A Framework for Parallel Tree-Based Scientific Simulations

Pangfeng Liu

Department of Computer Science
National Chung Cheng University
Chia-yi, Taiwan 62107

Jan-Jan Wu

Institute of Information Science
Academia Sinica
Taipei, Taiwan 11529

Abstract

Abstract

This paper describes an implementation of a platform-independent parallel C++ N-body framework that can support various scientific simulations that involve 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 N-body problems, and let the framework take care of the tedious and error-prone details that are common among N-body 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 N-body applications, to run on different parallel platforms, and to deliver good performance as well.

1 Introduction

1.1 N-body problem and tree codes

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 [1] and Barnes and Hut [2], there has been a flurry of interest in faster algorithms. Greengard and Rokhlin [5] developed the fast multipole method with $O(N)$ arithmetic complexity under uniform particle distribution. Sundaram [15] 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 [12, 13, 14, 16, 17], and parallel fast multipole implementations include [6, 7, 11, 13].

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 power-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, which our implementation is based upon.

1.2 N-body framework

Most of the N-body tree codes use similar tree structures and exhibit similar computation patterns. There are two levels of similarity. First, a fluid mechanics code and a molecular dynamics code may differ only in the interaction rules. The tree structures are basically the same except for the data stored in tree nodes and the implementation-dependent tree representation. Secondly, different N-body tree 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 N-body framework helps in developing tree codes for different N-body domains, and in implementing different N-body algorithms as well.

Unfortunately, all the previous N-body implementations did not consider reusability and portability – they do not separate the generic data structure from the application-dependent computation kernel, and they are built for one N-body problem on one particular machine. Therefore, it will take considerable efforts to convert an astrophysics simulation code running on one machine into a fluid dynamic 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.

In addition, parallel machines are notoriously difficult to program. One must "think in parallel" to write programs that not only compute the results correctly, but also schedule all the processors properly to avoid racing, even deadlock conditions. As a result, parallel programming often involves many intricate and error-prone details. Therefore, users should reuse existing working codes whenever possible. In the context of N-body computation, we should abstract out the common ingredients of tree codes so that they can be reused in different N-body problems.

The goal of this project is to develop a general N-body framework that eases the difficult task of writing efficient parallel N-body codes. The framework was developed based on our previous CM-5 implementations [3, 4, 10], in which we developed sound techniques that have been carefully studied both experimentally [9] and mathematically [8]. We expect that these proven techniques will guide us towards the ultimate goal of writing efficient parallel N-body programs with ease.

The remainder of this paper is organized as follows. Section 2 explains 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 our C++ N-body framework, and Section 6 concludes.

2 The Barnes-Hut algorithm

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 mass and the location of the centers-of-mass of the internal nodes are computed in one phase up the tree, starting at the leaves.

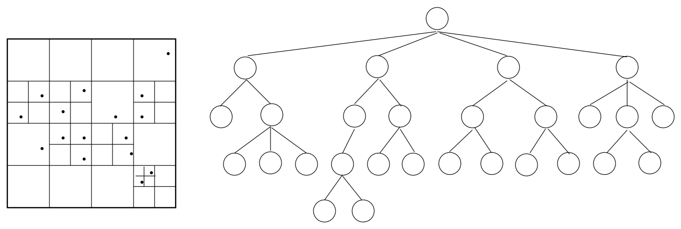
Figure 1: BH tree decomposition

To compute accelerations, we loop over the set of particles observing the following rules. 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)$/\theta$, 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, is now repeated for the desired number of time steps.

[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

3 Parallel Implementation

In the following subsections, we point out the differences between our parallel implementations [3, 4, 9, 10] 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. The ORB decomposition can be represented by a binary tree, which is stored in every processor. The ORB tree is used as a map which 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 [13], the cost of incremental load-balancing is negligible from our experience [9].

3.2 Building the BH-tree in parallel

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 which are split by ORB bisectors. A similar process was developed independently in [13].

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.

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 u to apply center-of-mass approximation, or far away enough to use u's parent's information. With the ORB map it is straightforward to locate the destination processors to which α might be essential. 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.

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. In an earlier paper [8] 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 2 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.

4 N-body Framework

We divide the C++ N-body framework into three layers: *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

```
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 2: Outline of code structure

simple tree construction and manipulation methods. System programmers can build special 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 these usage by writing a gravitational N-body code (Sec 4.3). Figure 3 illustrates the class hierarchy in these three layers.

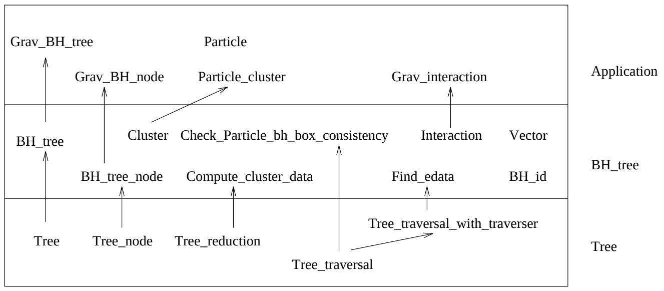

Figure 3: The class hierarchy in *generic tree*, *Barnes-hut tree*, and *application* layers.

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 a 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, 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.

```
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 4: Generic tree and reduction/traversal classes.

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 4). We implement these two tree operations as separate classes instead of methods in the `Tree` class mainly because they require their own data (not shown in Figure 4 for clarity) to function. For example, both operations maintain their current locations in the tree for easy access to tree nodes. In addition, we implement the class `Tree_traversal_with_traverser`, a traversing class in which we have to specify who is traversing the tree, as a subclass of `Tree_traversal`. For example, a particle will use `Tree_traversal_with_traverser` to collect its essentail data because we need the position of the particle to determine the distance, and we need only `Tree_traversal` to reset the tree node data.

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

```
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 5: 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.

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

Cluster data computation is implemented as a tree reduction (Figure 5). `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 how 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 6). 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 on an internal cluster.

```
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 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 6: Class for finding essential data and interaction computation.

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 6) 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 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 be 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, and relatively cheap to patch up.

4.3 Application Layer

The gravitational N-body application is built upon 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 7). Next, in the `Particle_cluster` class we define the methods for computing/combining center of mass and the methods for testing essential data.

```
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 7: Classes for a gravitational N-body application.

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`. 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`.

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

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.

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.

In our previous parallel C implementation, we constructed a 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. 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.

```
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;
};
```

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.

```
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;
};
```

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`.

5 Experimental Results

We demonstrate the flexibility of our library by writing a gravitational N-body code on a network of workstations. It took us only a few days to write all the necessary data structures and control logics for the gravitational simulation, since we inherited most of the tree and cluster structures from the BH tree layer. All we had to write are those segments specific to the gravitational simulation, including body-to-body and body-to-cluster interactions, the rules to combine center-of-mass, and data structures for particles and particle clusters.

N	sequential time	parallel time	speed up
8000	14.39	5.84	2.46
16000	20.40	6.28	3.25
24000	33.05	9.80	3.37
32000	45.91	13.25	3.46
40000	60.07	17.5	3.43
48000	73.82	21.03	3.50
56000	90.23	26.17	3.44
64000	103.59	29.17	3.55

Table 1: Timing comparison between the parallel C++ code using the framework and a sequential C implementation.

We conducted the experiments on four Ultra Sparc workstations connected by a fast ethernet network, located at Academia Sinica, Taiwan. The communication library functions were implemented in MPI version 1.0.4. To get a fair speedup number we compare our parallel execution time with the timing from a highly optimized sequential C code written by Barnes and Hut. Both the sequential and the parallel code use exactly the same Barnes-Hut algorithm. The input configuration is a set of uniformly distributed particles in three dimension. Table 1 summarizes the timing results from both codes.

Our parallel code developed from the tree framework has overhead from both the communication and extra function calls inevitable in object-oriented style of programming. However, the timing data shows a reasonably good speedup, even if compared with a highly optimized sequential C code. As the problem size increases, the speedup reaches a steady 3.5 with four processors. We plan to conduct more experiments on larger number of processors to evaluate the effects of communication on the simulation efficiency.

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, then compute the interactions one element at a time from the list. We chose this method mainly to separate the data collection process from the computation. However, we pay the overhead of allocating linked list element 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 approach would be to compute the interaction on-the-fly while traversing the tree. Both methods will be implemented and included into the final version of the library.

6 Conclusion

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

Our experience 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 code.

We have implemented a gravitational N-body code using the class libraries provided in this framework. As expected, using the framework greatly shortened the development time of this code. The performance of this code is competitive to its C implementation as well. To further evaluate our framework, we plan to implement a number of application programs, including a molecular dynamics code, a vortex simulation code, and the 3-d fast multipole method, using the class libraries we have developed.

Acknowledgments

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

References

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

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

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

[4] 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.

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

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

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

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

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

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

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

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

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

[14] 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.

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

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

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