Object-Oriented Support for Adaptive Methods on Parallel Machines

SANDEEP BHATT[1], MARINA CHEN[2], JAMES COWIE[2], CHENG-YEE LIN[2], AND PANGFENG LIU[2]

[1]*Bell Communications Research, Morristown, NJ 07962*

[2]*Department of Computer Science, Yale University, New Haven, CT 06520*

ABSTRACT

This article reports on experiments from our ongoing project whose goal is to develop a C++ library which supports adaptive and irregular data structures on distributed memory supercomputers. We demonstrate the use of our abstractions in implementing "tree codes" for large-scale N-body simulations. These algorithms require dynamically evolving treelike data structures, as well as load-balancing, both of which are widely believed to make the application difficult and cumbersome to program for distributed-memory machines. The ease of writing the application code on top of our C++ library abstractions (which themselves are application independent), and the low overhead of the resulting C++ code (over hand-crafted C code) supports our belief that object-oriented approaches are eminently suited to programming distributed-memory machines in a manner that (to the applications programmer) is architecture-independent. Our contribution in parallel programming methodology is to identify and encapsulate general classes of communication and load-balancing strategies useful across applications and MIMD architectures. This article reports experimental results from simulations of half a million particles using multiple methods.

1 INTRODUCTION

Broadly speaking, parallel programs are written in either of two styles. In the reactive style of programming, the user specifies the local computation and interaction between individual processors. On the other hand, within the global style, the user invokes operations on global data structures; the local behavior of each processor is derived by the underlying programming language implementation.

The message-passing programming paradigm, an example of the former, has been used in research and development of parallel programs for at least a decade. High Performance Fortran (HPF) [1], however, is the result of a recent effort in supporting the latter. By and large, HPF inherits the Fortran programming model, whose implementations support array computation most efficiently. From a standpoint similar to HPF, our goal is to adapt C++ (or some other object-oriented language) for the global style of parallel programming, but to stress the language interfaces and implementation strategies that will provide significant data structure support in C++ where it is lacking in HPF.*

Received April 1993

Scientific Programming, Vol. 2, pp. 179–192 (1993)
CCC 1058-9244/94/040179-14

* Fortran90, the base language of HPF, has adequately addressed derived data structure issues, and paid a great deal of attention to pointers. However, HPF is currently based on a subset of Fortran90 where the distribution of such derived data structures is not addressed. Fortran90 also lacks object-oriented support, which appears to pose a limitation for constructing flexible and specializable distributed data structures.

The application that motivates this work is N-body simulation [2, 3], which presents several challenges:

1. Data structures for solving such problems may change in a noncontinuous fashion during computation; for example, a small change in a particle location can result in a tree node removed from one subtree and grafted onto another.
2. For true scalability, it is desirable that data structures be built, accessed, and updated in an incremental and distributed fashion.
3. Data reference patterns need to be computed dynamically because they change continuously during the computation. This situation is different from that of sparse matrix computations where a reference pattern can be learned, cached, and reused at run-time [4, 5].
4. Data must be redistributed to processors to maintain load balance.

By way of analogy with Fortran90/HPF, we know that whatever distributed data structure we support must have global operators analogous to Fortran90 array intrinsics (e.g., CSHIFT, EOSHIFT) and data distribution directives analogous to BLOCK and CYCLIC strategies in HPF. Yet to meet these challenges, we can no longer retain a single-layered application-system interface such as that implied by HPF.

A more general treatment of distributed derived data structures will require freer control and data flow between application and system modules. The challenges that seem considerably harder for derived data structures than for arrays are to support resolution between global and local naming spaces, as well as fine coordination between structure traversal and per-element computation. This style of interface, in turn, requires linguistic mechanisms such as polymorphic classes and functions (called templates in C++, not to be confused with TEMPLATE in HPF).

From the user's standpoint, the distributed data structure (DDS) run-time system is a layer beneath an object-oriented application data structure (e.g., a tensor library [6]) that an application writer may define and an end-user may inherit and refine for a specific problem [7]. Figure 1 illustrates this layered approach.

The DDS layer is common to multiple machines but optimized for porting to each machine. Thus, it shields the user from the details of machine architecture. The DDS layer also contains a library of distribution classes from which a user may choose a particular strategy for the specific algorithm at hand. Clearly, the distribution class library writer, analogous to a compiler writer responsible for implementing the BLOCK or CYCLIC strategy, needs much more intimate knowledge of the DDS run-time system than the application writer.

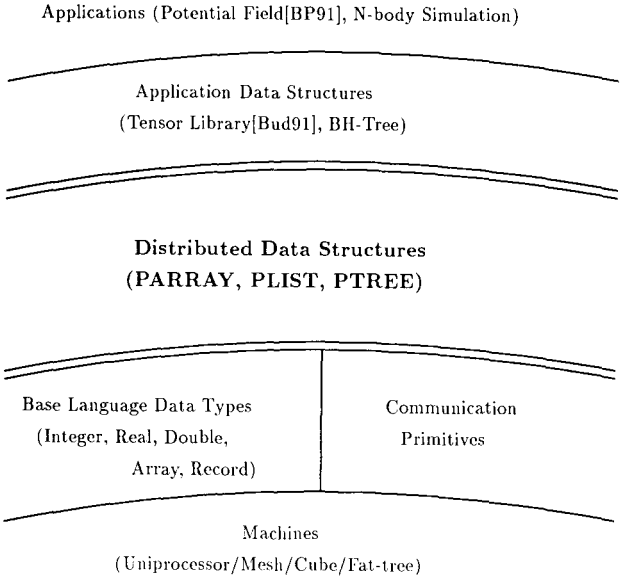


FIGURE 1 The layered approach.

We now focus on one type of DDS, a distributed tree called PTREE for discussing the run-time system organization.

1.1 PTREE: Top Level Overview

PTREE separates tree operations into two interacting run-time subsystems, link traversal and per-node computation. A tree traversal begins as a single-link traversal to the logical root, following which control alternates between single-link traversal and per-node evaluation until the traverse is complete.

A third subsystem, the *mailbox module*, supports the sending of raw or typed data between any two nodes, using some application-supplied global addressing method. Complex application-specific protocols for PUT and GET, involving multiple synchronization and communication phases, can be constructed from this primitive basis.

These three logical PTREE subsystems are independent of the physical implementation, de-

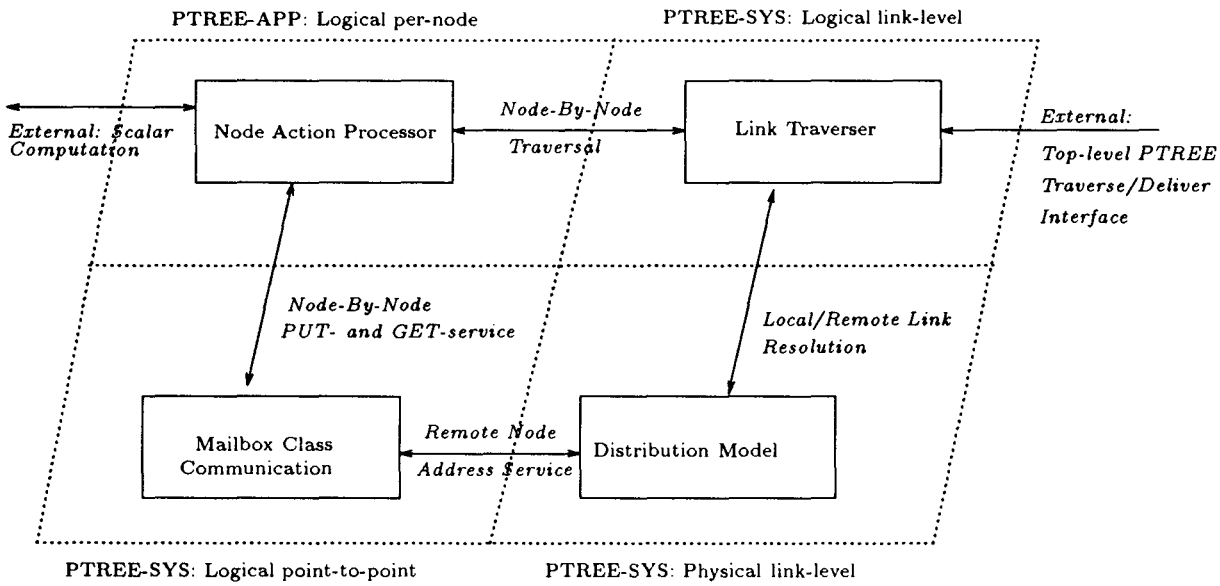


FIGURE 2 Modules, ownership, and control flow within PTREE.

coupling logical tree operations from knowledge about the extent or topology of the underlying processor space. To accomplish this, a fourth subsystem, the distribution module, provides key services to the other three, resolving logical link traversals and global mailbox addresses to either local pointers or remote processor/pointer pairs.

Implementation of the subsystems takes place in stages. The physical details of distribution are coded as one class, the logical protocols for traversal and delivery in another two, and the per-node computations in a fourth. Application details (distinguishing specific instances of user trees) are separated from generic tree operations, and these in turn from details of their multiprocessor instantiations. This top-level structure for PTREE is shown in Figure 2.

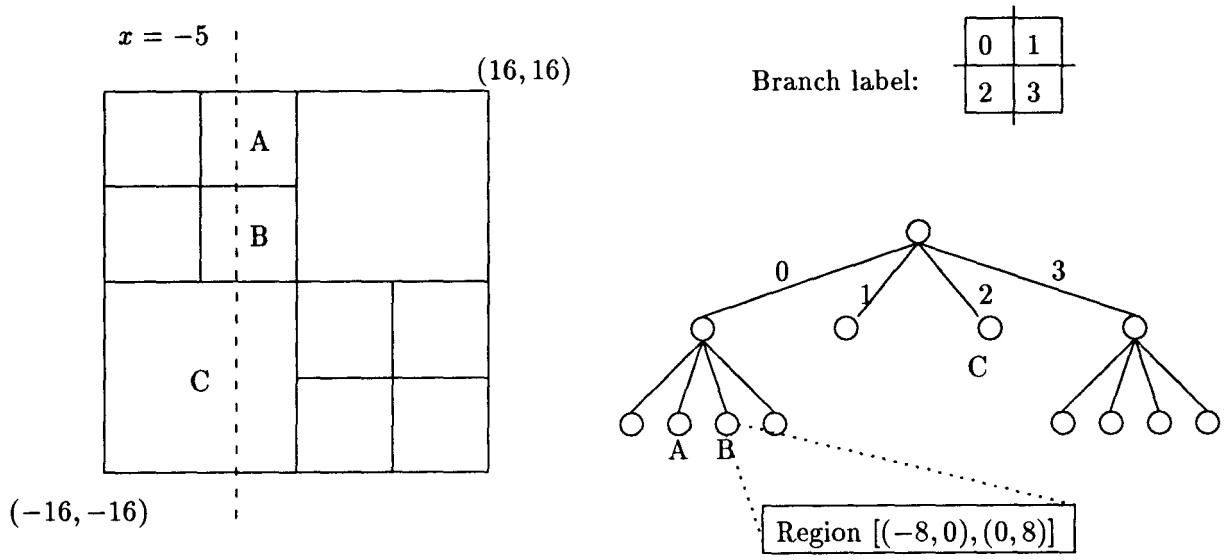
1.2 Related Work

Significant use of object-oriented technology in scientific computing and its advantage has been demonstrated [8–10]. C++ has been adapted for parallel computing in a variety of contexts: Paragon C [11] supports distributed arrays; P++ [12] employs array class extensions specialized for concurrent structured-grid computation; PC++ [13, 14] contains distributed data structures such as arrays, priority queues, lists, etc., which can be distributed using directives such as whole, block, cyclic, random; and CC++ [15] is compositional language that allows a program to be built from parts that are based on different programming styles and models.

The rest of the article is organized as follows: We show the key components of a PTREE runtime system and their interactions via an application tree `Geo_tree` in Section 2. Mechanisms for maintaining a distributed data structure and allowing adaptive remapping are described. The concept of structural coherence is introduced. **Max-scan**, an example of a global tree operator, is used to illustrate the link traverse and per-node computation dichotomy as well as the relationship between logical data transfer versus physical communication. Section 3 describes an N-body simulation code written using classes of the PTREE system and some preliminary performance findings. Some future directions are discussed in Section 4.

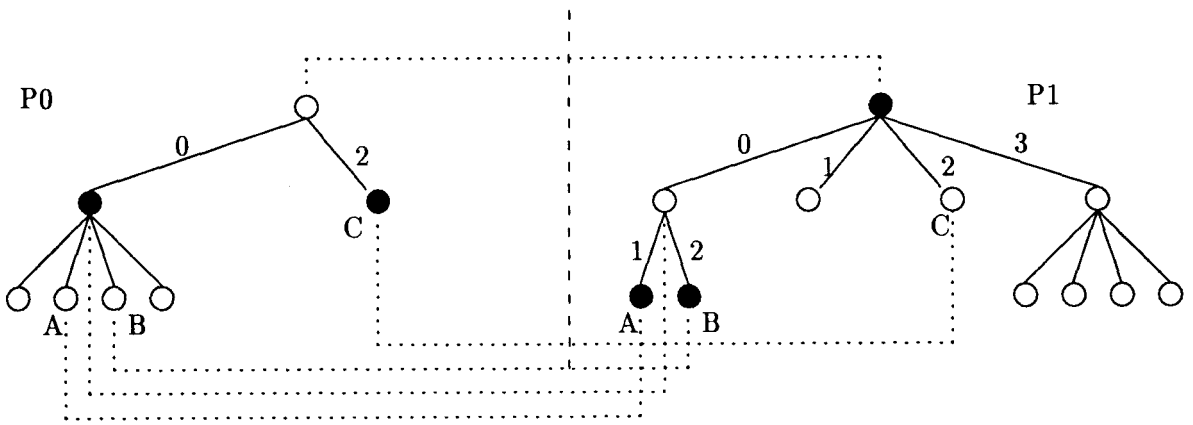
2 DISTRIBUTED DATA STRUCTURES: AN EXAMPLE

Our goal is to provide the user with the perception of a global data structure while efficiently implementing it as a collection of local data structures on the processors of a scalable, distributed-memory machine. Let us begin with an example tree, called `Geo_tree`, which represents a two-dimensional (2-D), rectangular domain that is divided recursively into smaller domains as shown in Figure 3a where the subdivisions are the children



(a) Recursive subdivision of a rectagular domain.

(b) A uniprocessor tree representation.



(c) Tree partition onto two processors.

FIGURE 3 Distribution strategy that is based on the ORB method.

nodes of the original domain. The global tree (or a uniprocessor implementation of it) and its branch labels are shown in Figure 3b.

A distributed implementation of traversal and computation over `Geo_tree` takes the form of four cooperating run-time subsystems: tree distribution, per-node computation, link traversal, and point-to-point mailbox services.

2.1 Distribution Module: Mechanisms to Keep Track of the Global Structure

Once a global encoding and distribution strategy for a data structure such as `Geo_tree` has been determined, its operations can be coded and managed in the form of a distribution class. Any distribution class will have two major components: distributed system state and system services, implemented, respectively, as private class data and public class methods.

2.1.1 Encoding, Partitioning, Sharing Strategy

As shown in Figure 4, tree nodes have an encoding, in this case, just the coordinates of the lower left and upper right corners of the rectangular boxes.

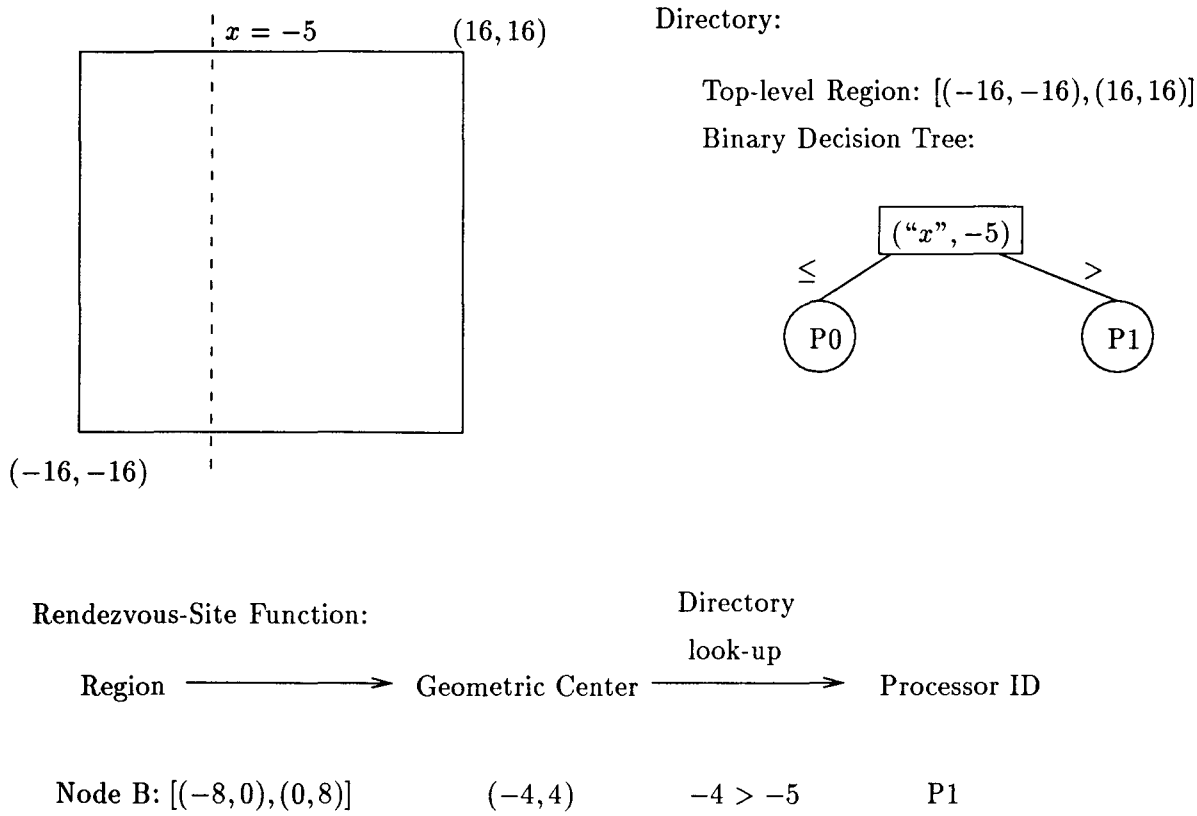


FIGURE 4 Encoding, global directory, and rendezvous-site function.

This `Geo_tree`, in reality, is distributed to multiple processors, where computations are carried out on each of its parts and the partial results are merged in some ways so as to achieve the same effect as the uniprocessor implementation. Figure 3c shows the `Geo_tree` partitioned into two parts by a bisection line. Some tree nodes in the global tree are replicated on more than one processor because the bisection line passes through the boxes these nodes represent.

Thus, the distributed implementation of a global tree node can be a set of shared nodes, each representing a portion of the box. A particular node of this node set, which is colored black in Figure 3c, is called the rendezvous site of the shared nodes. The rendezvous site is where the partial results from each processor are merged and sent back to other shared nodes in the same set. In this example, the rendezvous site of a set of shared nodes is the node representing the portion of a box that contains the geometric center of that box.

2.1.2 System State

The distribution class tracks which nodes are shared between processors, where each associated rendezvous site resides, and which edges leaving these nodes actually lead off-processor. For `Geo_tree`, this means building a private table on each processor describing the geometric bounding boxes owned locally, as well as the degree of spatial overlap between the coverage provided by "neighboring" processors.

To locate an arbitrary node in the tree, the distribution class also maintains a path prefix directory, which can be used locally to reconstruct the global tree distribution. In the `Geo_tree` example, the directory consists of the representation of the 2-D domain and a binary decision tree that

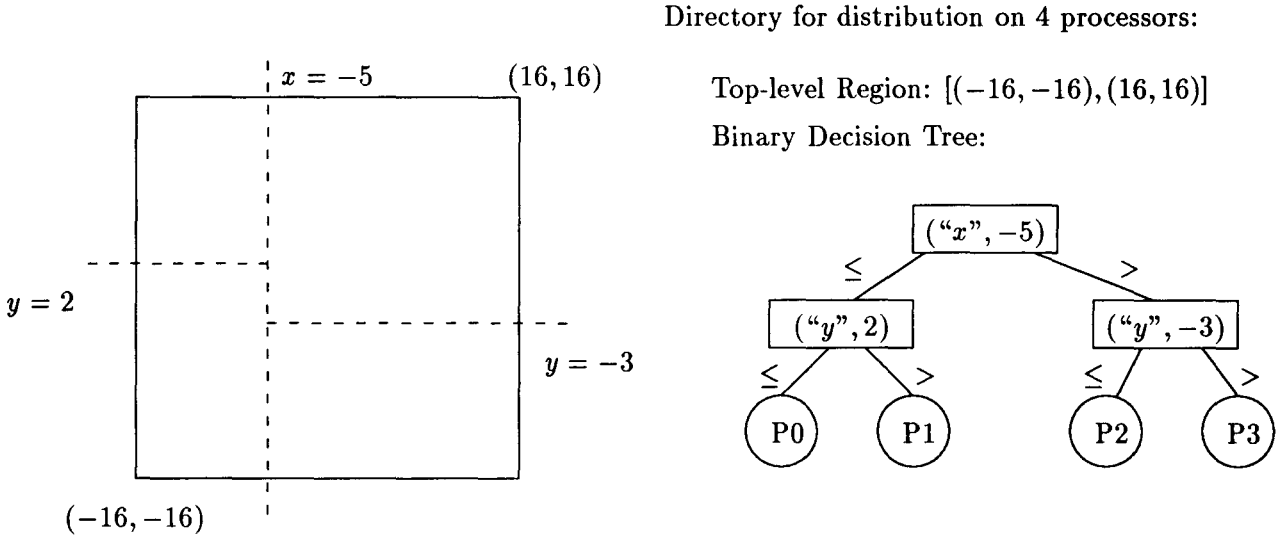


FIGURE 5 Global directory for partitions on to four processors.

encodes the bisection lines. Figure 5 illustrates the directory structure for a four-processor implementation of the tree.

2.1.3 System Services

Using these fragments of system state, the distribution class provides certain services to application code, including the resolution of global node encoding to processor/pointer pairs, determination of whether a given node is locally resident, and interception of edge-traversals that leave the resident node set.

These services are rarely used in their raw form; rather, they participate in the design by the application writer of more abstract interfaces such as global Traverse, Deliver, and Remapping (described later) over the `Geo_tree`.

2.2 Per-Node Functions: Computational Kernels

Traditionally, tree computations are expressed as single decision and computation routines, executed on each visit to a tree node, which compute results from node contents and "decide" which of the children to visit, and in what order.

The per-node class provides a basis for separating the processes of decision and computation from the system implementation. In the case of `Geo_tree` used in the context of N-body simulation, a geometric traversal must decide at each node, based on particle density and the bounding box size, whether to accumulate results from subbisections or to settle for the current approximation.

This can be expressed as a public method of the per-node class that repeatedly passes control to the `Geo_tree` link traverser for each child visit, accumulating a final result to be returned to the link traverser responsible for the original incoming tree edge.

2.3 Link Traverse: Neighbor Access

The link-traverser class provides the "control glue" to carry out each of the traversal requests made by per-node methods. To accomplish this, the traverser makes use of the link resolution service provided by the system in the `Geo_tree` distribution class.

This corresponds, in `Geo_tree`, to making transitions between bounding boxes, pursuing the bisection tree into smaller and smaller regions of the problem space, potentially wandering out of the set of bounding boxes resident on this processor.

2.4 Mail Boxes: Point-to-Point Access

The mailbox class provides a basis for direct node-to-node data transfer. Like link traversal, mailboxes work at the purely logical level, relying on the distribution class to resolve target node encoding to processor/pointer pairs.

In `Geo_tree`, mailboxes are used to construct methods for tree accesses that do not respect bisection boundaries (i.e., dataflow between bounding boxes that are spatial neighbors, but not tree siblings under the given bisection scheme). By us-

ing the distribution class services to identify shared nodes, mailboxes can also be used to make local results available to other sharing-set participants and to pack multiple incoming results according to a rule (such as maximum, exclusive-or, or first-arrival).

2.5 Global Operations

Using these four interacting run-time subsystems, the application writer builds derivative methods implementing more abstract global operations over `Geo_tree`. Some examples of characteristic global operations of this type are, in increasing order of complexity and specialization:

1. Traverse and Deliver, which do not modify tree structure and, therefore, distributed system state
2. Insert and Delete, which modify distributed system state but can be coded as specializations of Traverse and Deliver
3. Remap, which provides very high-level support for an adaptively load-balanced `Geo_tree` class

Two basic derivative methods with natural application for `Geo_tree` are Traverse and Deliver. Traverse, which performs the distributed local equivalent of a global tree traversal, computes a (generally associative, nondestructive) function from data in the nodes encountered along the way. Deliver performs data migration en masse among processors to propagate local results among sharing sets, synthesizing a global result.

2.5.1 Traverse

Global Traverse can be defined as a per-node program, guiding the computation and visitation of children. To implement the global Traverse, the per-node and link-traversal subsystems repeatedly trade control to carry out a traversal of each processor's local resident subtree. To determine whether a tree link wanders out of the resident node set, the link traverser consults the `Geo_tree` distribution class. When this occurs, processors must synchronize and partial results must be communicated through each rendezvous site.

2.5.2 Deliver

This communication phase is implemented in Deliver by a combination of both local traversal (again, via the per-node and link-traverser subsystems) and point-to-point mailbox traffic (the mailbox subsystem). For many functions (associative, nondestructive), Traverse and Deliver are completely separable, as only one global synchronization is required. Others, with multiple communication phases, will require Traverse and Deliver to be mutually recursive, and to execute as paired coroutines. We give more details of this process in the later discussion of "structural coherence."

As an example, operation max-scan is defined as a scan operation on the global `Geo_tree` such that field max of each node is assigned the maximum of the value fields of its descendants (Fig. 6). This global operation can be implemented as a call to the Traverse function followed by a call to the Deliver function; Traverse computes partial results (local maxima) for all the nodes on the local tree, and then Deliver combines the partial results into final results by accumulating the local maxima on the shared nodes into the "home" nodes resident at the rendezvous site.

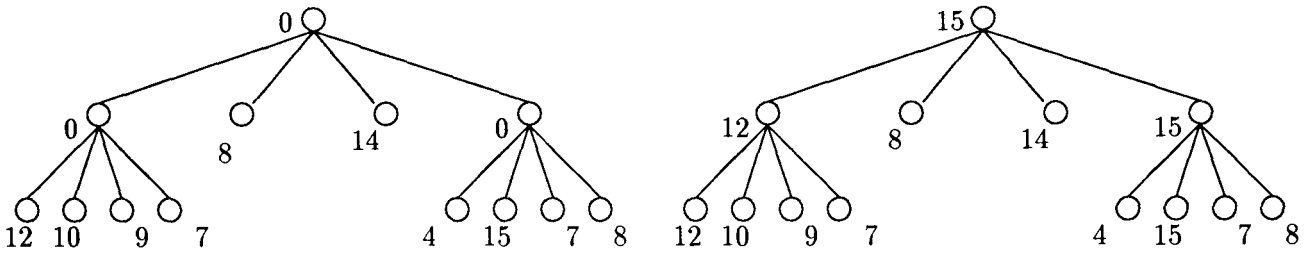
2.5.3 Insert and Delete

Given these basic methods, node insertion and deletion can be coded as traversal and delivery processes that modify the tree structure, using more complicated per-node functions that combine normal traversal-oriented search with logic to sever and create tree edges. Again, the distribution class must be consulted to correctly handle insertion and deletion of shared nodes, and to keep the distributed system state coherent.

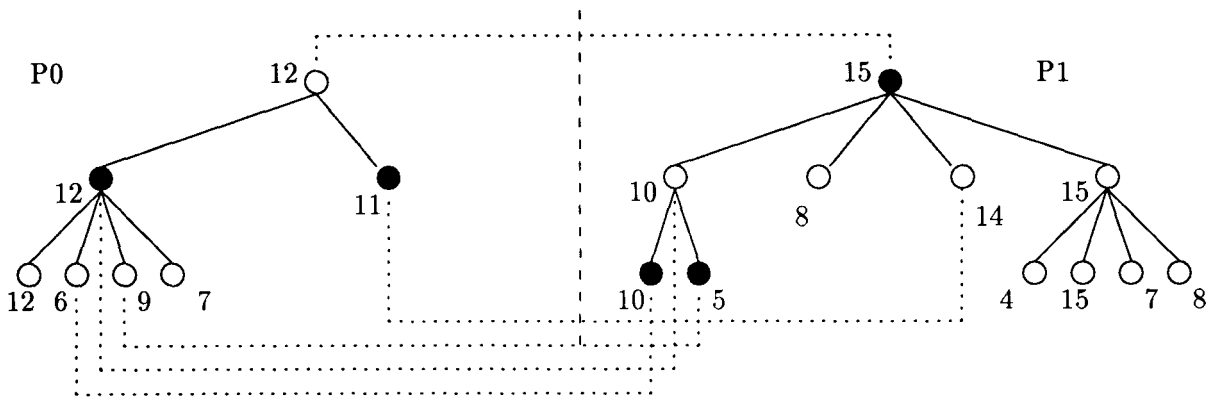
2.5.4 Remapping

What happens if we want to change the distribution due to change of workload represented by each tree node? Suppose the new bisection line for `Geo_tree` is now $x = 2$ as shown in Figure 7a. The updated directory and the new distribution, shared nodes, and rendezvous sites are shown in Figure 7.

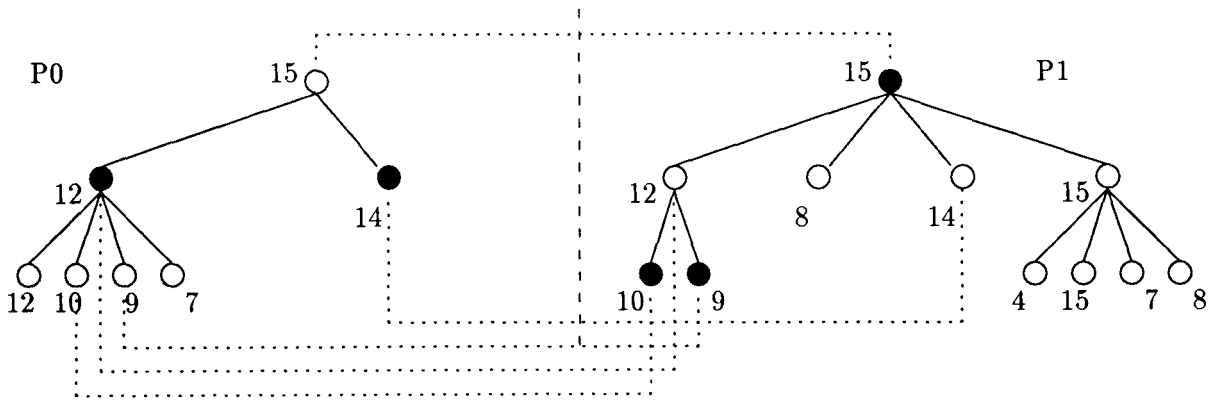
Fortunately, the `Geo_tree` remapping can be expressed in terms of elemental insertion and deletion, serialized across processors. The Remap method can therefore take the form of a simple wrapper around insertion and deletion, plus some coordinating code between each processor instance of the distribution class to determine the serialization ordering. Additional optimizations to pursue might include on-line batching of the Insert and Delete orders in some consistent way.



(a) Initial values and final results of **max-scan** on a uniprocessor tree.



(b) Traverse phase for the tree distributed onto 2 processors.
Each local tree has partial results.

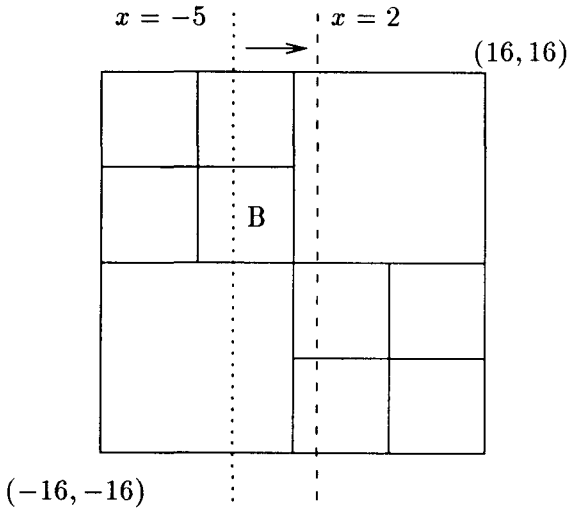


(c) Deliver phase for local trees.
The partial results in shared nodes are combined into final results.

FIGURE 6 Compute **max-scan** on distributed trees.

2.6 Structural Coherence

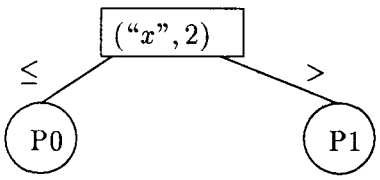
Finally, we want to bring attention to the way we distribute and build a correct ensemble of local structures that is equivalent to the global structure. Clearly, in general, partitioning a global structure built from an input data set would not necessarily yield the same result as partitioning the input set and then building the local trees. Figure 8a shows a uniprocessor in-order search



Directory:

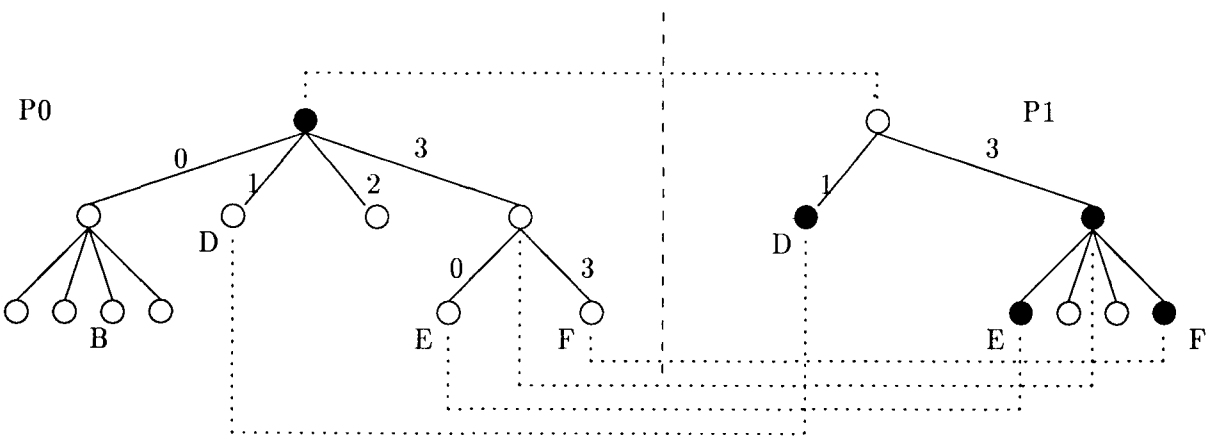
Top-level Region: $[(-16,-16),(16,16)]$

Binary Decision Tree:



Rendezvous-Site Function:		Directory look-up	
Region	Geometric Center		Processor ID
Node b: $[(-8,0),(0,8)]$	$(-4,4)$	$-4 < 2$	P0

(a) Adjust the bisection line.

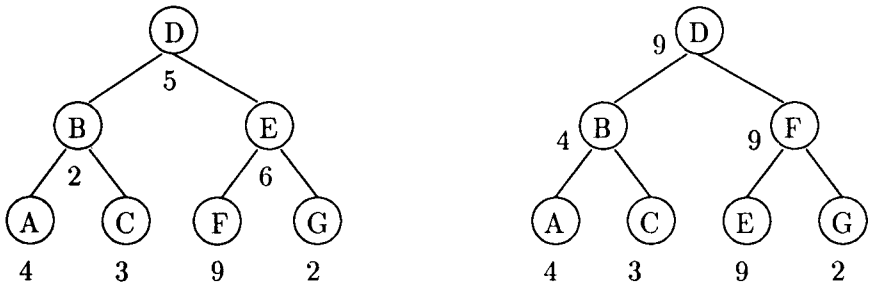


(b) New distribution with bisection line $x = 2$.

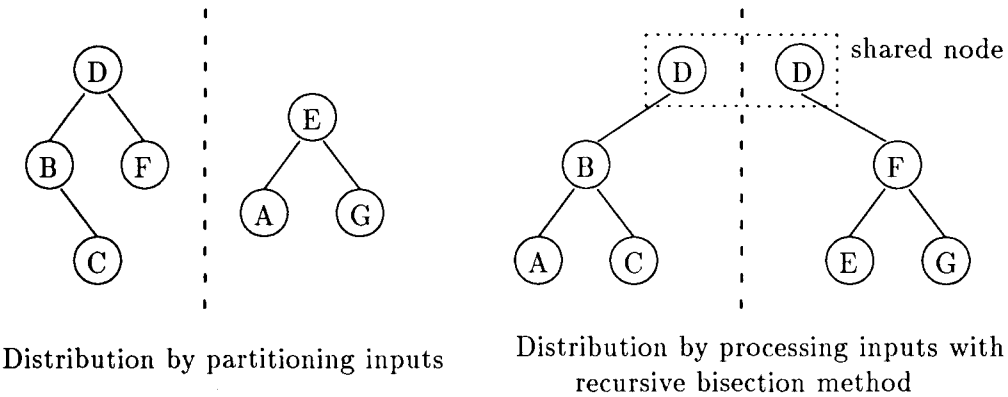
FIGURE 7 Remapping: Adjust the bisecting line.

tree built from an input set of key and value pairs. Figures 8b and c contrast two different ways of distributing the global tree, and the erroneous result one might get for a distributed data structure that does not have structural coherence. The forest on the left is a result of dividing the input set into two portions, one for each processor, and then building two local in-order trees. The forest on the right has shared nodes and is obtained by preprocessing the input set by recursively bisect-

Inputs (key, value): (D, 5), (B, 2), (F, 6), (C, 3), (E, 9), (A, 4), (G, 2).



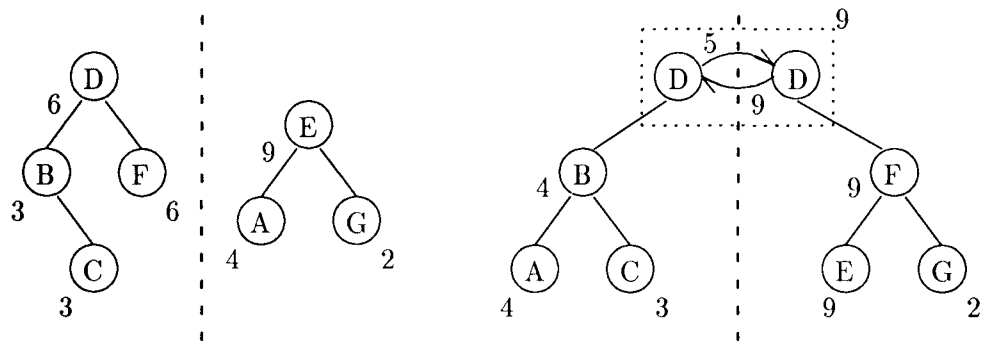
(a) Initial values and final results for **max-scan** on a uniprocessor inorder search tree.



Distribution by partitioning inputs

Distribution by processing inputs with recursive bisection method

(b) Two different tree distributions onto 2 processors.
The distribution on the left builds incoherent local search trees, while the distribution on the right builds coherent local trees.



(c) Find **max-scan** results:
incorrect values for a distributed that is NOT structurally coherent.

FIGURE 8 Structural coherence.

ing the input set using "good" pivot elements that help to generate a balanced global tree. Such preprocessing can be done on a central host, but large problems would require a distributed bisection sort over some arbitrary or random input distribution (fortunately, there exist many good parallel algorithms that accomplish this). For on-line problems, the input stream should be batched so that the preprocessing can be done incrementally and distributed over the processor set with amortized cost.

3 APPLICATION: *N*-BODY SIMULATION

3.1 Problem Description

Computational methods to track the motions of particles that interact with one another have been the subject of extensive research for many years. So-called "N-body" methods have been applied to problems in astrophysics, semiconductor device simulation, molecular dynamics, plasma physics, and fluid mechanics. In this section we consider the example of gravitational N-body simulation.

The gravitational N-body simulation is simply stated as follows. Given the initial positions and velocities of N particles, update their positions and velocities every τ time-steps. The instantaneous acceleration on a single particle can be directly computed by summing the contributions from each of the other $N - 1$ particles. Although this method is conceptually simple, its $O(N^2)$ arithmetic complexity rules it out for large-scale simulations.

Beginning with Appel [16] and Barnes and Hut [2], there has been a flurry of interest in faster algorithms for large-scale particle simulations. The number of operations per time-step is $O(N)$ for Appel's method, and $O(N \log N)$ for the Barnes-Hut algorithm provided the particles are uniformly distributed in physical space. Greengard [17] presented an $O(N)$ algorithm that is provably correct to any fixed accuracy. However, because of the complexity and overhead in the fully adaptive version of Greengard's algorithm, the algorithm of Barnes and Hut continues to enjoy application in astrophysical simulations.

All these N-body algorithms are based on a divide-and-conquer strategy. The basic idea is to group particles within an oct-tree, which is used to calculate interactions. Because the above algorithms share the tree structure in common, they are all commonly referred to as "tree codes."

To organize particles into a hierarchy of clusters, the Barnes-Hut algorithm computes an oct-tree partition of the 3-D box (region of space) enclosing the set of particles. The partition is computed recursively by dividing the original box into eight octants of equal volume until each individual box contains exactly one particle.† 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 filled in a bottom-up phase.

The tree codes all exploit the idea that the effect of a cluster of particles at a distant point can be approximated by a small number of initial terms of an appropriate power-series. The Barnes-Hut algorithm uses a single-term, center-of-mass approximation.

To compute accelerations, we perform a top-down BH-tree traversal for each particle. When the traversal reaches an internal node, we compute the distance between the particle and the box of this internal node.‡ If the distance is less than RADIUS(box)/θ then the particle visits each of the children recursively; otherwise, the acceleration due to the cluster is approximated by a single two-body interaction between the particle and a point mass located at the center-of-mass of the cluster.

Once the accelerations on all the particles are known, the new positions and velocities can be computed. The entire process, starting with the construction of the BH-tree, is then repeated for the desired number of time-steps.

3.2 Distribution Strategy: ORB Class

The orthogonal recursive bisection (ORB) is a well-known method for distributing particles to multiple processors by recursively bisecting a given region along alternating dimensions such that each subregion contains an approximately equal number of particles. This goal is to keep the processes load balanced. Our example `Geo_tree`, an abstraction of the BH-tree, uses this same method over a 2-D domain.

In N-body simulation, the particles are first distributed onto processors by using ORB. Each processor builds a local portion of the BH-tree with its

† In practice it is more efficient to truncate each branch when the number of particles in its subtree decreases below a certain fixed bound.

‡ The distance measured can be either the distance from the particle to the center-of-mass, or to the boundary of the box, and this distance can be in any preferred metric. The radius of a box is simply the length of a side of the box.

own share of particles. All the local trees share the same root corresponding to the enclosing region of all particles. Because the region for a node can be shared (or split) by more than one ORB partition, such nodes are shared between a set of processors to keep the tree structurally coherent—the superposition of local trees is equivalent to the global tree.

For the node encoding scheme, the geometric center of a node uniquely defines the node's position, i.e., path and depth, in the global tree. The global directory replicated across all processors contains ORB bisection information, represented as a binary decision tree as shown in the `Geo_tree` example.

The node-to-processor mapping can be derived by looking up the geometric center of a node in the directory. This mapping also determines the rendezvous site of a set of shared nodes.

3.2.1 Generic Distributed Tree: PTREE Class

The support for distributed tree structures consists of two parts: (1) a polymorphic C++ class for defining a distributed tree structure, and (2) polymorphic functions for applying computation and communication on the tree structure. An application tree class is defined by customizing PTREE with type parameters for (1) the input data list for tree building, (2) the actual contents of a tree node, and (3) the distribution strategy.

3.2.2 Customizing PTREE: BH-Tree Class

BH-tree is the main data structure in this application, it customizes the PTREE class template by supplying the types of particle lists, the node contents, and the ORB distribution.

3.2.3 Per-Node Functions for BH-Tree Class

To perform global operations on the BH-tree, the application needs to supply the node computational kernels specific to this applications. One per-node function compares the geometric centers of the input node with the node currently being visited to determine if the search has terminated and, if not, which branch to extend the search along. Another wraps the particle data from an input list into a node to be inserted into the tree.

3.2.4 Structural Modification and ORB Remapping

In this application, particles move in space during the simulation, which may cause load imbalance and trigger remapping. Particles may move out of the region represented by a BH-tree leaf, causing the leaf to be deleted from the tree; conversely, when a particle enters a spatial region represented by a given BH-tree node, that leaf node may split, spawning a new subtree. In particular, when a particle moves across the boundary of ORB subregions, interprocessor communication is involved. Function `deliver` together with PTREE class per-node functions perform the interprocessor node movement in a way consistent with the global tree structure.

The ORB class provides a hierarchical remapping function to rebalance the workload in an incremental way. Instead of remapping on a global scale, the remapping is confined to processors within imbalanced regions.

The remapping process works in a top-down fashion. On each level of bisection:

1. Check whether the load difference between the two sides of bisection is within a balancing threshold.
2. If true, remapping is not necessary on this level. Move down to the bisections on both sides.
3. Otherwise:
   (a) Adjust the current bisecting plane to balance the load on both sides (using per-node functions with lookup service from the ORB class).
   (b) Relocate particles with respect to the adjusted bisection (a simplified version of the method used for structural change mentioned above).
   (c) Recursively adjust the bisections on both sides.

3.2.5 Some Preliminary Performance Findings

To determine the overhead due to the object-oriented abstraction, a hand-written C version was compared with the C++ version using the PTREE distributed data structure. The hand-coded version is based on traversals of lists and trees, whereas the C++ version uses derived classes to encapsulate the same algorithm. For the experiments, a half million particles are simulated on a 64-node iPSC/860 machine. This shows a rea-

Step	t_{hand}(ms)	t_{PTREE}(ms)	Overhead(%)
Update BH-tree			
list deliver	2937	3116	6.1
list traverse	46	52	—
Compute & transmit Center-of-Mass			
tree traverse	558	635	13.8
tree deliver	374	415	11.0
Compute positions & velocities			
tree deliver	7928	8086	2.0
list traverse	87917	88774	1.0
Move to new positions			
list deliver	963	990	2.8
Remapping			
list deliver	829	854	3.0
Total	101552	102922	1.3

FIGURE 9 Comparisons between handwritten and PTREE version.

sonably good performance on PTREE version, with less than 2% overhead over its hand-written counterpart. With a 64-processor configuration and 8 K particles per processor, Figure 9 summarizes the comparisons of timing results of averaging of 10 simulation steps for five runs.

These early experiments indicate that if BH-tree performance is to scale well with problem size, four sources of distributed overhead must be amortized over the amount of per-node computation available:

1. The basic communication cost of maintaining shared node sets, which must synchronize and share partial tree results
2. The simple overhead of performing global node insertions and deletions to model particle migration, exclusive of load-balancing considerations
3. The cost of performing global load balancing, either incrementally or periodically, to derive compact, balanced load trees within each processor and a roughly equitable distribution of particles among all processors
4. The overhead of supporting a fine-grain interface between application and system code, consisting of compositions of many lightweight class methods, with a significant amount of time tied up performing function calls

Paying the cost of adaptive load balancing should help a BH-tree application recover some of the overhead of sharing, insertion, and deletion. If the distributed tree can be kept globally in balance and locally compact, the number and size of shared node sets can be minimized. Furthermore, load balancing reduces the average local path length for search and computation, keeping down costs of traversal and delivery, as well as tree maintenance.

Finally, our comparisons to C hand-coded versions of identical distributed algorithms suggest that the last source of overhead, the only one that arises from adherence to an object-oriented interface, is not significant compared to the others, which are common to any adaptive distributed implementation.

It is important to note, though, that the performance figures presented in the tables are preliminary, in that they represent runs on a single initial distribution of particles. We are planning further, rigorous, experiments that will be reported in the future.

4 CONCLUDING REMARKS

In the future, users will continue to require increasingly generic, specializable, distributable, flexible data structure packages. Explorations with PTREE indicate that this will create certain tensions for the implementors of object-oriented languages, including future versions of C++.

Unlike HPF-style parallel arrays, the new wave of inheritable generic data structures will offer little opportunity for lifting out large blocks of common functionality to be implemented as monolithic library interfaces. Rather, distributed data structure implementations will very likely require

1. A higher degree of coordination between system and application code
2. A proliferation of lightweight methods for providing small, encapsulated run-time services;
3. Attention to reducing the amount of state retained by each run-time subsystem thread, with an emphasis on continuation-passing programming styles and reentrant application code
4. More aggressive compiler support for inlining, perhaps at the expense of restricting the separate compilation model

More people will likely participate in the design of a distributed PTREE code than for an HPF application, though, to manage this complexity by

stages. A "system provider" might well code the underlying distribution class support, and a "data structure developer" the global tree operations (and the underlying common subsystems for per-node and mailbox functionality). The end-user would perform the final (hopefully relatively trivial) instantiation and class linking required to actually embed the PTREE-derived library in a given application. For example, the N-body simulation will be written with a major user class BH-Tree derived from the application classes GeoTree and ORB, which in turn are defined in terms of the system classes PTREE and DISTRIBUTE.

REFERENCES

[1] High Performance Fortran Forum, *High Performance Fortran Language Specification (DRAFT)*. Version 1.0 Draft, January 1993.

[2] J. Barnes and P. Hut, "A hierarchical O(N log N) force-calculation algorithm," *Nature*, vol. 32, pp. 446–449, 1986.

[3] J. Salmon, "Parallel hierarchical N-body methods," PhD thesis, Caltech, 1990.

[4] L.-C. Lu and M. Chen, *Fourth Workshop on Languages and Compilers for Parallel Computing*, 1991.

[5] J. Saltz, R. Mirchandaney, and K. Crowley, "Run-time parallelization and scheduling of loops," *IEEE Trans. Comput.* vol. 40, pp. 603–612, 1991.

[6] K. G. Budge, *Physlib: A C++ Tensor Class Library*. Technical Report, Sandia National Laboratories, Albuquerque, NM, October 1991.

[7] K. G. Budge and J. S. Peery, *A Numerical Representation of Fields Using C++ Classes*. Technical Report, Sandia National Laboratories, July 1991.

[8] I. G. Angus and W. T. Thompkins, *Data Storage Concurrency, and Portability: An Object Oriented Approach to Fluid Mechanics*. Palos Verdes Peninsula, CA: Northrop Research and Technology Center, 1989.

[9] D. W. Forslund, C. Wingate, P. Ford, S. Junkins, J. Jackson, and S. C. Pope, "Experiences in writing a distributed particle simulation code in C++," *USENIX C++ Conference*, pp.1–19, 1990.

[10] J. S. Peery, K. G. Budge, and A. C. Robinson, "Using C++ as a scientific programming language," *CUG11*, 1991.

[11] C. M. Chase, A. L. Cheung, and A. P. Reeves, *The Paragon Programming Environment User's Guide*. Ithaca, NY: School of Electrical Engineering, Cornell University, 1991.

[12] M. Lemke and D. Quinlan, "P++, a parallel C++ array class library for architecture-independent development of structured grid applications," *Workshops on Languages, Compilers, and Run-Time Environments for Distributed Memory Multiprocessors*. Boulder, CO, 1992.

[13] J. K. Lee and D. Gannon, "Object-oriented parallel programming experiments and results," *Supercomputing*, 1991.

[14] S. X. Yang, J. K. Lee, S. P. Narayana, and D. Gannon, "Programming an astrophysics application in an object-oriented parallel language," in *Scalable High Performance Computing Conference SHPCC*. Williamsburg, VA: IEEE Press, 1992.

[15] K. M. Chandy and C. Kesselman, *Compositional C++: Compositional Parallel Programming*. Technical Report Caltech-CS-TR-92-13, Computer Science Department, California Institute of Technology, 1992.

[16] A. W. Appel, "An efficient program for many-body simulation," *SIAM J. Sci. Stat. Comput.*, vol. 6, pp. 85–103, 1985.

[17] L. Greengard, *The Rapid Evaluation of Potential Fields in Particle Systems*. MIT Press, 1988.