resulting color and opacity into the color an opacity of the ray.

Adaptive termination of ray tracing is done by quickly identifying the last sample location along a ray that significantly changes the color of the ray i.e if $C_{out}(\mathbf{u};\mathbf{U}) - C_{in}(\mathbf{u};\mathbf{U}) > \epsilon$, for some small $\epsilon > 0$. Since $\alpha_{in}(\mathbf{u};\mathbf{U})$ increases monotonically along the ray, no significant color change occurs beyond the point where $\alpha_{out}(\mathbf{u};\mathbf{U})$ first exceeds $1 - \epsilon$. Higher value of ϵ reduce rendering time, while lower values reduce image artifacts.

An algorithm in which image quality is adaptively refined over time is presented in [35]. An initial image is generated by casting a uniform but sparse grid of rays into the volume data, less than one ray per pixel, and interpolating between resulting colors and resampling at the display resolution. Subsequent images are generated by alternately casting more rays and interpolating. Rays are distributed according to measures of local image complexity. Recursive subdivision based on color differences is used to concentrate these rays in regions of high image complexity, and recursive bi-linear interpolation is used to form images from the resulting non-uniform array of colors.

Figure 10 shows in two dimensions how a typical ray might traverse a hierarchical enumeration. The level-zero cell corresponding to each nonempty voxel is denoted by a shaded box. The largest empty cell enclosing each empty voxel is denoted by an unshaded box. The sequence of points where the ray enters the next cell at the same level is denoted by circular dots. In regions containing many nonempty level-zero cells, the spacing between these dots is close to the spacing between voxels. These points are not evenly spaced on the ray. If the data is resampled at nonuniformly spaced points, a noise component may be added to the resulting image. To avoid this, a set of evenly spaced sample locations is superimposed, shown as dividing lines in Figure 10, and limit to resampling the data at these locations.

Recently a shear-warped algorithm has been reported in [30]. It is currently acknowledged to be the fastest sequential volume rendering algorithm. It is a modification of the object space technique discussed above. Considering a $N \times N$ pixel viewing plane and an $N \times N \times N$ voxel dataset, we observe that $\theta(N^2)$ rays are driven through the N slices of the volume (the volume is N slices of $N \times N$ voxels). A total of N^3 interpolations and $k \times N^3$ resampling weights are computed for each iteration, where k is the number of weights that need to be computed for each iteration. Shear-warp method reduces the resampling calculations to N, by shearing the volume such that each ray can be assumed perpendicular to the slices. Each slice can then be translated and resampled using weights which are invariant across the slices. However, this genrates an intermediate image which then needs to be warped to produce the final image. For each pixel in the final image, the four nearest neighbors in the intermediate image are located and the final value of the pixel is interpolated from the color value of these neighbors. This requires $\theta(N^2)$ computation. Using early ray termination, skipping runs of transparent voxels by using run-length encodings, this technique has been improved further.

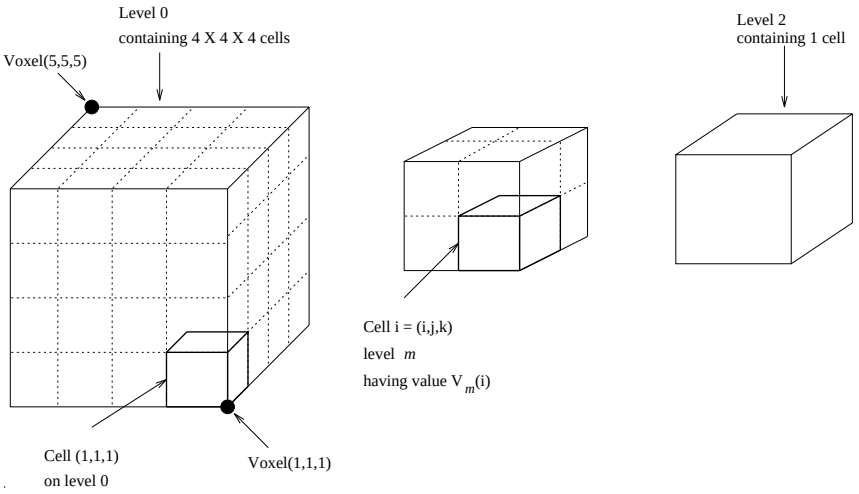

Figure 9: Hierarchical enumeration of object space for N = 5

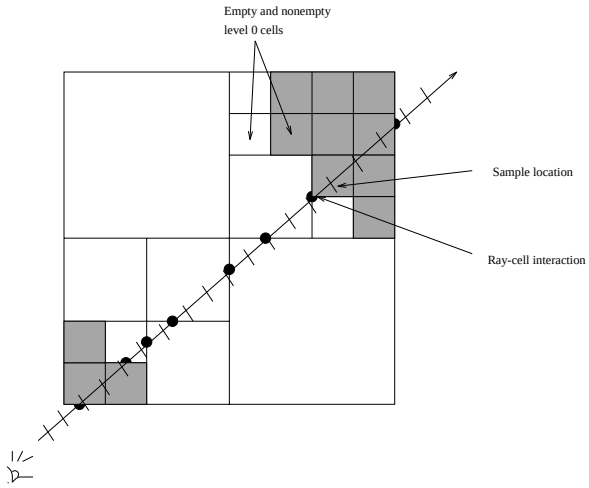

Figure 10: Ray tracing of hierarchical enumeration

3.3.1 Parallelization on Shared Memory Machines

An implementation of ray traced parallel volume rendering using the single address space distributed memory has been done on the Stanford DASH [42]. The data volume is partitioned and distributed to the local memories of the multiprocessor nodes. The image space is statically divided and allocated in equal areas to the processors. Rays are cast by each processor from the subimages that they are responsible for. Voxels required at the resampling points along each ray are accessed directly if available in local memory, or are fetched from the remote nodes where they reside. Dynamic load balancing is realized by allowing idle processors to grab images from others that still have work to do. The algorithm uses an octree representation, adaptive image sampling and early ray termination as optimizations. The performance of the algorithm depends on data coherence. Adjacent rays traversing through the same volume will tend to access the same voxel. Communication overheads will diminish at each processor if this fact is utilized in the algorithm. Though good cache performance is realized in [42], no analysis is presented for expecting such behavior. We do not discuss other efforts on shared memory machines, since out primary focus is on distributed memory machines.

3.3.2 Parallelization on Distributed Memory Machines

Parallel data distributed volume rendering was first done on an NCUBE by Montani *et al.*, dividing the volume in slices along one dimension of the volume and using a static load balancing scheme to redistribute the data among processors [41]. Processing nodes are organized as clusters and the image space is partitioned as to assign a subset of pixels to each cluster. A hybrid strategy to ray tracing parallelization is applied, using ray-dataflow within an image partitioning approach.

A ray tracing volume renderer that samples data volumes decomposed and distributed over the local memories of the parallel nodes is implemented by Karia [27]. Every node renders and creates a partial image of the projection of each of the subvolumes it holds. Subsequently, all nodes communicate and composite their partial images in a divide and conquer way to generate the final image. The rendering stage, which requires the bulk of the computation, does not require any communication. Communication is only required in the final compositing stage. Prior to rendering every processor classifies the data stored locally by mapping each voxel's value to its respective color and opacity. After data distribution, certain processors may hold subvolumes that are empty, and avoid rendering them altogether. This creates an imbalance among the nodes in the amount of useful rendering being done for the volume.

Green and Paddon [65] have discussed methods of exploiting coherence of references to entries in the object space which use a combination of dynamic and static caching techniques. A frequency distribution of object usage is determined by measuring the frequency of reference of objects while processing different rays. They have advocated the use of a software cache on a processor to exploit the *locality of reference* that arises from consecutive memory references to similar addresses in main memory.

The colors and opacities computed at each sampling point along a ray are composited using the **over** operator. For any two sample points S_i and S_j, whose colors and opacities are respectively $[c_i, \alpha_i]$ and $[c_j, \alpha_j]$, their composition using the over operator is defined as $S_i \quad \textbf{over} \quad S_j = S_i + (1 - \alpha_i)S_j$

The *over* operator is associative [28]. Hence, its application to any sequence of samples $S_i ... S_n$ may be grouped arbitrarily as follows

$$\begin{aligned} & (S_1 \quad \textbf{over} \quad S_2 ... S_i) \\ \textbf{over}\ & (S_{i+1} \quad \textbf{over} \quad S_{i+2} ... S_j) \\ \textbf{over}\ & ... \\ \textbf{over}\ & (S_{k+1} \quad \textbf{over} \quad S_{k+2} ... S_n) \end{aligned}$$

Table 2 summarizes the various approaches that have been reported in the literature for volume rendering on distributed memory machines.

3.3.3 Data Partitioning and Coherence Issues

Parallelization strategies for volume rendering have two goals. Each processor needs to be assigned equal load and any mapping of data to processors needs to maintain locality. The former helps to reduce processor idle time and the latter helps in keeping overheads of communication low. These are referred to as *load balancing* and maintaining *data locality*, two often conflicting goals. We

Approach	Target Architecture	Description
Montani et al. (1992)	nCUBE	Hybrid image partitioning - ray dataflow approach. Processing nodes organized as a set of *clusters*. Image space is partitioned, Volume data is replicated on each cluster. Static load balancing is used for distributing data.
Nieh (1992)	Stanford DASH	Data interleaved among processor memories. Image partitioned into contiguous blocks for assignment to processors. Task Stealing is used for dynamic load balancing.
Schröder & Stoll (1992) Vezina et al. (1992)	CM-2 MP-1	Data parallel SIMD implementation, rays proceed in lock step Also SIMD with volume transposition to localize data access
Ma, Painter et al. (1993)	CM-5	Static input data partitioning into subvolumes using a k-D tree Processing nodes perform local raytracing of their subvolume concurrently.
Karia (1994)	Fujitsu AP1000	Data is decomposed into subvolumes and rendered locally on each processor Scattered decomposition is used for load balancing.

Table 2: Different approaches on parallel volume rendering

discuss each of these to motivate the approach we have taken to analyze requirements for each of the two goals.

Strategies used for data partitioning are classified as follows.

Image Space Partitioning The pixels of an image are distributed across processors. Each processor traces rays for the pixels assigned to it. The volume data is replicated on each processor. Portions of the image from each processor are then combined to yield the final images. This method achieves near linear speedup but is not feasible if the object data set is larger than the available memory on each node.

Object Space Partitioning The volume data is partitioned and distributed among processors. Each processor traces each ray in the local partition only. Each non-resolved ray is transmitted to the next processor for further tracing. Once each ray has finished, the final composited values are collected to form the final image.

Object Dataflow A partition of the image is assigned to each processor, which locally traces and resolves each assigned ray. Volume data is partitioned among nodes too. Non-resolved rays will be sent to appropriate processors for tracing and the "owner" of the ray will get the finished result back.

Image/Object Partitioning The volume data is partitioned among processors. The image data is also partitioned among processors. Each processor is responsible to trace rays from pixels assigned to it. Pixels may be traced in the local volume data that is in the processors memory or it might fetch data that it needs from other processors.

Communication costs are typically higher than computation costs on most real machines. To

keep communication costs down, various forms of data locality need to be exploited. There are three kinds of coherence in images which we discuss next.

Image Coherence Image coherence is the property that adjacent pixels of an image are illuminated in a similar way. Portions of the image are similar in nearby areas, and this fact can be exploited when allocating pixels to processors. Nearby pixels go to same processors. This coherence exists in two dimensions. Exploiting this property for load balancing is not as straight forward. In an irregular image, where some portions are bright and some are dark, the work done in compositing a ray can vary a lot. Any load distribution strategy should take that into account.

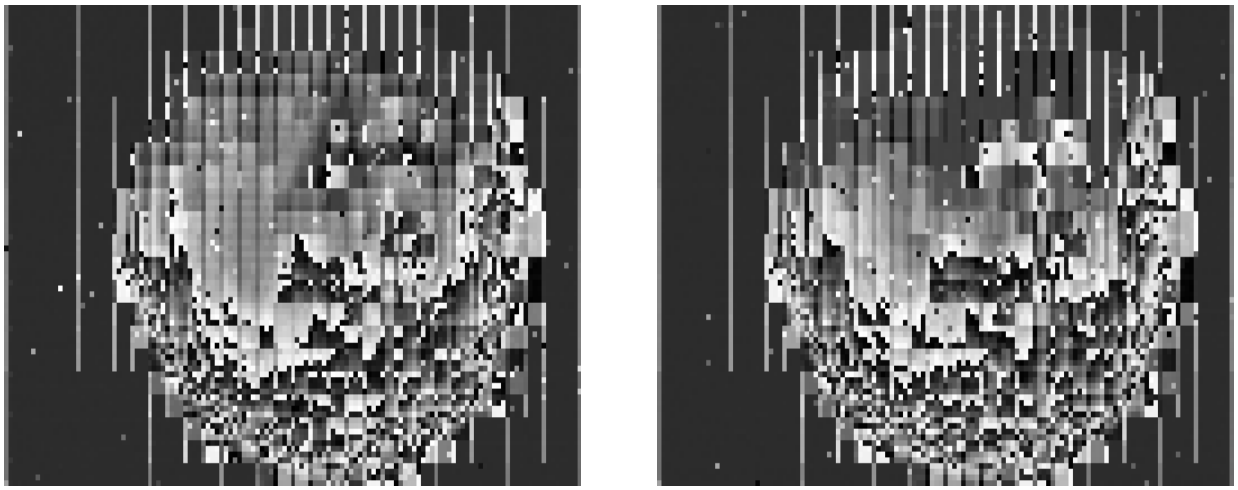

Figure 11: Frame 1: Ray work profile for **brainsmall** and Frame 2 at a 5^o rotation - Each pixel shows the computation work of a scanline. Scanlines for a slice progress from left to right on the horizontal axis. Slices of a frame are top to bottom on the vertical axis.

Object Coherence Adjacent rays will travel similar portions of the object. This is the essential idea of object coherence. In hierarchical volume representations, ray intersections with the octree can be optimized for adjacent rays (explained in a later section). This helps in reducing communication costs for essential volume data on processors for ray tracing by allowing reuse of offprocessor data. The data can be incrementally modified as the previous data can usually be reused. Another kind of coherence is available as the rays traverse the slices of the volume. The volume data gradually changes as the ray traverses through each slice. From one slice to another *slice coherence* is present as seen in Figure 11.

Frame Coherence Consecutive frames in a multiframe sequence are quite similar. Figure 11 shows frame coherence for the *brain* dataset. This fact is used in the dynamic load balancing strategy that we have proposed in [20]. Workload information from previous frame can be used to predict the load characteristics of the current frame.

Data partitioning for the image space needs to provide image coherence. By proceeding scanline by scanline within a slice, scanline coherence is exploited. To preserve this, image partitioning is done by dividing the scanlines to processors, keeping the load balanced. A record of the load on each processor is kept which is used to partition scanlines in the next frame. The first frame is load-balanced by looking at the load of scanlines of each slice on the processor. This strategy works well with a replicated object. For distributed volume, data partitioning the image and the volume are complimentary. Allocation of rays to processors needs to be guided by the volume data that is assigned to a processor. Two dimensional locality needs to be exploited for maximum reuse of data on processors to keep the communication costs low.

Each ray intersects the tree starting at the root. Tree traversal is done to find the smallest cell with a uniform color (white or black) that the ray intersects in a slice. The size of the cell determines the number of rays that are covered by this traversal. There is no need for additional tree traversals for these rays for the slice. This is done for each scanline in a slice. Rays in the scanline then pass through each slice accumulating opacities as they traverse the volume.

3.3.4 Summary

In this section we present the requirements for maintaining hierarchical data structures for volume rendering of multiple scenes of a 3D volume.

- *Data Partitioning:* Pixels from the resulting 2D view frame are distributed to processors, such that the work on each processor is balanced. This cannot be ensured by allocating equal number of pixels, since pixels have different work associated with them. A measure of work of rays being traced through pixels is maintained to estimate the workload on a processor. Work is represented by resampling, translation, tree traversal and compositing for the ray fired through a pixel. Scanline coherence can be preserved by allocating contiguous rows of scanlines to processors. The number of scanlines per processor would be data dependent and can either be adjusted between slices or between frames. See [20] for details.

- *Tree Construction:* The object data is represented by an octree. In parallel implementations, a distributed tree needs to be maintained to distribute data across the available processors, considering that each processor has memory only addressable by it. A k-d tree can be maintained to organize spatial volume data into partitions. A local tree can be built from the data a processor owns.

- *Locally essential data:* Each processor fires rays into the volume through the pixels it owns. Since the volume data is distributed, rays might need to fetch offprocessor volume data for compositing calculations. It can be determined by each processor independently which processor requires the data it owns, depending on the orientation of the view plane. A *sender-*

oriented protocol can be used to supply each processor with the data it needs for intersection and compositing calculations.

- *Incremental updates:* For rendering multiple sequences, with a change in the viewing angle, the rays fired from the pixels of each processor will intersect different parts of the volume data. Owing to object coherence and small angle changes in contiguous frames, most of the offprocessor data can be reused. The tree can be incrementally modified for each frame, instead of fetching all the data from scratch. Figure 12 shows the incremental data region when the view angle is changed by a small amount.

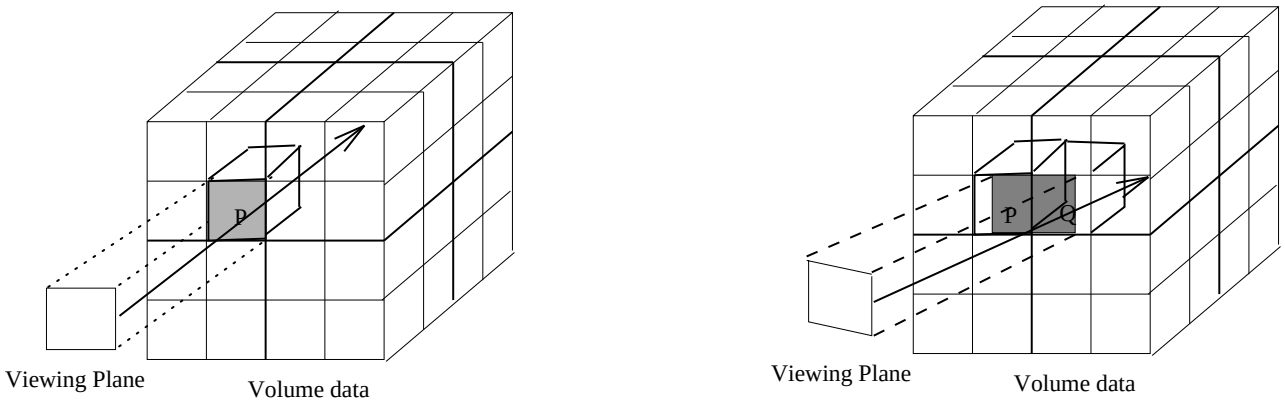

Figure 12: Ray intersections with volume (a) Viewing plane is aligned with the XY plane (b) Viewing plane is at an angle from the XY plane

- *Tree traversals:* Intersection of rays with voxel data at the appropriate level by skipping the empty regions is achieved by a traversing the tree from the top and finding the cell node with a homogeneous color. Intersection calculations are done at the highest level so that unncessary calculations can be saved.

3.4 Adaptive meshes

Many large physical systems can be modeled and represented by partial differential equations. Multigrid methods have been used to find solutions to these equations. A continuous domain can be discretized by overlaying a grid on top of it. The new discretized domain is now defined by the grid points. The grid points define a mesh. The characteristics of the solution guide the structure of the starting grid. In many problems, portions of the domain where high resolution is desired are localized. These allow the use of adaptive mesh refinement techniques, which allow a finer solution in more interesting areas of the problem domain and coarser solutions in areas of lesser interest. This can be considered as a multilevel method, where new levels are created by subdividing the

mesh at the existing level. Subdivision is adaptive and is controlled by an error estimate. The lower the error tolerance for a specific subdomain, the finer is the mesh at a deeper level of the grid hierarchy. Adaptive mesh refinement can be considered as a hierarchical tree representation of grids with each level, except the root, representing a uniform subdivision of the parent grid. The root represents the starting mesh with the initial grid points.

The Berger-Oliger adaptive mesh refinement scheme [8] is popularly accepted for the formulation of adaptive finite difference (AFD) methods. An adaptive grid hierarchy can be represented as a directed acyclic graph (DAG), where each node of the graph represents a component grid. The root corresponds to the base grid and the levels of the DAG correspond to the refinement in the grid hierarchy, nodes at the same level comprising of the component grids at the same level of refinement. A node can be denoted as G_n^l, $0 \leq l < L$, L being the lowest level in the hierarchy, and $0 \leq n < 2^l$. The adaptive grid hieraarchy is shown in Figure 13. The grid spacing at a level l is an integral multiple of the grid spacing at level $l+1$. Also, component grids at the same level must be locally uniform with space and time resolutions. The AFD integration algorithm defines the order of operations on the grid hierarchies and is composed of the *time integration*, *error estimation and regridding* and *inter-grid operations* components. All component grids at a level $l+1$ must be integrated to the current time T before integration begins for level l at time $T + \delta t$. Regions needing refinement are flagged based on the error estimate and a refined grid is generated wherever the error estimate is greater than required. Inter-grid operations are needed for the following:

- Initialization of the refined component grids by using the interior values of an intersecting component grid at same level or by prolongating values from the underlying coarser component grid.
- Updating underlying coarse component grids using values on a nested finer grid integrated to the same time T. This can be done by using the same values (injection) or an appropriate interpolation.
- Averaging any overlapping component grids at any level to update the coarser component grids underlying the overlap region.

The hierarchical structure of adaptive mesh refinement technique can be modeled as the generic tree-structured computation we are addressing in this paper. Inter-grid operations can be viewed as operations on tree nodes.

3.4.1 Parallelization on distributed memory machines

A parallel implementation of the adaptive mesh refinement method will require operations on the hierarchy of grids which include, creation of the grid, grid partitioning among processors,

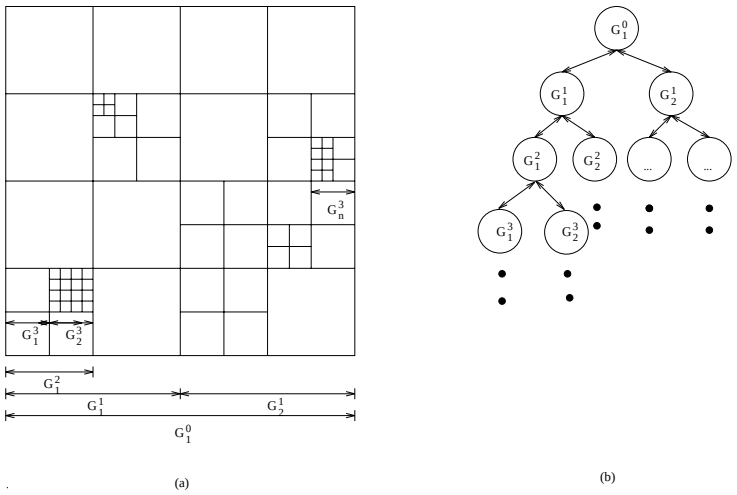

Figure 13: (a) Grid refinement and (b) associated grid hierarchy

communication among grids at one level and communication between grids at different levels. This allows us to primarily exploit *data-parallelism* by partitioning the grids across processors and *task-parallelism* expressed as independent updates to component grids and integration across levels in the grid hierarchy. The refinement at a particular level is not known a priori and hence can differ across the processors. This can lead to imbalance in the amount of work allocated to each processor. Hence dynamic load balancing is needed at runtime to distribute the work evenly among the processors.

A distributed DAG of grids has to be maintained across the processors. This is updated when grids are refined at each level. Communication is generated for inter level grid updates since every processor needs to keep a global view of the grid hierarchy. Prolongation defined from a coarser grid to a nested finer grid, and restrictions defined from a finer grid to it's parent grid, generate irregular communication of *scatter/gather* nature. Generally, a finer grid is distributed over a larger number of processors than a coarser grid, since the former has more grid points and hence more work associated with it. Intra-grid interactions may result in near-neighbor communication. This can be overlapped with computation if the interior-to-boundary ratio is large. Communication patterns are random when component grids are clustered or during grid redistribution. The grid decomposition scheme is critical to the effective parallelization of AFD methods. A composite distribution scheme can limit the cost of irregular communication for inter-grid updates by mapping the overlapping grids at different levels to the same processor. Locality maintaining mappings like the space filling curves define a mapping from a d-dimensional space to a linear ordering. Especially, *Morton ordering* and *Peano Hilbert ordering* are useful in partitioning space to preserve locality. Data partitioning can be performed by splitting the one-dimensional list appropriately. Redistribution can also be done by readjusting the points of the split. Grid partitioning among processors can be done in one of the following ways [44]:

- *Individual grid partitioning:* Each grid at every level is partitioned across processors. This generates communication pattern in which many processors may communicate with one processor generating a bottleneck. This happens when a fine grid needs to update its underlying coarse grid. Moreover, inter-level parallelism is not expolited since the grids at different levels are allocated to the same processor and their integration is sequential.

- *Comprehensive partitioning:* The work total across all the grids at all levels is distributed across processors. This distribution does not exploit the parallelism available in the problem since it will leave some processors idle while performing update calculations.

- *Independent level partitioning:* Each level of the grid hierarchy is partitioned independently. This scheme expolits the parallelism across grids at the same level, but generates irregular communication that if not scheduled appropriately may cause a bottleneck.

The fast adaptive composite grid method (FAC) [40] is a multilevel scheme that nominally uses global and local uniform grids for adaptive solution of partial differential equations. It provides parallelism by producing several independent refinement regions. Asynchronous version of FAC (AFAC) allows for processing of the refinement levels in a parallel mode. Using multigrid as the individual grid solver, FAC has been applied to a variety of fluid flow problems, including incompressible Navier-Stokes equations, in both two and three dimensions. The local grids add computational load irregularly to processors. Load balancing is thus required to implement AFAC efficiently on distributed memory machines. The independence of the various refinement levels in the AFAC process allows the assignments of computational tasks to processors to be made level by level. This simplifies the load balancing, since the levels can be ordered in a list and the partition of such a list is inherently one-dimensional. The relationship between the levels of the composite grid is expressed in a tree of arbitrary branching factor at each vertex. Each grid exists as a data structure at one node of this tree. The composite grid tree is replicated in each node. In the case of a change to the composite grid (adding, deleting or moving a grid), the change is communicated globally to all processors so that the representation of the composite grid in each processor is consistent. Storage of the matrix values uses a one-dimensional array of pointers to row values. These rows are allocated and deallocated dynamically, allowing the partitioned matrices to be repartitioned without recopying the entire matrix.

Edelsohn [15] has discussed the motivation of using the hierarchical approach for adaptive subdivision of meshes.

3.4.2 Summary

- *Data Partitioning:* Space filling curves can be used to partition the grid hierarchy across the processors. By maintaining appropriate data structures, repartitioning during load balancing can also be achieved.

- *Tree Construction:* Grid hierarchy represented as a distributed DAG is a hierarchical representation with an arbitrary degree at each node. A distributed tree structure with the appropriate fields is constructed as the base grid.
- *Incremental updates:* The grid hierarchy is dynamic and hence needs to be updated at each level. Indexing by code of the space filling curve can be used to move data appropriately.
- *Tree traversals:* Inter-grid and intra-grid updates need to go up or down in the grid hierarchy. This can be modeled as tree traversals on the nodes which basically are coarser or finer grids at different levels of the tree.

3.5 Hierarchical Radiosity

A more complex problem using hierarchical algorithms is that of calculating radiosity of a scene in Computer Graphics. The *radiosity* of a surface is defined as the light energy leaving the surface per unit area. Given a description of a scene the idea is to calculate the radiosities of all surfaces resulting in the calculation of illumination of the scene. A *scene* is a collection of large polygonal patches. These polygons are subdivided into small enough *elements* that the radiosity of an element can be assumed to be uniform over its surface. Any larger piece is termed as a *patch*, formed by combining elements or other patches including the original polygon. The radiosity of an element i can be expressed as a linear combination of all other elements j. The coefficients in the linear combination are the **form factors** between the elements. Form factor between element j and i (F_{ji}) is the fraction of light energy leaving element j arriving at i. This leads to a linear system of equations which can be solved for the element radiosities once all the form factors are known. Enforcing a energy balance at every element yields a system of equations of the form :

$$B_i = E_i + \rho_i \sum_j^n F_{ij} B_j$$

where B_i is the radiosity, E_i is the emissivity, ρ_i is the diffuse reflectance, F_{ij} is the form factor and n is the number of elements in the scene. This system of equations can be efficiently solved using iterative techniques like the Gauss-Siedel method. Its physical implication is equivalent to each patch successively *gathering* light. The other alternative is of *shooting* light from patches in order of their brightness. The most expensive part of the calculation is computing form factors, given for two infinitesimal elements by

$$F_{ij} = \frac{\cos\theta_i \cos\theta_j}{\pi {r_{ij}}^2} dA_j$$

The angle $\theta_i(\theta_j)$ relates to the normal vector of element i(or j) to the vector joining the two elements.

To take into account occlusion, differential form factors are accumulated only if the two infinitesimal elements are mutually visible. The form factor matrix is $n \times n$, where n is the number of elements. The order of form factor calculation is thus $O(n^2)$. Applying the insights of the N-body problem that

1. Numerical calculations are subject to error, and therefore, the force acting on a particle need only be calculated to within a given precision.
2. The force due to a cluster of particles at some distant point can be approximated within a given precision, with a single term, reducing the total number of interactions.

the complexity is reduced to $O(n + k^2)$, where k is the number of polygons.

The radiosity problem shares many similarities with the N-body problem. First there are *n(n-1)/2* pairs of interactions in both. The magnitude of the form factor falls off as $1/r^2$, same as the gravitational force.

The major difference between the two problems is the way the hierarchical data structures are formed. The N-body algorithm begins with n particles and clusters them in larger and larger groups. the hierarchical radiosity algorithm begins with a few large polygons and subdivides them into smaller and smaller patches. Subdividing is based on the error of a potential interaction which gives an automatic method for discretization of the scene. The principle of superposition, that the potential due to cluster of particles is the sum of the potentials of the individual particles, cannot be directly adopted because of occlusion. Intervening opaque surfaces can block the transport of light between two other surfaces which makes the system non-linear. Lastly, the N-body problem is based on a differential equation , whereas the radiosity problem is based on an integral equation.

3.5.1 Sequential Algorithm

The input to the algorithm is a set of polygons depicting the scene. These are inserted into a Binary Space Partitioning (BSP) tree [18] to facilitate efficient visibility computation between pairs of patches. Every input polygon is initially given a list of other input polygons that are potentially visible from it to enable it to compute interactions. The polygon radiosities are computed by iterating over the steps shown in Figure 14.

The tree data structure used here is not a single tree but a forest of quadtrees representing individual polygons. Each polygon has its own quadtree, with the roots being leaves of the BSP tree used for visibility testing. At every quadtree node visited in this traversal, interactions of the patch at that node are computed with all other patches in its interaction list. The interaction between two patches involves computing both the *visibility* and the *unoccluded form factor* between them and multiplying the two to obtain the actual form factor. Both of these quantities are

For every polygon quadtree
Start at the root, computing the form factors due to all polygons on its interaction list, subdividing it or other polygons hierarchically as necessary.
Compute the radiosity of the polygon first by a downward traversal of the tree followed by a upward sweep adding the radiosities.
Continue till convergence is achieved.

Figure 14: Radiosity algorithm

computed approximately, introducing an error in the computed form factor. An estimate of the error is also computed. If this is larger than a user defined tolerance the patch with the larger area is subdivided to compute a more accurate interaction. Children are created for the subdivided patch in its quadtree if they do not already exist. If the patch being visited (say i) is subdivided, patch j is removed from its interaction list and added to each of its children's interaction lists. If patch j is subdivided, it is replaced by its children on patch i's interaction list. Patch i's interaction list is completely processed in this manner before visiting its children in the tree traversal. During this downward traversal, the radiosities gathered at the ancestors of patch i and at patch i itself from other patches are accumulated and passed on to i's children. After the traversal of a quadtree is completed, an upward pass is made through the tree to accumulate the area-weighted radiosities of a patch's descendants into its own radiosity. Thus the radiosity of a patch is the sum of three quantities

1. Area-weighted radiosities of its descendants
2. The radiosity a patch gathers in its own interactions
3. The radiosity gathered by its ancestors

3.5.2 Available Parallelism

Parallelism is available at three stages. First, the polygons are independent of each other and can be processed simultaneously. Second the visibility computation can proceed in parallel. Third, the interactions computed for patches can be done in parallel. Singh [57] has discussed the parallel approaches on shared and distributed memory machines.

3.5.3 Parallel approaches on a shared memory machine

To exploit the parallelism across polygon-polygon interactions every processor needs to be assigned an equal number of them. This can either be done statically or processes can obtain interactions dynamically until none are left in the queue. Singh shows that the dynamic scheme works better than the static scheme. Apart from the distance between two patches, the angle and visibility between them are also important factors for interactions. Usual schemes that rely only on spatial distribution thus do not suffice. There are three primary forms of locality in the application.

1. A form of object locality can be obtained by having a processor work mostly on interactions involving the same input polygon or its subpatches in every iteration.
2. Locality can be exploited across each patch by processing sibling patches consecutively, using a breadth first traversal of the quadtree.
3. During visibility testing, using a depth first search, locality is exploited by ensuring that the same subset of BSP-tree nodes is traversed by a processor in successive visibility calculations

Load balancing can be provided by allowing on the fly task stealing. Each processor has a queue of polygons on which the interaction have to be calculated. A processor can either steal polygons from other processors and process them or steal patch-patch interactions. The granularity of tasks is a patch and all its interactions in the first case and a single patch-patch interaction in the second. If an interaction subdivides one of the patches and thus spawns a new interaction it is placed at the end of the creating processor's queue.

3.5.4 Parallel approaches on a distributed memory machine

Singh [57] suggests two parallel implementation of the radiosity algorithm on distributed memory machines which we discuss next.

The local quadtrees approach lets each processor maintain local copies of all the quadtree data that they need, modify the data locally as needed in an iteration and only communicate the modifications to other interested processors at iteration boundaries. In the absence of task stealing, the approach consists of phases of local computation punctuated by phases of communication. Also, there is no complete logical version of the forest of quadtrees on any one processor. Polygons are statically assigned to processors. The growing interactions on processors might lead to load imbalance in the system. Allowing idle processors to steal tasks from other processors can lead to load balancing.

In the global quadtrees approach a single copy of the forest of quadtrees is maintained in distributed form over the processing nodes. Every processor holds the BSP tree and the quadtrees

assigned to it in its local memory and knows the locations of the rest of the quadtrees through a global naming scheme. A processor can store non-local data it references in a local cache. To maintain coherence caches can be flushed at iteration boundaries. Modifications to data are always communicated to the master copy. The main disadvantages of this approach are that it requires fine-grained communication that is not phase-structured. It requires every reference to a quadtree datum to check whether the datum is local, nonlocal but cached locally, or remote.

3.5.5 Discussion

The complications to hierarchical radiosity arise from the dynamically changing quadtrees of patches, since they are built as computation proceeds. These data structures are not read-only but are actively read and written by different processors in the same computational phase during the calculation of the form factors. The following issues have to be addressed for any viable implementation

1. Naming of patches on different processors in a globally consistent manner to ease access to data during form factoe calculations.

2. Local quadtree versus Global quadtree approach needs to be investigated for its communication overhead. A level-by-level approach is proposed in this paper. Polygons are distributed among the processors. Each processor processes upto k levels, for a constant k, at which time the load at each processor is checked and if it falls below a threshold, patch-patch interactions are transferred to it from processors having more load.

3. Task stealing has been advocated for load balancing which has not been implemented efficiently yet on a distributed memory machine. Coherency is complicated by movement of patches between processors. We will investigate issues in handling messages for data, control, coherence, synchronization and load balancing while performing computation.

3.6 Image Compression

Applications using wavelet theory can use quadtree based decomposition for thier solution. Binary data compression and image compression are areas where *wavelet theory* has been widely applied. An approximation to the original image can be formed by using a *wavelet transform* A discrete sequence $x(n)$ of length N where $n, N \in Z$ is used to derive two subsampled signals $y_h(n)$ and $y_g(n)$ corresponding to the low and high pass versions of the original sequence $x(n)$ respectively. Each of the signals is of length $N/2$. The signal $y_h(n)$ is obtained by convolving the sequence $x(n)$ with a low pass filter $h(n)$ and dropping every other sample. Similarly $y_g(n)$ is obtained by using a high pass filter $g(n)$. The process of decomposing the sequence $x(n)$ into two subsequences at half

resolution can be iterated on either or both sequences. To achieve better frequency resolution at lower frequencies, the scheme is commonly iterated on the lower band.

In the first stage of wavelet decomposition this scheme is applied to 2-D images by applying the above scheme along the rows and then along the columns. The second stage applies the same procedure for the low-pass band. To obtain further stages of wavelet decomposition, the procedure can be applied to the low-pass band of the previous stage. This generates a pyramidal representation of the input image. Figure 15 shows a subband decomposition scheme that can be represented as a tree for computation.

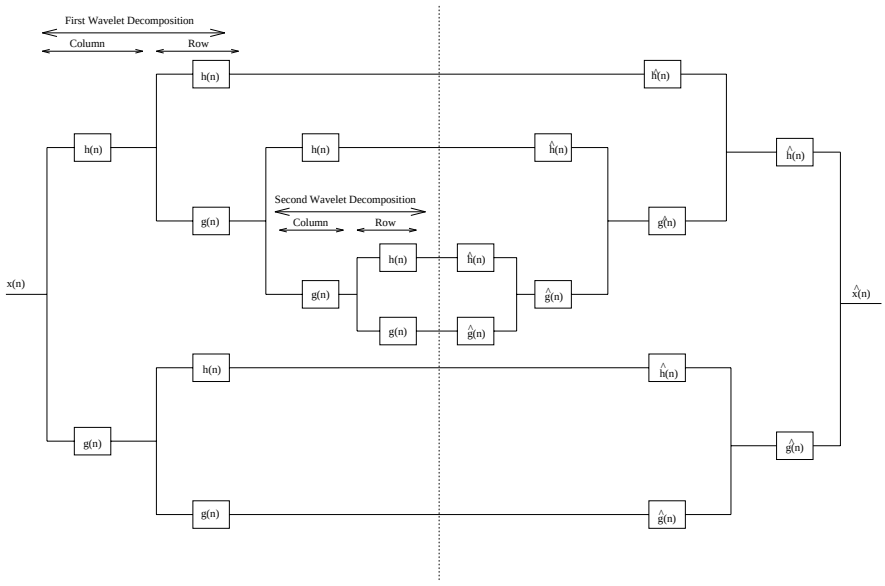

Figure 15: A subband decomposition scheme to be used with the Discrete Wavelet Transform.

Each wavelet picks up information about the image essentially at a given location and at a given scale. For portions of an image that has more interesting features, more coefficients can be used and where the image is nice and smooth a fewer coefficients can result in a good quality approximation. Hence there is an adaptive nature to the subdivision of the image form which makes image compression fall in the category of applications discussed so far. The details of this method can be found in [66] [67]. The Discrete Wavelet Transform (DWT) was developed in filter bank representations for subband coding for images and speech signals. The work on Quadrature Mirror Filters (QMF) has beeen used to decompose and reconstruct a signal.

4 Software System Requirements

The software requirements for hierarchical applications can be divided into *Architecture dependent* and *Architecture independent* activities. Current distributed memory machines are available in dif-

ferent configurations with varying interconnection networks. Message passing routines are typically architecture dependent and provided by the vendor. Global communication operations built using the basic communication primitives *send* and *receive* are also implemented by the machine vendor.

Hierarchical applications deal with treecodes and require basic tree manipulation routines. Queries on the tree data structure need to be performed to access and modify data structures. A library of routines providing primitives for tree construction and providing operations on the distributed data structure is needed. Load balancing aspects for the tree should be addressed at different levels by Global tree and Local tree management routines. Queries will extensively use data structure movement, which in most cases will be subtrees and will need encoding on the sender side for transmission. The receiver will decode the data to reconstruct the subtree. This *high level* data movement will use the low level data movement provided by the machine, using global communication whenever appropriate.

The main objective is to build an architecture independent software system for hierarchical applications and we would like to keep the architecture dependent part as small as possible. Communication libraries like PVM and MPI provide a layer of architecture independence and can be used.

Figure 16 highlights the software system that is needed for the applications discussed in this paper.

5 Conclusions

We have enumerated the computational structure of applications that use hierarchical algorithms. A discussion on the primitives for tree-codes, required for an effective solution on distributed memory machines, has been presented. We have identified a class of problems that require similar software support on parallel machines. Hierarchical methods were first used for N-body simulations and were later extended to other problems notably in Computational Fluid Dynamics and Computer Graphics. However, not much effort has been made for a coherent software support system for these methods on parallel machines, notably on distributed memory systems. Our effort is to develop a common paradigm for these problems which makes programming these methods easier for their users.

References

[1] A. Agarwal, D. Chaiken, G. D'Sousa, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B. G. Lim, G. Maa, D. Nussbaum, M. Parlin, and D. Yeung. "The MIT Alewife Machine:

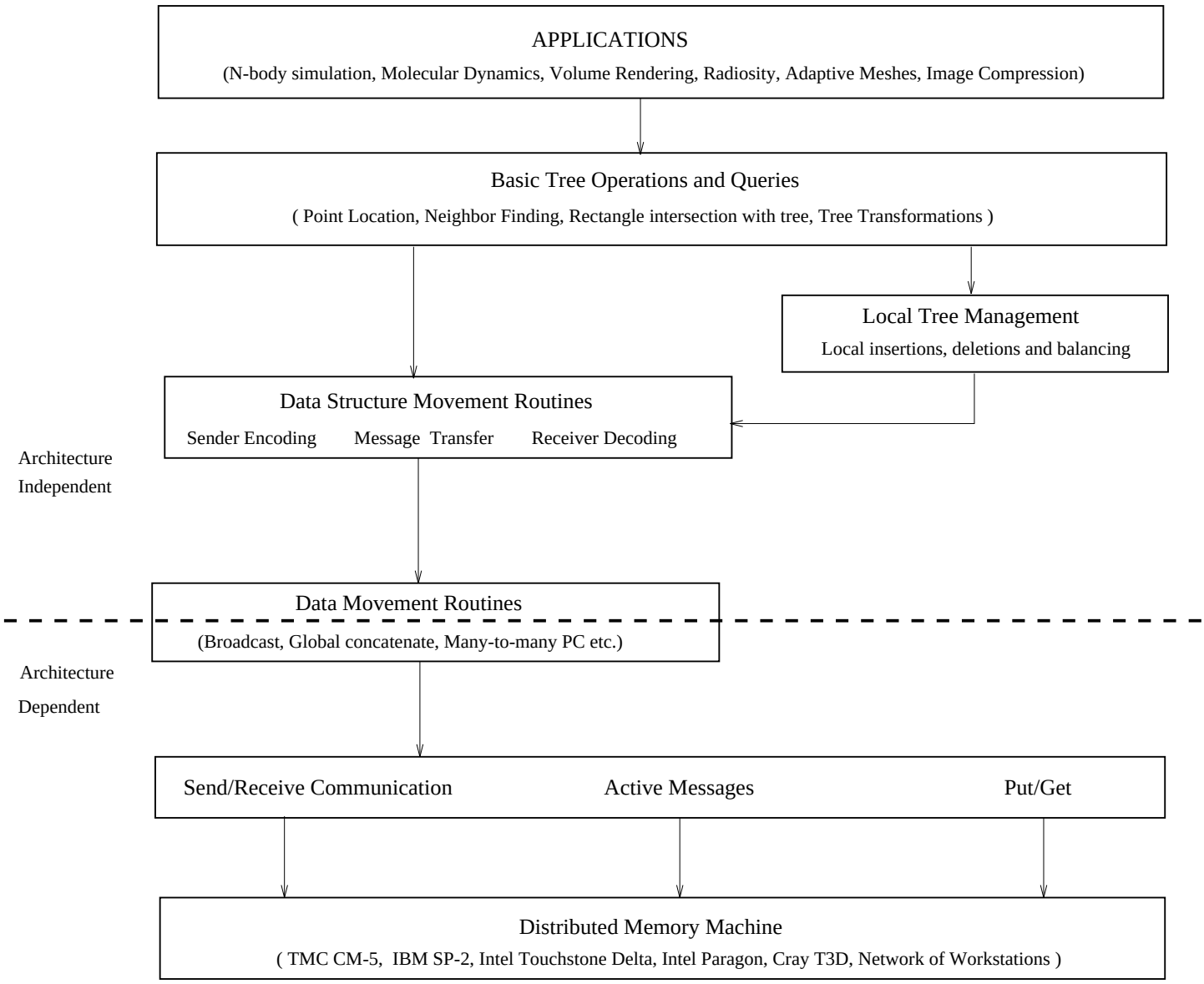

Figure 16: Software system for hierarchical applications

A Large-Scale Distributed Memory Multiprocessor," Technical Report MIT/LCS TM-454, Massachusetts Institute of Technology, Boston, MA, 1991.

[2] Appel A.W., *An efficient program for many-body simulation.*, SIAM Journal on Scientific and Statistical Computing,6,1985.

[3] Aupperle L., *Hierarchical algorithms for illumination.*, PhD thesis, Princeton University, 1993.

[4] Barnes J. and Hut P., *A hierarchical O(N log N) force calculation algorithm.*, Nature, 324, 1986.

[5] Barnes J., *A modified tree code: Don't laugh; it runs*, Journal of Computational Physics, 87, 1990.

[6] Bhatt S., Chen M., Lin C., Liu P., *Abstractions for parallel N-body simulation*, Scalable High Performance Computing Conference SHPCC, 1992.

[7] Badouel D., Bouatouch K. and Priol T., *Ray Tracing on Distributed Memory Parallel Computers: Strategies for distributing computations and data*, Parallel Algorithms and Architectures for 3D Image Generation - ACM SIGGRAPH '90 Course Note No. 28, pp 185-198, July 1990.

[8] Berger M. and Oliger J., *Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations*, Journal of Computational Physics, pp 484-512, 1984.

[9] Eric A. Brewer, Bradley C. Kuszmaul, "How to Get Good Performance from the CM-5 Data Network," *Proceedings of the 8th International Parallel Processing Symposium*, 8th IPPS, April 1994.

[10] Brooks B. R. and Hodoscek M., *Parallelization of CHARMM for MIMD machines*, Chemical Design Automation News, 7:16, 1992.

[11] Cameron G. G. and Undrill P. E., Rendering volumetric image data on a SIMD-architecture computer, *Proceedings of the Third Eurographics Workshop on Rendering*, 135-145, Bristol, UK, May 1992.

[12] Bozkus Z., Ranka S., Fox G.C., Benchmarking the CM-5 Multicomputer, *Proceedings of the Frontiers of Massively Parallel Computation*, 1992.

[13] Willian J. Dally and Chuck L. Seitz. "Deadlock-Free Message Routing in Multiprocessor Interconnection Networks," *IEEE Trans. on Computers*, 36(5):pp. 547–553, May 1987.

[14] Lenoski D., Laudon J., Gharachorloo K., Gupta A., Hennessey J., *The directory based cache coherence protocol for the DASH multiprocessor*, Proceedings of the 17th Annual International Symposium on computer Architecture, May 1990.

[15] Edelsohn David, Hierarchical Tree-Structures as Adaptive Meshes, International Journal of Modern Physics C, Vo. 4, No. 5, pp 909-917, 1993.

[16] G. C. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors*, vol. 1. Prentice Hall, Englewood Cliffs, NJ, 1988.

[17] Fox G., Hiranandani S., Kennedy K., Koelbel C., Kremer U., Tseng C., Wu M., *Fortran D Language Specification*, High Performance FORTRAN Forum, January 1992.

[18] Fuchs H., Abram G. D., Grant E. D., *Near Real-Time Shaded Display of Rigid Objects*, Computer Graphics, Vol. 17, No. 3, July 1983.

[19] Goil Sanjay, *Primitives for problems using hierarchical algorithms on distributed memory machines*, The First International Workshop in Parallel Processing, Bangalore, India, December 1994.

[20] Goil S. and Ranka S., *Dynamic Load balancing for Raytraced volume rendering on distributed memory machines*, International Conference on High Performance Computing, New Delhi, India, December 1995.

[21] Goldsmith J. and Salmon J., *Automatic Creation of Object Hierarchies for Ray Tracing*, IEEE Computer Graphics and Applications, May 1987.

[22] Greengard L. and Rokhlin V., *A fast algorithm for particle simulations.*, Journal of Computational Physics, 73, 1987.

[23] Hernquist L., *Vectorization of tree traversals*, Journal of Computational Physics, 87, 1987.

[24] High Performance Fortran Forum, *High Performance Fortran Language Specifications*, CRPC-TR92225, Center for Research in Parallel Computing, Rice University, January 1993.

[25] W. J. Dally et al. "The J-Machine: A fine-grain concurrent computer," *Proceedings of the IFIP Congress*, G. X. Ritter, ed., pp. 1147–1153, North-Holland, August 1989.

[26] Katzenelson J., *Computational Structure of the N-Body Problem*, SIAM Journal of Scientific and Statistical Computing, Vol 10, No. 4, July 1989.

[27] Karia R., *Load balancing of Parallel Volume Rendering with Scattered Decomposition*, Techical Report, Dept. of Computer Science, Australian National University, Canberra, Australia.

[28] Ma K., Painter J. S., Hansen C. D., Krogh M. F., *A Distributed Parallel Algorithm for Ray Traced Volume Rendering*, Parallel Rendering Symposium, October 1993.

[29] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*, Benjamin-Cummings, 1994.

[30] Lacroute P. and Levoy M., Fast Volume Rendering Using a Sheat-Warp Factorization of the Viewing Transformation, *SIGGRAPH'94*, August 1994, Florida.

[31] Leiserson Charles, *Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing*, IEEE Transactions on Computers, Vol. C-34, No. 10, October 1985.

[32] C. Leiserson et al. "The Network Architecture of the Connection Machine CM-5," *Proc. 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, San Diego, CA, 1992.

[33] Levoy M., *Display of Surfaces from Volume Data*, IEEE Computer G raphics & Applications, May 1988.

[34] Levoy M., *Efficient Ray Tracing of Volume Data*, ACM Transactions on Graphics, Vol. 9, No. 3, pp 245-261, July 1990.

[35] Levoy M., *Volume Rendering by Adaptive Refinement*, Visual Computer, Vol. 6, No. 2, pp 2-7, 1990.

[36] Liu P., *The Parallel Implementation of N-body Algorithms*, PhD Thesis, Rutgers University, 1994.

[37] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, T. von Eicken, "LogP: Towards a Realistic Model of Parallel Computation," *Proceedings of 4th ACM Symposium on Principles and Practices of Parallel Programming*, pp. 1–12, 1993.

[38] Makino J., *Comparison of two different tree algorithms.*, Journal of Computational Physics, 88, 1990.

[39] Beazley, P. et. al. *Parallel Algorithms for Short-Range Molecular Dynamics*, World Scientific's Annual Reviews in Computational Physics, vol. 3 (1995).

[40] McCormick S., Quinlan D., *Asynchronous multilevel adaptive, methods for solving partial differential equations on multiprocessors : Performance Results*, Parallel Computing, 12, 1989.

[41] Montani C., Perego R., Scopigno R., *Parallel Volume Visualization on a Hypercube Architecture*, Proceedings of the Boston Workshop on Volume Visualization, Boston, MA, October 19-20, 1992.

[42] Nieh J., Levoy M., *Volume Rendering on Scalable Shared-Memory MIMD Architectures*, Proceedings of the Boston Workshop on Volume Visualization, Boston, MA, October 19-20, 1992.

[43] Lionel M. Ni and Philip K. McKinley. "A Survey of Wormhole Routing Techniques in Direct Networks," *IEEE Computer*, 26(2):62–76, February 1993.

[44] Parashar M. and Browne J., *An Infrastructure for Parallel Adaptive Mesh-Refinement Techniques*, Technical Report, University of Texas, Austin, 1995.

[45] Choudhary A., Fox G., Hiranandani S., Kennedy K., Koelbel C., Ranka, S., Saltz, J., *Software Support for Irregular and Loosely Synchronous Problems*, Proc. of the Conf. on High Performance Computing for Flight Vehicles, 1992.

[46] A. Bar-No. and S. Kipnis. "Designing Broadcasting Algorithms for the Postal Model for Message-Passing Systems," *Proceedings of 4th ACM Symposium on Parallel Algorithms and Architectures*, 1992, pp. 13–22.

[47] Preparata F. and Shamos I., *Computational Geometry - An Introduction*, Springer-Verlag, 1985.

[48] Ranka S., Aluru S., Goil S., and Al-Furaiah I., *Parallel tree construction on a MIMD machine*, Under preparation, Syracuse University.

[49] Reif J. and Tate S., *The complexity of N-body simulation*, International Colloquium on Automata Languages and Programming, 1993.

[50] Salmon, J., *Parallel Hierarchical N-body methods* PhD thesis, Caltech, 1990.

[51] Hwang Y., Das R. and Saltz J., *A Data-Parallel Implementation of Molecular Dynamics Programs for Distributed Memory Machines*, University of Maryland Technical Report.

[52] Samet H., *The Quadtree and Related Hierarchical Data Structures*, ACM Computing Surveys, Vol. 16, No.2, June 1984.

[53] Samet H., *The Design and Analysis of Spatial Data Structures*, Addision-Wesley Publishing Company, 1990.

[54] Samet H., *Application of Spatial Data Structures*, Addision-Wesley Publishing Company, 1990.

[55] S. Bae and S. Ranka. "Experimental Evaluation of Different Message Paradigms on the CM-5 for Irregular Problems," *Frontiers '95*. To appear.

[56] Schröder P. and Stoll G., Data parallel volume rendering as line drawing, *Proceedings of the Boston Workshop on Volume Visualization*, pp 25-32, Boston, October 1992.

[57] Singh J. P., *Parallel Hierarchical N-body Methods and thier Implications for Multiprocessors.*, PhD thesis, Stanford University, 1993.

[58] Singh J.P., Gupta A., Levoy M., *Parallel Visualization Algorithms: Performance and Architectural Implications*, IEEE Computer, July 1994.

[59] Sundaram S., *Fast Algorithms for N-body Simulations*, PhD thesis, Cornell University, 1993.

[60] Warren M. and Salmon J., *Astrophysical N-body simulations using hierarchical tree data structures.*, Proceedings of Supercomputing, 1992.

[61] Warren M. and Salmon J., *A parallel hashed oct-tree N-body algorithm*, Proceedings of Supercomputing, 1993.

[62] Spach S., Pulleybank R., *Parallel Raytraced Image Generation*, Hewlett-Packard Journal, June 1992.

[63] T. von Eicken, D. E. Culler, S. C. Goldstein, K. E. Schauser. "Active Messages: a mechanism for integrated communication and computation," *Proceedings of the ISCA '92*, Gold Coast, Australia, May 1992.

[64] Kobayashi H., Nishimura S., Kubota H., Nakamura T., Shigei Y., *Load balancing strategies for a parallel ray-tracing system based on constant subdivision*, Visual Computer, Vol. 4, No. 4, pp 197-209, October 1988.

[65] Green S. A., Paddon D. J., *A Highly Flexible Multiprocessor Solution for Ray Tracing*, Visual Computer, Vol. 6, No. 2, pp 62-73, 1990.

[66] Jawerth B. and Sweldens W., *An overview of wavelet base multiresolution analyses*, SIAM Review, Vol. 36, pp 377-412, September 1994.

[67] Krishnaswamy D. and Orchard M., *Parallel Algorithms for the two-dimensional discrete wavelet transform*, Proceedings of International Conference on Parallel Processing 1994.

[68] Wilhelms J. and Allen Van Gelder, Octrees for faster isosurface generation., *Computer Graphics*, 24(5):pp 57-62, November 1990.

[69] Yau M. and Srihari S., *A Hierarchical Data Structure for Multidimensional Digital Images*, Communications of the ACM, July 1983, Volume 26, No. 7.

[70] Zima H., Chapman B., *Vienna FORTRAN - A Fortran Language Extension for Distributed Memory Multiprocessors*, High Performance FORTRAN Forum, 1992