Supporting Efficient Tree Structures for Distributed Scientific Computing

Pangfeng Liu
Department of Computer Science
and Information Engineering
National Chung Cheng University
Chia-yi, Taiwan 62107
pangfeng@cs.ccu.edu.tw

Jan-Jan Wu
Institute of Information Science
Academia Sinica
Taipei, Taiwan 11529
wuj@iis.sinica.edu.tw

Abstract

This paper describes an implementation of a platform-independent parallel C++ framework that can support various scientific simulations involving tree structures, such as astrophysics, semiconductor device simulation, molecular dynamics, plasma physics, and fluid mechanics. Within the framework the users will be able to concentrate on the computation kernels that differentiate different tree-structured scientific simulation problems, and let the framework take care of the tedious and error-prone details that are common among these applications.

This framework was developed based on the techniques we learned from previous CM-5 C implementations, which have been rigorously justified both experimentally and mathematically. This gives us confidence that our framework will allow fast prototyping of different scientific simulation applications that run on different parallel platforms and deliver good performance as well.

We used MPI to implement the communication routines within the framework for better portability, therefore the parallel library, as well as the applications developed within the framework, can run on every parallel machine where MPI is available. The applications remain portable across different platforms, and the communication library within the framework can be customized to explore possible performance gain based on individual characteristics of different parallel machines.

To demonstrate the flexibility and performance of this framework we implemented a gravitational force field computation code and a multi-filament vortex method on a SPARC Ultra workstation cluster, and report very competitive speedup even compared with a highly optimized sequential C implementation.

1 Introduction

Distributed memory parallel machines are notoriously difficult to program. The huge amount of data, common in large-scale scientific simulations, must be partitioned among processors evenly while maintaining data access locality. In a message-passing communication environment one must "think in parallel" to write programs that not only compute the results correctly, but also send the right data to the right place at the right time. Additionally the programmer must schedule all the processors properly to avoid racing, even deadlock conditions. As a result, parallel programming for distributed memory systems often involves many intricate details that are error-prone and difficult to debug. Therefore, users should reuse existing working codes whenever possible, and a runtime library that supports high level communication and data structure abstraction is a very important tool for developing parallel programs.

Related works

To make it easy to write parallel programs on distributed memory systems; there has been a lot of research efforts focusing on runtime library that abstracts out the intricate parallel programming details. Examples of work along this line include the Paragon package [9], which supports a special class PARRAY for parallel programming, the A++/P++ Array class library [34], PC++ proposed by Lee and Gannon [24, 43], which consists of a set of distributed data structures (arrays, priority queues, lists, etc.) implemented as library routines, where data are automatically distributed based on directives. Interwork II Toolkit [5] described by Bain supports user programs with a logical name space on machines like iPSC. The user is responsible for supplying procedures mapping the object name space to processors. These libraries greatly reduce the programming complexity and let the users concentrate on the computation, not the communication details. Unfortunately, these libraries are mostly for regular multi-dimensional arrays, and there has been very little support for dynamic and irregular pointer-based data structures (e.g. a tree), either from runtime systems or compilers.

We propose a runtime library that supports irregular and dynamic tree structures in distributed memory systems. These tree structures are widely used in scientific simulations, including astrophysics, semi-conductor device simulations, molecular dynamics, and fluid dynamics. The tree structure is adaptive to the simulation configuration; therefore it can have extremely irregular shape and dynamically evolve as the simulation proceeds. As a result it is extremely difficult to implement this dynamic tree structure using primitive message-passing library. By hiding the communication and data structure details, our library can remove the burden of writing complicated parallel programs from the application writers.

There have been a lot of implementations of tree-based scientific simulations on various application domains. Barnes and Hut [6] developed the first tree algorithms for gravitational simulation. Parallel implementations of Barnes-Hut's algorithms for astrophysics, fluid dynamics computations are described in [8, 35, 37, 38, 40, 41], and parallel fast multipole implementations include [20, 21, 33, 37, 44]. In a related work by ourselves [27, 29, 30], we implemented a parallel gravitational N-body simulation using Barnes-Hut's algorithm in C and CM-5 communication library. Due to the non-uniform tree structure, none of these implementations is data parallel – they are progrmmed in either SPMD or shared-memory programming model. Recently, Hu et al[42] report a data parallel N-nody code in which the irregular tree structure is linearialized and partitioned so that data parallel progrmming techniques can be applied. For example, HPF language/compiler/runtime support were used to achieve efficiency.

Unfortunately, all of these previous implementations pursue only execution efficiency, and none of them consider systematic software *reusability* and *portability.* These implementations did not separate the generic data structure from the application-dependent computation kernel, and they are built for one simulation problem on one particular machine. Therefore, it takes tremendous efforts to convert an astrophysics simulation code running on one machine into a fluid dynamics code running on another, even though many aspects of the codes are similar. One must reorganize the code to salvage any reusable parts manually, and piece together these fragments to form a new program which the new computation kernel will hopefully fit into. This "cut-and-paste" human intervention is time-consuming and error-prone.

To eliminate the duplicated programming investments in developing similar tree-based simulation codes, our runtime library promotes code reusability among tree-based computations. Most of the tree-based simulation codes use similar tree structures and exhibit similar computation patterns. There are two levels of similarities. First, a fluid mechanics code and a molecular dynamics code may differ only in the interaction formula. The tree structures are basically the same except for the data stored in tree nodes and the implementation-dependent tree representation. Secondly, different simulation algorithms may use the same data structure. For example, fast multipole method and Barnes-Hut's algorithm use the same oct-tree structure – they differ only in how they manipulate the trees. Therefore, a general framework for tree-based scientific computation helps in developing tree codes in a way that basic tree constructs and operations can be shared among different application domains and simulation algorithms.

Our runtime library also achieves portability on the source program level. The library hides the implementation details in portable MPI library so that it can run on every platform where MPI is available. In addition, the object-oriented approach allows us to use the best communication protocol on a particular platform to achieve maximum efficiency. These communication details are hidden from the users so that the same source code can run on different platforms with maximum efficiency.

Our effort has similar goals and approaches to the POOMA package [4] and the Chaos++ library [36]. POOMA supports a set of distributed data structures (fields, matrices, particles) for scientific simulations. To our knowledge, POOMA has not supported adaptive data structures as our library does. Chaos++ is a general-purpose runtime library that supports pointer-based dynamic data structures through an inspector-executor-based runtime preprocessing technique. On the other hand, our framework focuses on dynamic trees and is able to exploit optimizations that would be difficult for a general preprocessing technique to find.

In addition to the above work on object-oriented parallelism which has influenced ours, a large body of work in the literature can be categorized as "object-parallelism," where *objects* are mapped to *processes* that are driven by *messages*. If a message is sent in between two processes residing on two different processors, this message will be implemented via inter-processor communication. Examples of parallel C++ projects using this paradigm include the Mentat Run-time System [17], Concurrent Aggregates (CA) [10] by Dally et al., and VDOM by [14]. Our use of object-orientation is for structuring the dynamic trees, which is entirely distinct in philosophy from that of object-parallelism. Applying these ideas on dynamic tree structures, we reported abstractions of adaptive load balancing mechanisms and complex, many-to-many communications as C++ classes for supporting tree-based scientific computations [31].

The goal of this project is to develop a general tree framework that eases the difficult task of writing efficient parallel scientific computing codes, and our framework achieves this goal by object-oriented programming paradigm. We chose C++ as the tool language for its rich set of object-oriented features, including class inheritance, virtual functions, data encapsulation, and information hiding. The class inheritance and virtual function features enable us to write basic tree data structures and operations within the library, then let the library users to derive their own customized tree data structures. The users only have to specify *what data they want to have* by adding data into classes they inrehit from the library, and *what operations they want to do* by writing application-dependent virtual functions. The system will instaniate these customized objects, and will call these user-defined functions in a prescribed manner to perform the computation. The programming process will be much like JAVA applet programming. The users do not have to understand a lot of details about object-oriented paradigm – All they need to know is the programming interface and the functionality of subroutines they want to override. That is, the data encapsulation and information hiding allow users to access a data structure through, and only through a carefully designed interface. which is clearly defined between the data structures and the user programs. The users are completely aware what functions they have to provide, and what functions will be carried out by the library.

The framework was developed based on our previous CM-5 implementations [8, 15, 27, 30], in which we developed sound techniques that have been carefully studied both experimentally [29] and mathematically [28]. We will demonstrate the power of the framework by two real applications. We show that the framework helps in reducing development time (from months down to a few hours) and code sizes (from more than ten thousand lines down to just a few hundreds) for the application programs, and deliver competitive performance as well.

Finally, we want to emphasize that this library will be extremely useful when dynamic tree becomes a part of any high performance parallel language (e.g. HPC++). We believe that by developing this tree library, in parallel with the development of high performance programming languages, we can provide immediately available experiences and technology should we decide to integrate dynamic trees into any parallel languages. In the mean time, we can still provide scientific application programmers a useful weapon to combat the difficulties of developing parallel software.

The remainder of this paper is organized as follows. In Section 2 we will focus on the N-body problem, an important simulation problem that can be solved by using tree structure, e.g. the Barnes and Hut's algorithm. Section 3 briefly describes our previous parallel N-body astrophysics code implemented on Connection Machine CM-5 using Barnes and Hut's algorithm, Section 4 describes the class hierarchy in the parallel tree framework, Section 5 reports some experimental results on a network of SUN Ultra workstations, and Section 6 concludes.

2 N-body problem and Barnes-Hut algorithm

Computational methods to track the motions of bodies which interact with one another have been the subject of extensive research for centuries. So-called "N-body" methods have been applied to problems in astrophysics, semiconductor device simulation, molecular dynamics, plasma physics, and fluid mechanics.

The problem can be simply stated as follows. Given the initial states of N bodies, compute their interactions according to the underlining physic laws, usually described by a partial differential equation, and derive their final states at time T. The common and simplest approach is to iterate over a sequence of small time steps. Within each time step the change of state on a single body can be directly computed by summing the effects induced by each of the other $N-1$ bodies. While this method is conceptually simple, vectorizes well, and is the algorithm of choice for small problems, its $O(N^2)$ arithmetic complexity rules it out for large-scale simulations involving millions of particles.

Beginning with Appel [2] and Barnes and Hut [6], there has been a flurry of interest in faster algorithms. Greengard and Rokhlin [16] developed the fast multipole method with $O(N)$ arithmetic complexity under uniform particle distribution. Sundaram [39] subsequently extended this method to allow different bodies to be updated at different rates. Thus far, however, because of the complexity and overheads in the fully adaptive three dimensional multipole method, the algorithm of Barnes and Hut continues to enjoy application in astrophysical simulations. Parallel implementations of Barnes-Hut's algorithms are described in [35, 37, 38, 40, 41], and parallel fast multipole implementations include [20, 21, 33, 37, 44].

All these N-body algorithms explore 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 powe series. The Barnes-Hut algorithm uses a single-term, center-of-mass approximation. To apply the approximation effectively, these so called "tree codes" organize the bodies into a hierarchy tree in which a particle can easily find the appropriate clusters for approximation purpose. We will describe this tree structure in details later in the discussion of Barnes and Hut's algorithm, upon which our implementation is based.

We will focus on the Barnes-Hut algorithm as an example of N-body tree code. The Barnes-Hut algorithm proceeds by first computing an oct-tree partition of the three-dimensional 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 undivided box contains exactly one particle[1]. An example of such a recursive partition in two dimensions and the corresponding BH-tree are shown in Figure 1. Note that 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 in one phase up the tree, starting from the leaves.

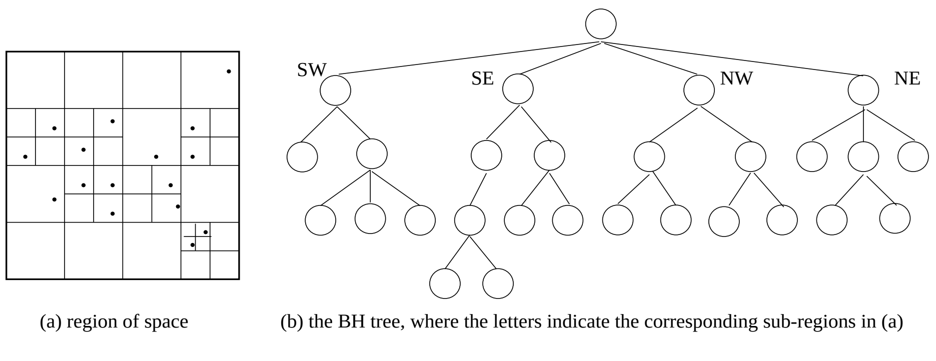
SW
SE
NW
NE
(a) region of space
(b) the BH tree, where the letters indicate the corresponding sub-regions in (a)

Figure 1: BH tree decomposition.

To compute accelerations, we loop over the set of particles observing the following rules.

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

Each particle starts at the root of the BH-tree, and traverses down the tree trying to find clusters that it can apply center-of-mass approximation. If the distance between the particle and the cluster is far enough, with respect to the radius of the cluster, then the acceleration due to that cluster is approximated by a single interaction between the particle and a point mass located at the center-of-mass of the cluster. Otherwise the particle visits each of the children of the cluster. Formally, if the distance between a particle and a cluster is more than RADIUS (cluster)/θ, then we will approximate the effect of that cluster as a point mass. We can adjust the value of θ to balance the approximation error and the execution time. Note that nodes visited in the traversal form a sub-tree of the entire BH-tree and different particles will, in general, traverse different subtrees. The leaves of the subtree traversed by a particle will be called *essential data* for the particle because it needs these nodes for interaction computation.

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, can now be repeated for the desired number of time steps.

3 Parallel Implementation

In the following subsections, we point out the differences between our parallel implementations [8, 15, 29, 30] and the generic sequential Barnes-Hut algorithm.

3.1 Data partitioning

The default strategy that we use to distribute bodies among processors is *orthogonal recursive bisection* (ORB). The space bounding all the bodies is recursively partitioned into as many boxes as there are processors and all bodies within a box are assigned to one processor. Each separator divides the workload within the region equally. When the number of processors is not a power of two, it is a trivial matter to adjust the division at each step accordingly. The ORB decomposition can be represented by a binary tree, which is stored in every processor. The ORB tree is used as a map that locates points in space to processors.

We chose ORB decomposition for several reasons. First, it provides a simple way to decompose space among processors, and a way to quickly map points in space to processors. Secondly, ORB preserves data locality reasonably well and permits simple load-balancing. Thus, while it is expensive to recompute the ORB at each time step [37], the cost of incremental load-balancing is negligible from our experience [29].

The ORB decomposition is incrementally updated in parallel as follows. The ORB tree structure is statically partitioned among processors as follows: each leaf represents a processor and each internal node is stored at the same processor as its left child. At the end of a time step each processor computes the total number of interactions used to update the state of its particles. 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 nodes at its level by a small, fixed percentage, say 5%. It is relatively simple to mark those internal nodes that are not overloaded but one of whose children is overloaded; call such a node an initiator. Only the processors within the corresponding subtree participate in balancing the load for the region of space associated with the initiator. The subtrees for different initiators are disjoint so that non-overlapping regions can be balanced in parallel.

At each step of the load-balancing step it is necessary to move bodies from the overloaded child to the non-overloaded child. This involves computing a new separating hyperplane; we use a binary search combined with a tree traversal on the local BH-tree to determine the total weight within a parallelpiped.

3.2 Managing a virtual global tree

We chose to construct a representation of a distributed global BH-tree because we wanted to investigate abstractions that allow the programmer to use a global data structure without having to worry about the details of distributed-memory implementation. For this reason we separated the construction of the tree from the details of later stages of the algorithm. This has proven to be extremely helpful in our framework implementation.

We construct the BH tree as follows. Each processor first builds a local BH-tree for the bodies within its domain. At the end of this stage, the local trees will not, in general, be structurally coherent. The next step is to make the local trees structurally coherent with the global BH-tree by adjusting the levels of all leaves that are split by ORB bisectors. Figure 2 shows the construction of a global BH tree on two processors.

A similar process was developed independently in [37]. The approach in [37] is to have each processor of a shared-memory machine (Stanford DASH machine) to build a local tree like we did, then let each processor traverse its local tree, and compare its local tree nodes with a global tree to determine whether a body is in its correct level. The processors have to lock shared global tree nodes to gain exclusive access to avoid race condition [37]. Our approach [7, 29] is for distributed memory environment and each processor sends out queries about the area where its leaves overlap with other processors' domains, and receives answers from these overlapping processors to determine how far a particle should be pushed downward.

Once level-adjustment is complete, each processor computes the centers-of-mass on its local tree without any communication. Next, each processor sends its contribution to an internal node to the owner of the node, defined as the processor whose domain contains the center of the internal node. Once the transmitted data have been combined by the receiving processors, the construction of the global BH-tree is complete.

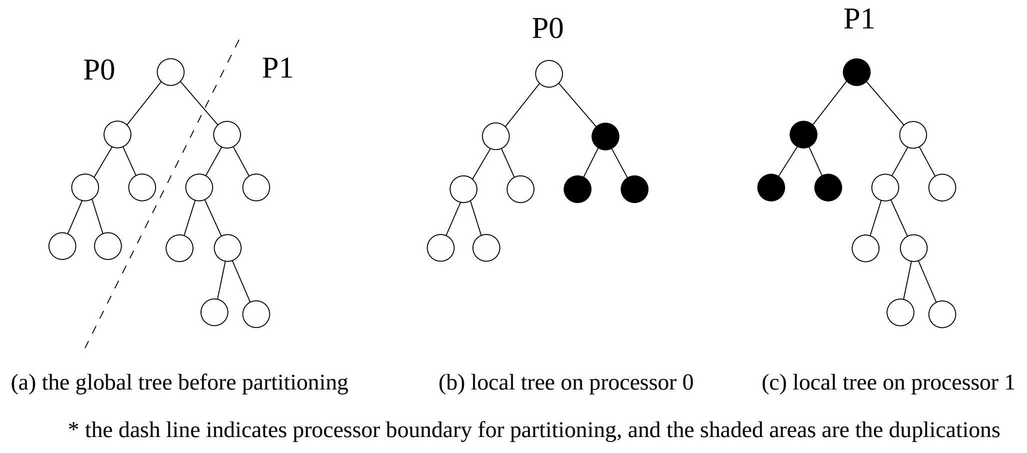
P0
P1
P0
P1
(a) the global tree before partitioning
(b) local tree on processor 0
(c) local tree on processor 1
* the dash line indicates processor boundary for partitioning, and the shaded areas are the duplications

Figure 2: Mapping a global BH tree to two processors

3.3 Collecting essential data

Once the global BH-tree has been constructed it is possible to start calculating accelerations. The naive strategy of traversing the tree and transmitting data-on-demand has several drawbacks: (1) it involves two-way communication, (2) the messages are fine-grain so that either the communication overhead is prohibitive or the programming complexity goes up, and (3) processors can spend substantial time requesting data for BH-nodes that do not exist.

It is significantly easier and faster for a processor to first collect all the essential data for its local particles, then compute the interactions the same way as in the sequential Barnes-Hut method since all the essential data are now available. In other words, the owner of a data must determine where its data might be essential, and send the data there. Formally, for every BH-node α, the owner of α computes an annular region called *influence ring* for α such that those particles α is essential to must reside within α's influence ring. Those particles that are not within the influence ring are either too close to α to apply center-of-mass approximation, or far away enough to use α's parent's information. With the ORB map it is straightforward to locate the destination processors to which α might be essential. Each processor first collects all the information deemed essential to other nodes, and then sends long messages directly to the appropriate destinations. Once all processors have received and inserted the essential data into the local trees, all the essential data are available.

3.4 Communication

The communication phases can all be abstracted as an "all-to-some" problem, in which each processor sends a set of personalized messages to dynamically determined destination processors. Therefore, the communication pattern is irregular and dynamically changing. For example, level adjustment is implemented as two separate all-to-some communication phases. The phase for collecting essential data uses one all-to-some communication.

The first issue is detecting termination: when does a processor know that all messages have been sent and received? The naive method of acknowledging receipt of every message, and having a leader count the numbers of messages sent and received within the system, proved inefficient.

A better method is to use a series of global reductions to first compute the number of messages destined for each processor. After this the send/receive protocol begins; when a processor has received the promised number of messages, it is ready to synchronize for the next phase. We noticed that the communication throughput varied with the sequence in which messages were sent and received. As an extreme example, if all messages are sent before any is received, a large machine will simply crash when the number of communication channels has been exhausted.

Instead, we used a randomized protocol to solve the all-to-some communication problem. The protocol alternates sends with receives to avoid exhausting communication channels reserved for messages that are sent but not yet received, and randomly permutes the destination so that any processor will not be flooded by incoming messages at any given time.

This approach was motivated by the observation that the communication throughput varied with the sequence in which messages were sent and received. For example, if all the processors sends messages in processor id order, then those processors with smaller processor ids will be flooded with messages at the beginning of the communication. A better way is to randomly permute the order in which the messages are sent, so that messages going to the same destinations are spread out in the time spectrum. In an earlier paper [28] we developed the atomic message model to investigate message passing efficiency. Consistent with the theory, we find that sending messages in random order worked best. Figure 3 outlines the randomized communication protocol used in the library.

```
all_to_some_communication
{
     generate all messages;
     compute the number of incoming messages;
     while there is message to send/receive
          if there is incoming message
               receive incoming message;
          pick a random destination d;
          if there is message to send to d and resource to send it
               send the message to d;
     endloop
}
```

Figure 3: The Communication Protocol

3.5 Summary

Figure 4 gives a high-level description of the parallel implementation structure. Note that the local trees are built only at the start of the first time step.

```
Build local BH trees.

For every time step do:

  1. Construct the BH-tree representation

     (a) Adjust node levels

     (b) Compute partial node values on local trees

     (c) Combine partial node values at owning processors

  2. Owners send essential data

  3. Calculate accelerations

  4. Update velocities and positions of bodies

  5. Update local BH-trees incrementally

  6. If the workload is not balanced update the ORB incrementally
```

Figure 4: Outline of code structure

4 A Dynamic Tree Framework

The parallel N-body framework defines three layers of C++ classes: *generic tree layer*, *Barnes-Hut tree layer*, and *application layer*. Each latter layer is built on top of the former layer. The *generic tree layer* supports simple tree construction and manipulation methods. System programmers can build special tree libraries using classes in the *generic tree layer*. For example, we have built a *Barnes-Hut tree layer* using the *generic tree layer* (Sec 4.2). The application programmer can write application programs using classes in the *Barnes-Hut tree layer*, or any other special library developed from the *generic tree layer*. We will demonstrate the power of this framework by writing a gravitational N-body code and a multiple-filament vortex simulation code (Sec 4.3). Figure 5 illustrates the class hierarchy in these three layers.

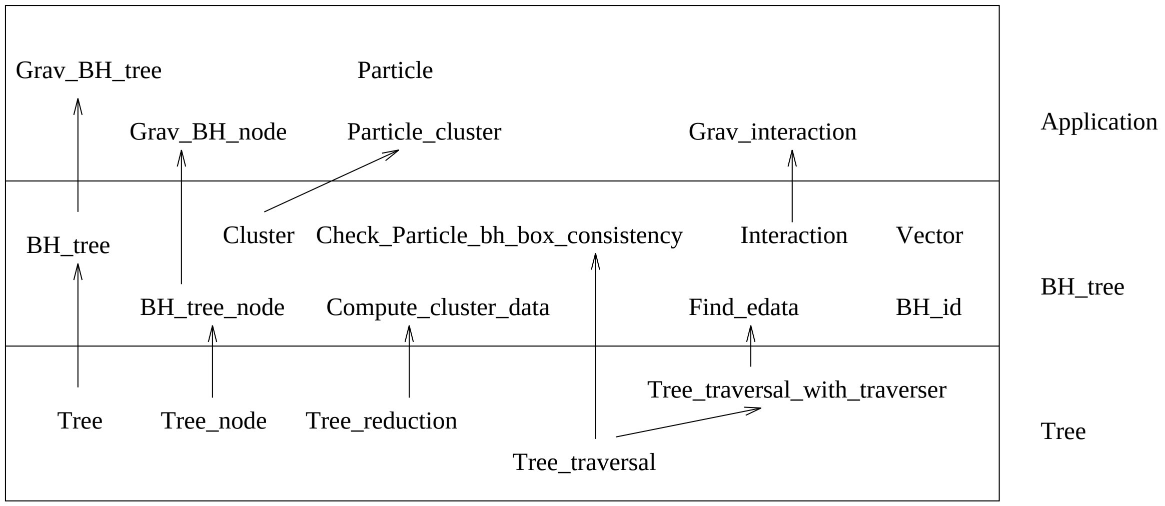

Figure 5: The class hierarchy in *generic tree*, *Barnes-Hut tree*, and *application* layers.

4.1 Generic tree layer

The *generic tree layer* is the foundation of our framework from which complex tree structures can be derived. The class `Tree` serves as a container class in which every tree node has a pointer to data of the given data type. The desired data type is given as a template parameter, along with the maximum number of children one tree node can have.

We define basic tree manipulation methods in the generic tree layer, including inserting a new child from a leaf, deleting an existing leaf, and performing tree reduction and traversal. We keep the interface simple by restricting all the deletion/insertion to the leaves and let the `Tree` class user take care of more sophisticated and specific tree structure updating.

We have also implemented two tree operations – *reduction* and *traversal*, as special classes. Objects instantiated from the `reduction` class compute the data of a tree node according to the data of its children, e.g. computing the center of mass in Barnes-Hut's algorithm. Objects instantiated from the `traversal` class walk over the tree nodes, and perform a user-defined operation (denoted as *per node function*) on each tree node (Figure 6).

The tree reduction/traversal operations were implemented in an application-independent manner. Both operations are implemented as class templates so that users can supply tree and tree node type for a customized tree reduction/traversal operations. For tree reduction,

```
template <class Data, const int n_children>
class Tree_node {
protected:
  Data *data;
  Tree_node *children[n_children];
};
template <class Data, class Tree_node, class Tree,
          const int n_children>
class Tree_reduction {
public:
  virtual void init(Data*) = 0;
  virtual void combine(Data *parent, Data* child) = 0;
  void reduction(Tree* tree);
};
template <class Data, class Tree_node, class Tree,
          const int n_children, class Node_id>
class Tree_traversal {
public:
  virtual bool process(Data*) = 0;
  void traverse(Tree *tree);
};
template <class Data, class Tree_node, class Tree,
          class Traverser>
class Tree_traversal_with_traverser :
public Tree_traversal<Data,Tree_node,Tree,N_CHILD,BH_id>
{
protected:
  Traverser *traverser;         // who is traversing?
};
```

Figure 6: Generic tree and reduction/traversal classes.

users are required to provide two functions: `init(Data*)` and `combine(Data *parent, Data* child)`, which tell `reduction` class how to initialize and combine the data in tree nodes, respectively. The class `Data` is the data type stored in each node of the tree on which the reduction operation is to be performed. For tree traversal, users are required to provide the per node function `bool process(Data*)` that is to be performed on every tree node. The boolean return value indicates whether the traversal should continue further down the tree. By separating the application code from the tree reduction/traversal classes, these operations become application-independent.

4.2 Barnes-Hut tree

On top of *generic tree* layer we build a layer called `BH_tree`. This layer supports tree operations required in most of the N-body tree algorithms – it supports tree operations common to both BH algorithm and fast multipole method, and all the special operations used in the Barnes-Hut method.

By extending the `Tree` class, each tree node in `BH_tree` contains a data cluster, and the data cluster of each leaf node contains a list of bodies.[2] The types of the particle and cluster are given by the user of the `BH_tree` class as template parameters `AppCluster` and `AppBody`. This abstraction captures the structure of a BH tree without any application-specific details.

[2]Recall that each leaf may have more than one particle.

```
template<class AppBody>
class Cluster {
protected:  Link_list<AppBody*> body_list;
public:  void add(AppBody* b);
};
template<class AppCluster, class AppBody>
class BH_tree : public Tree<AppCluster, N_CHILD> {
 public:
  void insert_body(AppBody*);
  void remove_body(AppBody*, Tree_node<AppCluster, N_CHILD>*);
};
template<class AppCluster, class AppBody, class Tree_node,
         class Tree, const int n_children>
class Compute_cluster_data: public
      Tree_reduction<AppCluster, Tree_node, Tree, n_children>{
 public:
  void init(AppCluster* cluster) {
    cluster->reset_data();
    if (cluster->get_type() == Leaf)
      for (every body in cluster's body_list)
        cluster->add_body(body); }
  void combine(AppCluster* parent, AppCluster* child)
    {parent->add_cluster(child);}
};
```

Figure 7: BH tree layer classes.

The `BH_tree` class also supports several operations: computing cluster data, finding essential data, computing interaction, and checking particle and BH box for consistency. Most of these methods can be reused in implementing the fast multipole method.

Cluster data computation is implemented as a tree reduction (Figure 7). `init(AppCluster* cluster)` resets the data in the cluster and if the cluster is a leaf, it combines the data of the bodies from the body list into the data of the cluster. The other function `combine(AppCluster* parent, AppCluster* child)` adds children's data to parent's. By defining the actual computation as a method of the cluster, the reduction class is independent of the way the data are combined in the application.

The essential data finding class `Find_edata` inherits `Tree_traversal_with_traverser` with two additional lists for essential clusters and bodies (Figure 8). The *traverser* is the particle that collects essential data. The per node function `process-(AppCluster*)` inserts the clusters that can be approximated into `essential_clusters` list, and adds the bodies from leaf clusters that cannot be approximated into `essential_bodies` list. The traversal continues only when traverser cannot apply approximation to an internal cluster.

After collecting the essential clusters and bodies, a body can start computing the interactions. We implemented the interaction computation in an application-independent manner. The computation class `Interaction` (Figure 8) goes through the essential data list[3] and calls for functions to compute body-to-body and body-to-cluster interactions defined by the user of `Interaction`.

After bodies are moved to their new positions, they may not be in their original BH boxes. Therefore, the tree structure must be modified so that it becomes consistent with the new

[3]Lists obtained from the class `Find_Edata`.

```
template<class AppCluster, class AppBody, class Tree_node, class Tree>
class Find_edata: public Tree_traversal_with_traverser
                  <AppCluster,Tree_node,Tree,AppBody> {
  Link_list<AppBody*> essential_bodies;
  Link_list<AppCluster*> essential_clusters;
public:
  bool process(AppCluster* c) {
    if (c->is_edata_for(traverser)) {
      essential_clusters.insert(c); return(0);
    } else if (c->get_type() == Leaf) {
      for (every body in c's body list)
        if (body != traverser)
          essential_bodies.insert(body);
      return(0);
    } return(1);  }
};
template<class AppBody, class AppCluster, class Result>
class Interaction {
  AppBody *subject;
  Link_list<AppBody*>* body_list;
  Link_list<AppCluster*>* cluster_list;
  Result result;
public:
  void compute() {
    result.reset();
    for (every body in body_list)
      result += body_body_interaction(subject, body);
    for (every cluster in cluster_list)
      result += body_cluster_interaction(subject,cluster);}
  virtual Result body_body_interaction(AppBody*,AppBody*)=0;
  virtual Result body_cluster_interaction(AppBody*,AppCluster*)=0;
};
```

Figure 8: Class for finding essential data and interaction computation.

particle positions again. We implemented this as a tree traversal class `Check_particle_bh_box_consistency`, which collects bodies that wandered off their BH boxes, followed by a series of insertion/deletion tree operations. This function is universally useful for all tree code because the dynamic tree structure is expensive to rebuild, but relatively cheap to patch up.

4.3 Application Layer

We now show an example of application development using the tree framework – a gravitational N-body application built on top of the `BH_tree` layer. First we construct a class `Particle` for bodies that attract one another by gravity, then we build the cluster type `Particle_cluster` from `Particle` (Figure 9). Next, in the `Particle_cluster` class we define the methods for computing/combining center of mass and the methods for testing essential data.

Then, in class `Grav_interaction`, which is derived from the class template `Interaction`, we define methods to compute gravitational interactions. We specify the gravitation interaction rules in the definition of `body_body_interaction` and `body_cluster_interaction`.

Finally, we define the BH-tree type `Grav_BH_tree` and tree node type `Grav_BH_node`.

```
class Particle {
protected:
  Real mass;
  Vector position;
  Vector velocity;
};
class Particle_cluster: public Cluster<Particle> {
protected:
  Center_of_mass center_of_mass;
public:
  void reset_data();   // center of mass computation
  void add_body(Particle *p);
  void add_cluster(Particle_cluster* child);
  bool is_edata_for(Particle*);   // find essential data
};
class Grav_interaction:
public Interaction<Particle, Particle_cluster, Vector> {
public:
  Vector body_body_interaction(Particle*, Particle*);
  Vector body_cluster_interact(Particle*,Particle_cluster*);
};
typedef Tree_node<Particle_cluster, N_CHILD> Grav_BH_node;
typedef BH_tree<Particle_cluster, Particle> Grav_BH_tree;
```

Figure 9: Classes for a gravitational N-body application.

These two data types serve as template parameters to instantiate BH-tree related operations, like `Compute_cluster_data`, `Find_edata`, and `Check_particle_bh_box_consistency`. Figure 10 shows the code segment for the simulation.

4.4 Parallel implementation

Using only the class libraries provided in the three layers described in previous subsections, we could model N-body simulations on uniprocessors. For parallel execution of programs, we require additional abstractions for parallelism.

In our current implementation, we assume SPMD (single program multiple data) model for parallel computation. Under this model, we would require abstractions for data mapping and interprocessor communication. We have designed two groups of classes for this purpose – `Mapper` classes that are responsible for defining the geometry of the tree structure, and `Communicator` classes that provide all-to-some communications that are common in N-body simulations.

Note that although using the same name, our Communicator class is quite different from the communication package in [11]. Our Communicator is a C++ class with a high level communication protocol. Therefore, the only optimizations we perform is message aggregation and random destination permutation (see Sec 3.4 for details), and leave all the other optimization to MPI library. Whereas the Communicator in [11] is a low level optimizer that will remove redundant communication, combining separate communications, and perform communication pipelining automatically.

```
void simulation_step(Grav_BH_tree *bh_tree, Link_list<Particle*>* p_list) {
  Compute_cluster_data<Particle_cluster, Particle, Grav_BH_node,
                      Grav_BH_tree, N_CHILD> compute_com;
  compute_com.reduction(bh_tree);
  for (every particle in p_list) {
    Find_edata<Particle_cluster, Particle, Grav_BH_node, Grav_BH_tree>
      find_edata(particle);
    find_edata.traverse(bh_tree);
    Grav_compute_interaction interaction(particle,
      find_edata.get_essential_bodies(), find_edata.get_essential_clusters());
    interaction.compute();
    // update particle positions & velocity according to the results
    // from interaction.
  }
  Check_particle_bh_box_consistency<Particle_cluster, Particle, Grav_BH_node,
  Grav_BH_tree> check_particle_bh_box_consistency;
  check_particle_bh_box_consistency.traverse(bh_tree);
  // move the out of box particle to correct BH bode.
}
```

Figure 10: One simulation step

Mapper classes

The Mapper classes define the geometry of data structures (e.g. BH trees in N-body simulations). Over the course of a simulation, Mapper objects are created during the construction of data structure objects (e.g. BH tree objects). When created, a Mapper object invokes the data partitioning function specified by the user or performs default behavior when no partitioning strategy is specified, it then gathers and caches geometry information from the partitioning function. In later stage of a simulation, the Mappers mediate object operations that require interprocessor communication.

```
template <class Data, class DataSet, class ProcessorDomain,
          class MappingTable>
class Mapper {
 protected:
   MappingTable table;
 public:
   virtual ProcessorDomain data_to_processor(Data*)=0;
   virtual Link_list<ProcessorDomain>
             dataset_to_processors(DataSet*)=0;
};
```

Figure 11: The Mapper class.

In our previous parallel C implementation, we constructed an ORB partitioner and two associated geometry resolution functions: data_to_processor (that translates a data coordinate to a processor domain) and dataset_to_processors (that translates a rectangular box, which contains multiple data, to a set of processor domains). In addition, we defined a simple data structure MappingTable to store the ORB map. These data and methods have been integrated into the Mapper classes in our parallel framework. Figure 11 outlines the interface of Mapper. As part of this research effort, we are also extending the Mapper class to incorporate a number of commonly used partitioning strategies and user-defined mapping methods.

Communicator classes

The `Communicator` classes support general-purpose all-to-some communications for N-body tree codes. A `Communicator` class defines two functions: `extract` (that, when given a data pointer, constructs an outgoing data) and `process` (that processes each incoming data). When a `communicator` is constructed, it goes over the list of data pointers, calls `extract` to build outgoing data, packs many outgoing data into actual messages, sends/receives all the messages according to the communication protocol, and finally unpacks messages and calls `process` to perform appropriate actions. Figure 12 outlines the interface of `Communicator`.

```
template <class Data, class DataPacket>
class Communicator {
 protected:
   Link_list<Data*> *data_list[MAX_NUM_PROCESSORS];
   DataPacket send_buffer[MAX_BUFFER_SIZE];
   DataPacket receive_buffer[MAX_BUFFER_SIZE];
 public:
   void communication_protocol();
   virtual DataPacket extract(Data*)=0;
   virtual process(DataPacket*)=0;
};
```

Figure 12: The Communicator class.

The technique we developed for `communicator` has proven to be both efficient and general enough to support all-to-some communication in N-body tree codes. For instance, the essential data gathering was implemented as a tree traversal followed by a communicator phase. The tree traversal goes over the BH nodes, computes the proper destination set where the tree node might be essential, and appends its address to a pointer list to that destination. Each destination processor will have a separate pointer list that contains the addresses of those tree nodes that might be essential to the destination's local particles. The `extract` routine assures that only essential parts of a tree node are transmitted. The `process` routine inserts incoming data into the local tree. All the message packing/unpacking/transmission are handled by `communicator`.

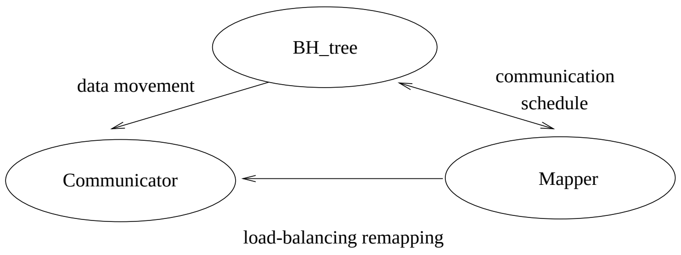

Figure 13: Interaction of classes in the dynamic tree framework.

Figure 13 depicts the interactions between the `BH_tree` class, the `Communicator` class, and the `Mapper` class.

5 Experimental Study

We demonstrate the flexibility of the parallel tree framework by implementing two applications – a gravitational force field computation and a multi-filament fluid dynamic calculation. Both applications were developed within the tree library framework; therefore, all the tree structure details and communications were taken care of by predefined tree operations and the communicator classes.

Our previous implementation of the gravitational computation on the CM-5 contained more than 12000 lines of C code and took more than 6 months to develop. Using the tree framework we have developed, the code size is reduced to just a few hundred lines. Furthermore, most of the gravitational code can be reused to develop the vortex method computation, as we will show in the following sections. As a result, it took us only a few hours to develop the vortex code.

The experiments were conducted on four UltraSPARC-II workstations located in the Institute of Information Science, Academia Sinica. The workstations arde connected by a fast Ethernet network capable of 100M bps per node. Each workstation has 128 mega bytes of memory and runs SUNOS 5.5.1. The communication library in the framework is implemented on top of MPI (mpich version 1.0.4 [3]).

Gravitational force field calculation

In the gravitational force field we solve the following familiar Newton's law of motion. Note that in Equation 5 we use the standard potential softening parameter ϵ to remove the singular point when $|\vec{x_i} - \vec{x_j}|$ is zero.

$$\frac{d^2\vec{x_i}}{d^2t} = \sum_j \frac{Gm_i m_j(\vec{x_i} - \vec{x_j})^3}{r(r^2 + \epsilon^2)} \tag{1}$$

As a basis for comparison, the direct method (which implements the $O(N^2)$ algorithm) on a single workstation took 20 seconds for 4000 particles, 85 seconds for 8000 particles, and 384 seconds for 16k particles.

Table 1 summarizes the speedup factors of our parallel implementation on a cluster of four workstations. To get fair speedup numbers, we compare the parallel execution time with the timing from a highly optimized sequential C code, implementing the same algorithm, written by Barnes and Hut. The input configuration is a set of uniformly distributed particles in three dimension. The C code uses various techniques including in-memory caching of the vector $\vec{x_i} - \vec{x_j}$ between determining whether to open a cell (traverse down the tree for smaller subcluster) and the actual evaluation of the potential filed and acceleration. Nevertheless, our implementation gives competitive performance, even compared with this highly optimized C code.

Multiple-filament vortex simulation

Many flows can be simulated by computing the evolution in time of corticity using the Biot-Savart law (Equation 2). Biot-Savart models offer the advantage that the calculation effort concentrates in the small regions of the fluid that contain the corticity, which are usually small compared to the domain that contains the flow[25]. This not only reduces considerably

problem size	8k	16k	24k	32k	40k	48k	56k	64k	128k	256k
sequential time	14.39	19.43	30.47	41.73	54.60	65.62	81.23	93.59	186.12	413.75
parallel time	5.84	6.28	9.80	13.25	17.50	21.03	26.17	29.17	58.78	125.78
speedup	2.46	3.10	3.11	3.15	3.12	3.12	3.12	3.17	3.12	3.23

Table 1: Timing comparison between the parallel C++ code using the framework and a sequential C implementation for gravitation field computation. Time units are seconds

the comutational expense, but allows better resolution for high Reynolds number flows. Biot-Savart models also allow us to perform effectively inviscid computations with no accumulation of numerical diffsion [1, 25, 26].

$$\frac{d\vec{x}_i}{dt} = -\frac{1}{4\pi}\sum_j \frac{(\vec{x}_i - \vec{x}_j) \times \Delta\vec{x}_j}{|\vec{x}_i - \vec{x}_j|}(1 - e^{-\frac{|\vec{x}_i - \vec{x}_j|^3}{\delta_j^3}})$$
$$\Delta\vec{x}_j = \frac{1}{2}(\vec{x}_{j+1} - \vec{x}_{j-1}) \qquad (2)$$

Vortex methods are also approiate for studying many complex flows that present coherent vortex structures. These coherent structures frequently are closely interacting tube-like vortices. Some of the flows of practical interest that present interacting tube-like vortex structures are wing tip vortices [12], a variety of 3D jet flows of different shapes [18, 19] and turbulent flows [13]. A generic type of interaction between vortex tubes is collapse and reconnection, which is the close approach of vortex tubes that results in large vorticity and strain-rate amplification [22]. Collapse an reconnection is an example of small scale formation in high Reynolds number flows. Small scale generation is very important for the energy transfer and dissipation processes in turbulent flows [32].

We implemented a multi-filament vortex method using our framework. We solve Biot-Savart's interaction between vortex elements using the algorithm by Knio and Ghoniem [23]. The multiple-filament vortex method computes the vorticity on each particle, and requires an extra phase in the tree construction to compute the vorticity. The vorticity of a particle is defined as the displacement of its two neighbors in the filament (see Equation 2). Once the vorticity on each particle is computed, we can compute the multipole moments on the local trees. Finally, each processor sends its contribution to a node to the owner of the node so that individual contributions are combined into globally correct information, as in the gravitational case.

Figure 14 gives a high-level description of the code structure. The code is very similar to the gravitational force field code since most of the data structure and communication issues are taken care of by the BH tree library. The major differences between the two applications are the data stored in each tree node, the way data in child nodes are merged into the parent node, and the interaction rules for the particles.

Figure 15 compares the two particle_cluster classes in these two applications. They both inherit the abstract class `Cluster` in the BH tree layer and take part in the tree reduction computation. Note that it is the application programmer's responsibility to decide what data should be kept in a cluster (e.g `Center_of_mass` in the gravitation code). In addition, the methods `add_body` and `add_cluster` are virtual methods defined in class `Cluster` in the BH

```
0. build local BH trees
   for every time step do:
1.  construct the BH-tree representation
1.1  adjust node levels
1.2  compute partial node values on local trees
1.3  combine partial node values at owning processors
2.  owners send essential data
3.  calculate induced velocity
4.  update velocities and positions of
    bodies
5.  update incremental data structures (local BH-tree and filament-trees)
6.  if the workload is not balanced
      update the ORB incrementally
   enddo
```

Figure 14: Outline of code structure for the vortex simulation

tree layer, and will be called when the actual reduction of tree data takes place. After defining the data in the cluster, the application progammer should also defines how to combine the data from children nodes in these two methods.

Figure 16 illustrates the two computation classes in these two applications. Both computations inherit the abstract class `Interaction` with three parameters, the particle type, the cluster type, and the type of the result from the computation. For example, in the gravitation code the parameters will be `Particle`, `Particle_cluster`, and `Vector` (for the computed acceleration) from Figure 9. The abstract class `Interaction` has a method `compute` that goes through a list of particles and another list of clusters, and computes their interactions with a given particle by calling `body_body_interaction` and `body_cluster_interaction`. The application programmer defines the interaction rules in these two functions, which in turn computes the results of the given data type. Note that we compute the particle-cluster interactions to the dipole terms in the fluid dynamic code and to the monopole term in the gravitational computation.

Table 2 summarizes the timing comparison between our parallel code and a sequential C code modified from the previously mentioned Barnes and Hut's implementation, which is highly optimized. Compared to the parallel implementation using our tree framework, it took us significant amount of efforts to convert the C program to the one that computes fluid dynamics in the vortex method. The main reason is that we had to trace almost the entire program to understand the data structures and the control flow before we could modify them. On the other hand, using the tree framework, it only required redefining a few functions to complete the parallel program, which resulted in total development time of just a few hours.

The fluid dynamics code developed using the tree framework also delivered competitive performance. The speedup factors are higher than those of the gravitation code, because the fluid dynamics code performs more computation on each particle, which amortizes the overhead of parallelization and object orientation.

In both applications the major overhead in the C++ version is in the essential data collection process. Our implementation collects all the essential data and put them in a linked list,

```
// Particle class used in the gravitational computation
class Particle_cluster: public Cluster<Particle>
{
protected:
  Center_of_mass center_of_mass;
public:
  // center of mass computation
  void reset_data() {center_of_mass.reset();}
  void add_body(Particle *p) {
    center_of_mass += Center_of_mass(p->get_mass(),p->get_position());
  }
  void add_cluster(Particle_cluster* child) {
    center_of_mass += child->get_com();
  }
};

// Particle cluster class in the fluid code
class Particle_cluster: public Cluster<Particle>
{
protected:
  Real delta;
  Vector alpha;
  Vector moments[DIMENSION];
public:
  // moments computation
  void reset_data() {
    int i;
    alpha.reset();
    delta = 0.0;
    for (i = 0; i < DIMENSION; i++)
      moments[i].reset();
  }
  void add_body(Particle *p) {
    int i;
    Vector disp = p->get_position() - bh_id.center();
    for (i = 0; i < DIMENSION; i++)
      moments[i] += p->get_alpha() * disp.get(i);
  }
  void add_cluster(Particle_cluster* child) {
    int i;
    Vector disp = child->bh_id.center() - bh_id.center();
    for (i = 0; i < DIMENSION; i++)
      moments[i] += child->get_alpha()*disp.get(i)-child->moments[i];
  }
};
```

Figure 15: The Particle cluster classes used in the fluid dynamic code and the gravitational force field code. The method `add_body` and `add_cluster` add a particle or a cluster to a tree node.

```
// Interaction class in the gravitational computation
class Grav_compute_interaction:
public Interaction<Particle, Particle_cluster, Vector>
{
public:
  Vector body_body_interaction(Particle* p1, Particle* p2);
  Vector body_cluster_interaction(Particle* p, Particle_cluster* c) {
    Vector acc, dr;
    Real dr2;
    n_interaction++;
    dr = c->get_com().get_position() - p->get_position();
    dr2 = (dr * dr) + parameter.eps * parameter.eps;
    acc = dr * (c->get_com().get_mass() / (sqrt(dr2) * dr2));
    return(acc);
  }
};

// Interaction class in the fluid dynamics code
class Fluid_compute_interaction:
public Interaction<Particle, Particle_cluster, Vector>
{
public:
  Vector body_body_interaction(Particle* p1, Particle* p2);
  Vector body_cluster_interaction(Particle* p, Particle_cluster* c) {
    dr = c->get_bh_id().center() - p->get_position();
    dr2 = dot_product(dr, dr);
    r = sqrt(dr2);
    r3 = r * dr2;
    invr3 = 1.0 / r3;
    invr4 = 1.0 / (dr2 * dr2);
    invr = 1.0 / r;

    expv = exp(-(r3/delta3));
    h = 1.0 - expv;
    induced_vel += p->get_alpha() * (dr * (h * invr3));
    hr3 = h * invr3;
    f1r = 3.0 * ((1.0 / (r * delta3) + invr4) * expv - invr4) * invr;
    for (i = 0; i < DIMENSION; i++) {
      scaled_dr = dr * (f1r * dr.get(i));
      scaled_dr.set(i, hr3);
      induced_vel += c->get_moments(i) * scaled_dr;
    }
    return(induced_vel);
  }
};
```

Figure 16: The computation classes in for the gravitational code and the fluid code.

problem size	8k	16k	24k	32k	40k	48k	56k	64k	128k	256k
sequential time	17.38	41.78	67.53	93.75	122.23	148.60	175.46	204.18	404.34	801.81
parallel time	5.51	12.81	19.77	27.84	34.96	43.00	51.37	59.05	117.20	231.07
speedup	3.15	3.26	3.42	3.38	3.46	3.42	3.42	3.46	3.45	3.47

Table 2: Timing comparison for the fluid codes. Time units are seconds. The parallel code were written using the tree framework, and the sequential code was converted from Barnes and Hut's code.

then computes the interactions one element at a time from the list. We took this approach mainly to separate the data collection process from the computation. However, we pay the overhead of allocating linked list elements through expensive dynamic memory allocation. We will improve the efficiency by a customized dynamic memory management mechanism in which we will have better control over the allocation/deallocation process. Another optimization we will do would be to compute the interactions on-the-fly while traversing the tree.

6 Conclusion

In this paper, we have presented the implementation of our framework for parallel tree-structured scientific computing. We start from the generic tree class and proceed to increasingly complex Barnes-Hut tree structures. By separating abstractions of data structures from computation details, our tree framework is applicable to other tree-based scientific simulations as well.

Our experiences with developing fast methods for gravitational simulations on the Connection Machine CM-5, and preliminary experience with vortex dynamics applications give us confidence that such a framework will be invaluable to applications scientists and engineers. For computer scientists, such a framework will also allow design effort and heavy-duty optimization to be expended exactly where it is most needed, without restricting the generality or portability of related codes.

We also showed that object-oriented programming paradigm can ease the difficult task of writing efficient parallel scientific computing codes. Using object-oriented features like class inheritance, virtual functions, data encapsulation, and information hiding, we are able to define a clear interface between library data structures and the user applications. This allows library users to safely access encapsulated data with ease, and enables the library builder to explore new implementation dimensions for greater performance, without disturbing the upper layer applications.

We have implemented a gravitational N-body code and a vortex method fluid dynamic code using the class libraries provided in this framework. We have shown that using the framework has greatly shortened the development time of these codes. Both implementations show competitive performance on a four-node workstation cluster. To further evaluate our framework, we plan to implement a number of application programs, including a molecular dynamics code and the 3-d fast multipole method, using the class libraries we have developed.

Acknowledgement

This research is supported by National Science Council of Taiwan under grant 86-2213-E-001-010, a special start-up grant from National Chung Cheng University for the first author, and in part by the Institute for Mathematics and its Applications with funds provided by the National Science Foundation.

References

[1] C. Anderson and C. Greengard. The vortex merger problem at infinite reynolds number. *Communications on pure and Applied mathematics*, XLII, 1989.

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

[3] Argonne National Labortory and University of Chicago. *MPICH, a protable implementation of MPI*, 1996.

[4] Susan Atlas, Subhankar Benerjee, Julian C. Cummings, Paul J. Hinker, M. Srikant, John V.W. Reynders, and Marydell Tholburn. Pooma: A high performance distributed simulation environment for scientific applications. In *Supercomputing95*, 1995.

[5] W. L. Bain. Aggregate distributed objects for distributed memory parallel systems. In *The 5th Distributed Memory Computing Conference, Vol. II*, pages 1050–1055, Charleston, SC, April 1990. IEEE.

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

[7] S. Bhatt, M. Chen, C. Lin, and P. Liu. Abstractions for parallel N-body simulation. In *Scalable High Performance Computing Conference SHPCC-92*, 1992.

[8] S. Bhatt, P. Liu, V. Fernadez, and N. Zabusky. Tree codes for vortex dynamics. In *International Parallel Processing Symposium*, 1995.

[9] C. M. Chase, A. L. Cheung, A. P. Reeves, and M. R. Smith. Paragon: A parallel programming environment for scientific applications using communication structures. In *1991 International Conference for Parallel Processing, Vol. II*, pages 211–218, August 1991.

[10] A. A. Chien and W. J. Dally. Concurrent aggregates (CA). In *2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 187–196, Seattle, Washington, March 1990. ACM.

[11] S.E. Choi and L.Snyder. Quantifying the effects of communication optimizations. In *Proceedings of the International Conference on Parallel Processing*, 1997.

[12] S. C. Crow. Stability theory for a pair of trailing vortices. *AIAA Journal*, 8(12), 1970.

[13] S. Douady, Y. Couder, and M.E. Brachet. Direct observation of the intermittency of intense vorticity filaments in turbulence. *Physical Review Letters*, 67(8), 1991.

[14] M. J. Feeley and H. M. Levy. Distributed shard memory with versioned objects. In *OOPSLA '92*, pages 247 – 262, Vancouver, BC, Canada, October 1992.

[15] V. Fernadez, N. Zabusky, S. Bhatt, P. Liu, and A. Gerasoulis. Filament surgery and temporal grid adaptivity extensions to a parallel tree code for simulation and diagnostics in 3d vortex dynamics. In *Second International Workshop in Vortex Flow*, 1995.

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

[17] A. Grimshaw. The mentat run-time system: Support for medium grain parallel computation. In *The 5th Distributed Memory Computing Conference, Vol. II*, pages 1064–1073, Charleston, SC, April 1990. IEEE.

[18] H.S. Husain and F. Hussain. Characteristics of unexcited and excited jets. *Journal of Fluid Mechanics*, 208, 1989.

[19] H.S. Husain and F. Hussain. Dynamics of preferred mode coherent structure. *Journal of Fluid Mechnics*, 248, 1993.

[20] L. Johnsson and Y. Hu. personal communication. 1993.

[21] J. F. Leathrum Jr. and J. Board Jr. The parallel fast multipole algorithm in three dimensions. *manuscript*, 1992.

[22] S. Kida and M. Takaoka. Vortex reconnection. *Annu. Rev. Fluid Mech.*, 26, 1994.

[23] O. M. Knio and A. F. Ghoniem. Numerical study of a three-dimensional vortex method. *Journal of Computational Physics*, 86, 1980.

[24] J. K. Lee and D. Gannon. Object oriented parallel programming experiments and results. In *Supercomputing '91*, pages 273–282, November 1991.

[25] A. Leonard. Computing three-dimensional incompressible flows with vortex elements. *Ann. Rev. Fluid Mech.*, 17, 1985.

[26] A. Leonard. Vortex methods for flow simulation. *Journal of Computational Physics*, 37, 1989.

[27] P. Liu. *The parallel implementation of N-Body algorithms.* PhD thesis, Yale University, 1994.

[28] P. Liu, W. Aiello, and S. Bhatt. An atomic model for message passing. In *5th Annual ACM Symposium on Parallel Algorithms and Architecture*, 1993.

[29] P. Liu and S. Bhatt. Experiences with parallel n-body simulation. In *6th Annual ACM Symposium on Parallel Algorithms and Architecture*, 1994.

[30] P. Liu and S. Bhatt. A framework for parallel n-body simulations. In *Third International Conference on Computational Physics*, 1995.

[31] P. Liu and J. Wu. A framework for parallel tree-based scientific simulation. In *26th International Conference on Parallel Processing*, 1997.

[32] A. J. Majda. Vorticity, turbulence and acoustics in fluid flow. *SIAM Review.*, 33, 1991.

[33] L. Nyland, J. Prins, and J. Reif. A data-parallel implementation of the adaptive fast multipole algorithm. In *DAGS/PC Symposium*, 1993.

[34] R. Parsons and D. Quinlan. A++/p++ array classes for architecture independent finite difference calculations. In *Proceedings of the Second Annual Object-Oriented Numerics Conference*, April 1994.

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

[36] J. Saltz, A. Sussman, and C. Chang. Chaos++: A runtime library to support distributed dynamic data structures. *Gregory V. Wilson, Editor, Parallel Programming Using C++*, 1995.

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

[38] J. Singh, C. Holt, T. Totsuka, A. Gupta, and J. Hennessy. Load balancing and data locality in hierarchical N-body methods. Technical Report CSL-TR-92-505, Stanford University, 1992.

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

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

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

[42] S. Johnsson Y. Hu and S.H. Tseng. High performance fortran for highly irregular problems. In *PPOPP*, 1997.

[43] 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-92*, pages 236 – 239, Williamsburg, VA, April 1992.

[44] F. Zhao and S.L. Johnsson. The parallel multipole method on the connection machine. Technical Report DCS/TR-749, Yale University, 1989.