Abstractions for Parallel N-body Simulations

(Extended Abstract)

Sandeep Bhatt Marina Chen Cheng-Yee Lin Pangfeng Liu

Department of Computer Science
Yale University
New Haven, CT 06520

Abstract

This paper introduces C++ programming abstractions for maintaining load-balanced partitions of irregular and adaptive trees. Such abstractions are useful across a range of applications and MIMD architectures. They free the user from low-level implementation details including interprocessor communication, data partitioning, and load balancing. We illustrate the use of these abstractions for gravitational N-body simulation.

Our strategy for parallel N-body simulation is based on a technique for implicitly representing a global tree across multiple processors. This substantially reduces the programming complexity and the overhead for distributed memory architectures. We further reduce the overhead by maintaining incremental data structures.

1 Introduction

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

1.1 A brief overview

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

Beginning with Appel [3] and Barnes and Hut [4], there has been a flurry of interest in faster algorithms for large-scale particle simulations. Experimental evidence shows that these heuristic algorithms require far fewer operations than the naive one for most initial distributions of interest, and that the resulting error is not unreasonable. Indeed, while there are pathological bad inputs for both algorithms, the number of operations per time step is $O(N)$ for Appel's method, and $O(N \log N)$ for the Barnes-Hut algorithm provided the particles are uniformly distributed in physical space. Greengard [12] presented an $O(N)$ algorithm which is provably correct to any fixed accuracy. However, because of the complexity and overheads in the fully adaptive version of Greengard's algorithm, the algorithm of Barnes and Hut continues to enjoy application in astrophysical simulations.

All the algorithms mentioned above are based on a divide-and-conquer strategy. The basic idea, as we shall see in Section 2, is to group particles within an oct-tree which is used to calculate interactions. As the particles move, the tree changes. Because the above algorithms share the tree structure in common, they are all commonly referred to as "tree codes." Not surprisingly, all the tree codes exhibit large amounts of parallelism.

Parallel implementations of the tree codes have been developed over the last few years. Zhao and Johnsson [22] describe a non-adaptive version of Greengard's algorithm on the Connection Machine CM-2. More recently, Salmon implemented the Barnes-Hut algorithm on message passing architectures including the NCUBE and iPSC [18]. Salmon in-

corporates multipole approximations into the Barnes-Hut algorithm, and demonstrates impressive speedup on MIMD architectures. For a reference to the extensive literature on N-body simulations, we refer the interested reader to Salmon's thesis which initially motivated this work.

In this paper we describe a simpler implementation on MIMD architectures which reduces the overhead of building and maintaining trees partitioned among multiple processors.

1.2 Object-Oriented Abstractions

Our primary objective in this paper is to show how our implementation can be coded in a manner that is independent of architectural and communication mechanisms. Our goal is that the user code manipulate only objects in physical space; all details concerning multiple processors, data partitions, communication, and load-balance must be hidden from the user. An additional advantage of the abstractions, which we will report in further work, is that large portions of our code can be shared by the different tree codes, and also by other particle-in-cell methods.

Object-oriented abstractions are useful in scientific code development [2, 10, 17]. The natural match between object-oriented languages and parallel machines has prompted several projects on parallel implementations of these languages, for example PARAGON[9], and PC++ [15].

PARAGON uses an embedded class PARRAY whose implementation takes care of message-passing and data distribution using a standard partition strategy such as blocking. Paragon contains PARRAY implementations as library routines for machines such as the Intel iPSC and FPS T20. PC++ contains a set of distributed data structures (arrays, priority queues, lists, etc.) implemented as library routines, where data are automatically distributed based on directives (whole, block, cyclic, random). PC++ attempts to support data abstractions in addition to parallelism. The library routines are written for the Alliant FX/8 and FX/2800, and the BBN GP1000.

In Section 3 we introduce the classes PTREE and PLIST to support parallel tree structures. Unlike the parallel data structures of Paragon and PC++ which are restricted to static partitions, PTREE and PLIST structures are partitioned dynamically among processors according to user-specified dynamic data distribution strategies, for example the balanced orthogonal recursive bisection (ORB) mentioned in Section 2).

While our classes PTREE and PLIST are similar in spirit to the classes supported in Paragon and PC++, there is an important distinction. We introduce the *Traverse-Deliver* programming model, necessary because the simple approach of interpreting index or pointer references at run-time as interprocessor communication [19] would be grossly inefficient for applications of the degree of complexity as N-body simulations. This comment applies to other work on object-oriented parallel programming such as Object-Oriented Interface (OOI)[20], Mentat Run-time System [13], and Concurrent Aggregates (CA)[7].

Our work is also related to various run-time systems for parallel programming such as PARTI[11] and Kali[14]. Again, neither simple run-time pointer interpretation, nor optimization such as hashed caches of PARTI, is sufficient for tackling the class of applications at hand. A high-level programming model such as our Traverse-Deliver model seems crucial for high performance.

2 The Barnes-Hut algorithm

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

To organize a hierarchy of clusters, 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 is shown in Figure 1; the corresponding BH-tree is shown in Figure 2.

Each internal node of the BH-tree represents a cluster. Once the BH-tree has been built, the centers-of-mass of the internal nodes are filled in. This is done in one phase up the tree, starting at the leaves. Next, to compute accelerations, we loop over the set of particles observing the following rules: (1) each particle starts at the root of the BH-tree, and (2) for any particle and internal node, if the particle lies outside the box and the distance between the particle and the box is less than RADIUS(box)/θ then the particle visits each of the children of the box[2]; otherwise, the acceleration

[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.

[2] The distance measured can be either the distance from the

due to the particles within the box is approximated by a single two-body interaction between the particle and a point mass located at the center-of-mass of the box.

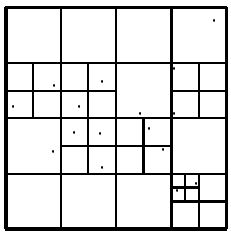

Figure 1: BH tree decomposition

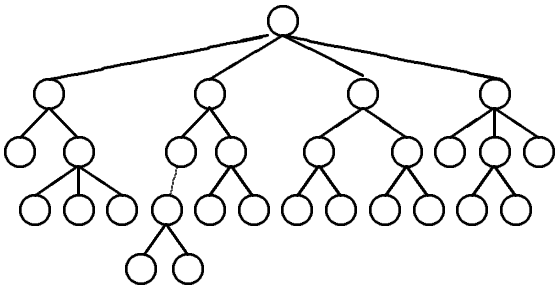

Figure 2: BH tree

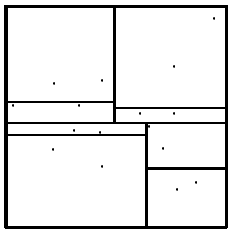

Figure 3: ORB

Each particle traverses the tree in a top-down manner; nodes visited in the traversal form a sub-tree of the entire BH-tree. Of course, different particles will, in general, traverse different subtrees.

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.

2.1 Overheads in parallel implementation

With a single processor, the overhead in building the tree, and traversing it while computing centers-of-mass and accelerations is negligible. With only ten thousand particles, more than 90% of the time is devoted to arithmetic operations involved in computing accelerations. Thus, it is reasonable to build the BH-tree from scratch at each iteration.

On a massively parallel machine, the entire BH-tree cannot be stored in any one processor. With the particles partitioned among the processors of the machine, the costs of building and traversing the BH-tree can increase significantly. In contrast, the time for arithmetic can, in principle, decrease linearly as the number of processors increases. This tension between the communication overhead and computational throughput is of central concern to both programmers and architects.

A simple way to initially distribute the set of particles among P processors is by recursively partitioning the enclosing region into P connected subregions such that each subregion contains $\lceil N/P \rceil$ particles. Figure 3 illustrates an example of an orthogonal recursive bisection, or ORB. The ORB can itself be represented by a binary tree, with leaves corresponding to processors, and internal node representing subregion that are further subdivided. ORB is common in geometric decompositions, and is used by Salmon as well.

Instead of building a global copy of the BH-tree, Salmon [18] instead creates at each processor a copy of the subtree (of the BH-tree) that is essential to the acceleration calculations at that processor (the "locally essential tree," in Salmon's terminology). The motivation for building a copy of the locally essential tree at every processor is that all data required to update positions and velocities is available at each processor locally before the final, computationally intensive, phase begins.

Building the locally essential trees constitutes the bulk of the overhead in Salmon's implementation. This phase is complex to program, and the flow of data is such that, in general, a processor will have to read and forward data that is not essential to it. Singh, Hennessy and Gupta [19] refer to this problem as well, and conclude that, unlike a shared-memory implementation, message passing implementations must suffer considerable overhead and require enormous programming complexity.

We have programmed a different message-passing implementation of the Barnes-Hut algorithm which is substantially simpler and requires less overhead. Our experience has been that the programming effort can be reasonable provided the right abstractions are first developed. The implementation, motivated by Salmon's work, loops over the phases outlined below. Initially, particles are distributed among proces-

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

sors according to the ORB scheme.

1. Update BH-tree for local particles.
2. Build an (implicit) representation of one shared copy the of global BH-tree.
3. Transmit each BH-tree node to all processors for which it is locally essential.
4. Compute the new positions and velocities.
5. Move particles to new positions, balance load, and update the ORB-tree.

Our experiments show that the time for Phases 1 and 2 is minuscule (less than 2%) even with over one million particles and several hundred processors. By storing a copy of the ORB-tree in each processor, it is an easy local calculation in Phase 3 to figure out which processors a BH-tree node is locally essential to, so that no "junk mail" is ever received by a processor. As expected, the volume of data transmitted makes this the second-most time-consuming phase. But, this volume of data transmission is inherent to the problem, not the implementation. Phase 4 requires no communication, and takes more than 75% of the total time. The overhead in Phase 5 is found to be small as well. By keeping all data structures incremental, we avoid the overhead of computing them from scratch at each iteration. These overhead figures for our initial codes will further reduce substantially as we begin to investigate alternative data structures and code optimization tricks. Further details on timings can be found in [5], and will be reported at the conference.

3 Programming abstractions

In the following, we show that proper abstractions not only reduce the complexity of the programming task but also allow the separation of implementation concerns from the algorithmic content of the application. This allows reuse of the algorithmic components (*e.g.* decomposition, communication, traversal on data structures) in many different contexts.

By building C++ library classes that capture useful data structures (*e.g.* lists, trees) as well as decomposition schemes and communication routines, application codes (*e.g.* N-body simulation) can be written entirely independent of the implementation details.

3.1 Phantom global structures

To ease the programming task, we don't want an application programmer to think about processors, sending messages, or how to distribute the data. In sequential programming, a user builds a data structure, traverses it, applies proper actions to the elements, and perhaps modifies the structure itself. Our goal is to allow that same level of convenience for programming massively parallel machines where data structures are distributed and where each processor has only a local view into the world. It is well understood how to provide the illusion of a global data structure for programs with only dense arrays and using few or no indirect references [1, 6, 8, 16, 9].

The difficulty with supplying the illusion of a global list, tree, or graph is twofold: (1) regular, static decomposition, or even randomization does not work well enough, and (2) references to the elements of these structures cannot be resolved at compile-time, and naive run-time resolution schemes are generally too inefficient to be useful.

Our solution is to provide a high-level abstraction, the Traverse-Deliver model, to be elaborated later, for accessing and modifying a phantom global data structures while, in reality, implementing it as a set of local data structures with communication among them. It turns out that elevating the user's interaction with the data structure from the pointer level to the Traverse-Deliver model is crucial for good performance.

In the following, we will describe how to construct a user-program using library classes and functions that support phantom data structures and the Traverse-Deliver model.

3.2 Customizing phantom structures

Three mechanisms are used to customize these classes and functions: (1) type parameters of `template` for either a class or a function, (2) function parameters where the definition of the function is supplied by the user, and (3) virtual methods of a class, where the body of a method is supplied by the user by defining a subclass.

The phantom global data structures perceived by the user are created by defining derivative classes, or subclasses, of class `PLIST` and class `PTREE` provided in our library. Both classes are templates which are customized by the structure of the node of the tree or list defined by the user.

For example, in the N-body simulation program, the user defines a tree structure called `BH_TREE` by declaring:

```
class BH_TREE:
      public BH_NODE,
      public PTREE<BH_TREE, BALANCED_ORB, BOX> {
 public:
    /* public functions for force computation
       and updates of particle positions. */ } ;
```

where BH_NODE is the node structure defined as

```
typedef struct
  BHNODE_ID id ;   short BHNODE_type ;
  int particle_number_in_subtree ;
  CENTER_OF_MASS CoM;   PARTICLE_LIST* particle ;
} BH_NODE ;
```

3.3 Customizing decomposition strategy

In addition to the user-defined node structure, another user-supplied parameter to the PTREE or PLIST class template is the decomposition strategy, in this case BALANCED_ORB. Since the strategy of ORB may be used in other application programs, it is supported as a library class template which is then customized for each application.

The decomposition strategy BALANCED_ORB contains a method (do_remapping) for dynamic load-balancing as well as a method (init) for partitioning the input data structure. The mapping from the user data-structure to processors is handled by the method procs_in_range.

The skeleton of the class BALANCED_ORB is given below:

```
template<class INPUT_NODE_TYPE, class STRUCTURE_TYPE,
         class RANGE_TYPE, class ACCESS_TYPE>
 class BALANCED_ORB {
  BISECTION_TREE_NODE* bisection_tree_table ;
  int current_load, num_of_procs, space_dimensionality ;
  BOUNDARY boundary_in_space ;
  ACCESS_TYPE (*access_function)(INPUT_NODE_TYPE) ;
  int (*regional_load)(STRUCTURE_TYPE*) ;
 public:
  void init(int, INPUT_NODE_TYPE*, ACCESS_TYPE
        (*node_access_function)(INPUT_NODE_TYPE)) ;
  PROCESSOR_SET procs_in_range(RANGE_TYPE) ;

  void do_remapping(STRUCTURE_TYPE*,
          int (*regional_load_estimate)
                           (STRUCTURE_TYPE*))
    { /* incremental remapping scheme */ }

  BOOLEAN load_balancing_criterion() {
  /* do reduction over the current_load of all
     partitions.  return true if maximal load
     > average_load*load_balancing_ratio. */ }  } ;
```

To customize BALANCED_ORB, the type parameters to the template are supplied. For our example, the following declarations appear in the user program:

```
BALANCED_ORB<INPUT_PARTICLE_LIST,
             PARTICLE_LIST, BOX, COORDINATE*>
```

where INPUT_PARTICLE_LIST is a class for the user's data structure to be decomposed, PARTICLE_LIST is a subclass of PLIST, BOX is a user struct type used to specify a range of nodes in the phantom data structure, and COORDINATE specifies the fields of the user's data structure that the ORB class needs to access.

3.4 The Traverse-Deliver model

The idea behind the Traverse-Deliver model is that computation on the phantom data structure alternates between (1) traversing it while applying actions on each node of the structure, and (2) extracting information from each node and exchanging information between nodes. This Traverse-Deliver model can be classified as a bulk-synchronous model [21] suggested by Valiant.

In the following, we present the Traverse-Deliver model for linear lists and trees. It generalizes to other structures easily.

Traverse Traversal on a tree structure, using depth-first order in this case, is defined to be a C++ function as shown below that returns nothing (void) and takes three function parameters pre_action, in_action, and post_action. The user supplies these functions as well as the structure at each node of the tree (NODE_TYPE). The function pre_action is applied when a node is first visited in the depth-first traversal order; in_action is applied when each of its child nodes is visited; post_action is applied when all children have been visited. Similar traversal functions using other orderings can be provided.

```
void depth_first_traverse
     (BOOLEAN (*pre_action)(NODE_TYPE*),
      BOOLEAN (*in_action)(NODE_TYPE*, NODE_TYPE*),
      void (*post_action)(NODE_TYPE*))
```

Deliver Deliever on a tree structure is defined to be a C++ *function template* shown below which allows the types of its parameters to be variant and be bound to the types of the actual parameters. The user provides the data structure over which the delivery is to be done, the type of "mail" to be delivered, and a delivery rule. The type of mail contains information about how to extract data for delivery and how to interpret data received from other nodes. The delivery rule specifies such information as the destination nodes and actions to be taken at each node during delivery.

```
template<class TREE_TYPE, class MAIL_TYPE,
         class DELIVERY_RULE, class RANGE_TYPE>
```

```
  void tree_deliver(TREE_TYPE*,
                    MAIL_TYPE&, DELIVERY_RULE&)
```

3.5 Customized mailtypes

The types of mail and delivery rules need to be supplied for Deliver. These types are in turn supported by library class templates and the user customizes them by supplying appropriate type parameters to the templates. We first discuss the class template `MAILBOX` defined as follows:

```
template<class OUTGOING_TYPE, class STRUCTURE_TYPE>
 class MAILBOX {
  /* buffers needed for communication. */
 public:
  virtual OUTGOING_TYPE
           make_package(STRUCTURE_TYPE*) {}
  virtual void
   process_incoming_package(OUTGOING_TYPE*) {}
  virtual void process_upon_acknowledge() {}

  /* other functions for buffer access. */ } ;
```

One example of using a mailbox is in the force computation of the N-body simulation where the center of mass associated with each node of a BH-tree is delivered to those nodes that need it for the force computation. We declare `Mail_USE_CoM` to be a subclass of `MAILBOX` as follows:

```
typedef class MAILBOX<MSG_TRUE_CoM, BH_TREE> MAIL_USE_CoM ;
```

where `MAIL_USE_CoM` is the structure defined to be

```
typedef struct {  BHNODE_ID id ; int BHNODE_type ;
         CENTER_OF_MASS CoM ; }     MSG_TRUE_CoM ;
```

Finally, the body of each of the three class methods need to be supplied by the user. For example, the `make_package` method is defined as follows:

```
MSG_TRUE_CoM
  MAIL_USE_CoM::make_package (BH_TREE* bh_node) {
  /* extract information from a bh_node
     and make a package of type MSG_TRUE_CoM */ }
void MAIL_USE_CoM::process_incoming_package
                          (MSG_TRUE_CoM* CoM)
   { /* place true CoM into BH_tree. */ }
void MAIL_USE_CoM::process_upon_acknowledge(){}
```

3.6 Customized Delivery Rules

We now turn to the delivery rules, defined as a class template as follows:

```
template<class NODE_TYPE, class RANGE_TYPE>
class DELIVERY_RULE {
 public:
  virtual BOOLEAN is_dead_end(NODE_TYPE*)
                       { return(FALSE) ; }
  virtual RANGE_TYPE deliver_area(NODE_TYPE*) {}
  virtual void action_on_site(NODE_TYPE*) {}  } ;
```

Delivery stops traversal further along a delivery path whenever it encounters a dead end. Otherwise, it determines its delivery area and sends out the mail. If delivery area covers includes the current site, it performs the action on site.

An example of use in the force calculation is to declare a subclass of class `DELIVERY_RULE`.

```
class USE_CoM_DELIVERY_RULE:
    public DELIVERY_RULE<BH_TREE, BOX> {} ;
```

And in application, supply the body of the class methods declared as virtual in its superclass.

```
BOOLEAN USE_CoM_DELIVERY_RULE::is_dead_end
                            (BH_TREE* bh_node)
   { return(check_true_CoM(bh_node)) ; }
BOX USE_CoM_DELIVERY_RULE::deliver_area
                            (BH_TREE* bh_node)
   { return (influence_area(bh_node)) ; }
void USE_CoM_DELIVERY_RULE::action_on_site
                            (BH_TREE* bh_node)
   { /* mark bh_node as used_on_site */ }
```

Summary The library provides classes `PTREE` and `PLIST` in which methods for the Traverse-Deliver Model are supported. Deliver in turn needs specification of mailtypes and delivery rules, and these are supported by classes `MAILBOX` and `DELIVERY_RULE`. In addition, decomposition using ORB is supported by the `BALANCED_ORB` class.

4 Reality and implementation

In reality, the user program, together with the illusion of a global data structure and library classes and functions, is itself an SPMD program that runs on multiple processors and multiple memories. Each processor as well as the host runs the same program.

All computation is done by calling Traverse and all communication is done by Deliver, namely Traverse with specific delivery task. Looking at the implementation of Deliver will help to see how the phantom structure appears from a collection of SPMD programs. We will use class `PTREE` as the example. For ease of exposition, we use a naive pointer-based implementation of the tree structure. A more efficient implementation is used in the library.

```
template<class NODE_TYPE,
         class DECOMPOSITION_TYPE, class RANGE_TYPE>
```

```
class PTREE {
 DECOMPOSITION_TYPE* decomposition ;

public:
 NODE_TYPE* parent ;
 NODE_TYPE* child[Nchildren] ;
 NODE_TYPE* init (NODE_TYPE*,
              DECOMPOSITION_TYPE&) ;
 NODE_TYPE* read (NODE_TYPE*) ;
 NODE_TYPE* write (NODE_TYPE*) ;
 void depth_first_traverse
  (BOOLEAN (*pre_action)(NODE_TYPE*),
   BOOLEAN (*in_action)(NODE_TYPE*,
                        NODE_TYPE*),
      void (*post_action)(NODE_TYPE*))
 { /* traverse in depth first order */ } } ;

template<class TREE_TYPE, class MAIL_TYPE,
         class DELIVERY_RULE, class RANGE_TYPE> {
 void tree_deliver
  (TREE_TYPE* tree_ptr, MAIL_TYPE& mail,
   DELIVERY_RULE& rule) {
  mail.init() ;
  tree_deliver_rec(tree_ptr, mail, rule) ;
  mail.clear_all_bufs() ;
  mail.sync(0, maxlevel) ;
  mail.free_bufs() ;  }

template<class TREE_TYPE, class MAIL_TYPE,
         class DELIVERY_RULE, class RANGE_TYPE> {
 void tree_deliver_rec (TREE_TYPE* tree_ptr,
    MAIL_TYPE& mail, DELIVERY_RULE& rule) {
  PROCESSOR_SET processor_set ;
  RANGE_TYPE  range ;   int p, c ;

  if (tree_ptr == NULL) return ;
  if (rule.is_dead_end (tree_ptr)) return ;
  range = rule.deliver_area (tree_ptr) ;
  processor_set = decomp.procs_in_range(range) ;
  if (is_non_empty_processor_set (processor_set)) {
    for (p = 0; p < P; p++)
      if (is_a_member (p, processor_set))
        if (p == my_node)
          rule.action_on_site (tree_ptr) ;
        else mail.pack_into_buf (tree_ptr, p) ;  }
  for (c = 0; c < Nchildren ; c++)
    tree_deliver_rec(tree_ptr->child[c],
                     mail, rule) ;   }
```

Note that **write** and **read** allow data to be delivered to and from the host to the processor in a massively parallel machine.

Implementation of **MAILBOX** class shows how communication between processors are actually done. Here we list the function header of its methods and sketch one particular method **pack_into_buf**.

```
template<class OUTGOING_TYPE, class STRUCTURE_TYPE>
class MAILBOX {
  OUTGOING_TYPE* *outbuf ;
  OUTGOING_TYPE  item ;
  int       *count ;
  int       numbuf, bufsize,  ack_received, buffer_sent ;

 public:
  virtual OUTGOING_TYPE make_package(STRUCTURE_TYPE*) {}
  virtual void process_incoming_package(OUTGOING_TYPE*) {}
  virtual void process_upon_acknowledge() {}
  void init() ;
  void free_bufs() ;
  void remove_from_buf() ;
  void clear_all_bufs() ;
  void sync(long, int) ;

  void pack_into_buf(STRUCTURE_TYPE* ptr, long p) {
    OUTGOING_TYPE* item ;
    item = make_package(ptr) ;
    /* place item into buffer, send out filled buffers,
       and remove data from incoming buffers */  }
```

5 The application program

The skeleton of the top-level user program appears below.

```
INPUT_PARTICLE_LIST* input_particle_list ;
PARTICLE_LIST* particle_list ;
BH_TREE*  bh_tree ;
BALANCED_ORB<INPUT_PARTICLE_LIST, PARTICLE_LIST,
             BOX, COORDINATE*> orb_map ;

/* Main program */
main() {
  int iteration ;
  read_particle_list(input_particle_list) ;
  orb_map.init(Dimension, input_particle_list,
               get_position) ;
  particle_list->init(input_particle_list,
                      orb_map) ;
  bh_tree->init(particle_list) ;
  for (iteration = 0 ;
       iteration < Max_iteration ;
       iteration++)
  { bh_tree->update() ;
    bh_tree->compute_CoM() ;
    bh_tree->compute_forces() ;
    bh_tree->new_position() ;   } }
```

The skeleton of the definition of class **BH_TREE** is given below.

```
class BH_TREE ;
class BH_TREE:
      public BH_NODE,
      public PTREE<BH_TREE, BALANCED_ORB, BOX> {
 public:
  void init(PARTICLE_LIST*) ;
  void update() ;
  void compute_CoM() ;
  void compute_forces() ;
  void new_position() ;   } ;
```

The last four functions are used within the inner loop for each iteration. We show below the code fragment for the user-defined function **compute_forces**. The example is meant to illustrate the linguistic power afforded by the abstractions provided by the library.

```
void BH_TREE::compute_forces() {
  /* deliver CoM for use wherever essential*/
  tree_deliver(bh_tree, mail_use_CoM,
               use_CoM_delivery_rule) ;
  /* compute velocity */
  particle_list->traverse
       (no_pre_action, update_velocity,
        no_post_action) ;    }
```

The interested reader is referred to [5] for further details.

Acknowledgments We thank Lennart Johnsson, Abhiram Ranade, John Salmon and Geoffrey Fox for helpful discussions. This research was supported in part by a grant from DARPA, monitored by Army DOC under contract DABT 63-91-C-0031, Air Force grant AFOSR-89-0382, NSF grant CCR-88-07426, and NSF/DARPA grant CCR-89-08285.

References

[1] *CM-Fortran Programmer's Manual*, 1990.

[2] I. G. Angus and W. T. Thompkins. Data storage concurrency, and portability: An object oriented approach to fluid mechanics. Technical report, Northrop Research and Technology Center, CA.

[3] A.W. Appel. An efficient program for many-body simulation. *SIAM J. Sci. Stat. Comput.*, 6, 1985.

[4] J. Barnes and P. Hut. A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature*, 324:446–449, 1986.

[5] S. Bhatt, M. Chen, C-Y Lin, and P. Liu. Abstractions for parallel N-body simulations. Technical Report DCS/TR-895, Yale University, 1992.

[6] M. Chen and J. Cowie. Prototyping Fortran-90 Compilers for Massively Parallel Machines. In *ACM SIGPLAN'92 PLDI Conference*, 1992.

[7] A. Chien and W. Dally. Experience with concurrent aggregates (CA): Implementation and programming. In *5th DMCC*, 1990.

[8] B. Chapman et. al. Vienna FORTRAN – A Fortran Language Extension for Distributed Memory Multiprocessors. In *High Performance FORTRAN Forum*, 1992.

[9] C. Chase et. al. Paragon: A parallel programming environment for scientific applications using communication structures. In *IEEE ICPP*, pages 211–218, 1991.

[10] D. Forslund et. al. Experiences in writing a distributed particle simulation code in C++. In *USENIX C++ Conference*, pages 1–19, 1990.

[11] S. Hiranandani et. al. Performance of hashed cache data migration schemes on multicomputers. *JPDC*, 12:415–422, 1991.

[12] L. Greengard. *The rapid evaluation of potential fields in particle systems.* MIT Press, 1988.

[13] A. Grimshaw. The mentat run-time system: Support for medium grain parallel computation. In *5th DMCC*, pages 1064–1073, 1990.

[14] C. Koelbel, P. Mehrotra, and J. V. Rosendale. Supporting Shared Data Structures On Distributed Memory Architectures. Technical report, ICASE, NASA Langley Research Center, 1990.

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

[16] J. Li and M. Chen. Compiling Communication-Efficient Programs for Masssively Parallel Machines. *IEEE Trans. Par. and Dist. Sys.*, (3), July 1991.

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

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

[19] J. Singh, J. Hennessy, and A. Gupta. Implications of hierarchical N-body methods for multiprocessor architectures. *manuscript*, 1992.

[20] T. Ungerer and L. Bic. An object-oriented interface for parallel programming of loosely-coupled multiprocessor systems. In *Proceedings of 2nd EDMCC, Springer-Verlag Lect. Notes in Comp. Sci.*, volume 487, pages 163–172, 1991.

[21] L.G. Valiant. A bridging model for parallel computation. *CACM*, 33, 1990.

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