Distributed Data Structure Design for Scientific Computation

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

Pangfeng Liu
Department of Computer Science
National Chung Cheng University
Chia-Yi, Taiwan 62107, R.O.C.
pangfeng@cs.ccu.edu.tw

Abstract *This paper gives an overview of the VGDS (Virtual Global Data Structure) project. The VGDS effort focuses on developing an integrated, distributed environment that allows fast prototyping of a diverse set of simulation problems in scientific and engineering domains, including regular, irregular, and adaptive problems. The framework defines three base libraries, Array, Graph, and Tree, that capture major data structures involved in scientific computation. The framework defines multiple layers of class libraries which work together to provide data-parallel representations to application developers while encapsulate parallel implementation details into lower layers of the framework. The layered approach enables easy extension of the base libraries to a variety of application-specific data structures. Experimental results on a Sun UltraSparc workstation cluster is reported.*

Keywords: distributed data structures, parallel scientific computation, object oriented framework

1 Introduction

With the advance of commercial off-the-shelf microprocessors and interconnection technology, distributed-memory parallel machines (e.g. MPPs, workstation clusters, and SMP clusters) have become an attractive computing platform for solving large problems. In the past few years, a large number of scientific computing applications have benefit from such parallel computers. Despite this success, parallel programming for distributed-memory machines remains a difficult task, mainly because it often involves many intricate details that are error-prone and difficult to debug. This has become the major bottleneck of HPC software development. This is particularly evident in development of irregular and adaptive applications. The goal of this effort has been to identify and nurture one enabling technology for high performance scientific computation: object-oriented construction of virtual global data structures.

Fortran 90 and High Performance Fortran (HPF) are counted among HPC's software successes, because they demonstrate the utility of parallel arrays for hiding low-level distributed memory details. This is possible because array structures are used in highly idiomatic ways in scientific applications, and compiler and library writers exploit those idioms to extract high performance—an HPF array being just one instance of a virtual global data structure.

The uses of irregular and adaptive data structures in commercial and scientific applications are even more stylized, precisely because of their irregularity and similarity. For example, most of the tree-based scientific codes use similar tree structures and exhibit similar computation patterns. 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. Furthermore, different simulation algorithms may use the same data structure. For instance, 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 tree framework helps in developing tree codes for different application domains, and in implementing different tree algorithms as well. Another example is unstructured mesh computation. Although unstructured mesh computations may perform different calculations according to the systems being simulated, they use the mesh virtually the same way – A mesh point updates its stored data, which represent some physical quantities at that point, by retrieving the data from its neighbors and performing calculations on them. Such property gives distributed data structure designers the ability to map them to HPC architectures and extract their parallelism.

In this paper, we present a data structuring framework, *Virtual Global Data Structures*, for parallel and distributed scientific computation. The VGDS framework defines three base libraries, Array, Graph, and Tree, that capture major data structures involved in scientific and engineering computation. The framework implemented distributed data structures as object-oriented classes with explicit associated method interfaces. Application-specific data structures can be derived from these base libraries through class inheritance.

In the past few years, many research efforts have been devoted to designing efficient numerical libraries for matrix-based parallel computation (e.g. MultiMATLAB [20], PETSc [16], and ScaLAPACK [7]). Research efforts in providing suitable object-oriented parallel languages/libraries for certain classes of applications have also been abundant [4, 8, 3, 13, 21]. Our VGDS effort distinguishes itself from others in two aspects. First, instead of tackling one particular data structure, we propose an integrated framework incorporating a diverse set of data structure classes that are essential in scientific and engineering computation. These include

regular (in which data reference patterns are uniform), irregular (in which data reference patterns are non-uniform), and adaptive (in which data reference patterns keep changing dynamically and incrementally) applications.

Secondly, we use layered object-oriented design and analysis in the construction of the VGDS base libraries. System objects in the upper layers of the framework are relevant to application specific domains (e.g. computational fluid dynamics simulation, molecular dynamics simulation, etc.) while objects lower in the framework capture the abstraction of parallelism and efficient processor-level computation. This layered approach provides a natural breakdown of responsibility in designing a complete HPC system, and allows design effort and heavy-duty optimization to be easily expended exactly where it is most needed.

This paper reports on the progress of our VGDS effort. Section 2 describes the organization and functionality of the VGDS framework. Section 3 discusses some implementation details of virtual global data structures in distributed-memory environments. Section 4 illustrates the core functionality of the VGDS framework using the Tree library as a running example. Section 5 reports our experimental results on a Sun UltraSparc-2 workstation cluster. Section 6 describes related work and Section 7 concludes.

2 The VGDS Framework

The VGDS framework defines three layers of C++ classes: the global layer, the parallel abstraction layer, and the local layer. Layers of application components can be built atop the VGDS basis. Figure 1 depicts the structure of this framework.

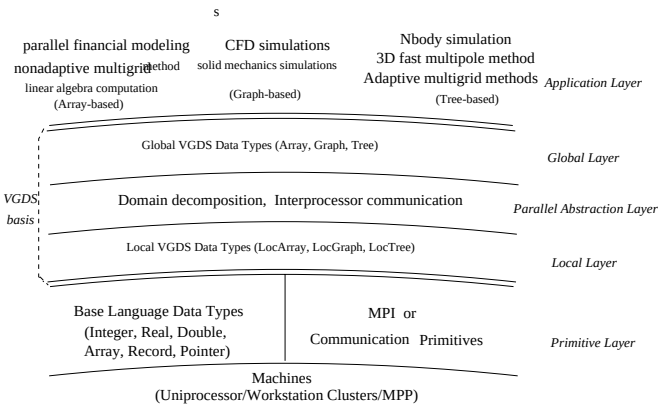

Figure 1: The VGDS Framework

The global and local layers together define various virtual global operations on regular (such as arrays), irregular (such as graphs, unstructured meshes), and adaptive (such as adaptive trees, and adaptive grids) data structures. The global layer defines global data types. Objects in the global layer are bookkeepers that delegate computational tasks to the local layer. The local layer implements generic, processor-local computational kernels for each VGDS component. The interactions between the global and the local layers are mediated by the parallel abstraction layer that captures the abstraction of parallelism, including data decomposition, interprocessor communication, and load balancing.

Currently, the VGDS framework provides three basic data structures: `Array` (for regular computation), `Graph` (for irregular graph-based computation), and `Tree` (for adaptive tree-based computation). Application-specific data structures can be derived from these basic data structures. For instance, a dense matrix class can be derived from the `Array` class by inheriting the `Array` class and defining additional methods essential in dense matrix computation. Table 1 outlines the classes and functionality of each layer, details of which are described in the following sections.

Table 1: VGDS Framework Functionalities

Layers	Classes		Functionality
	Base	Derived	
Global	*Array*	*Grid* *Matrix*	data-parallel operations
	Graph	*UMesh*	
	Tree	*BHTree* *AdaptGrid*	
Local	*LocArray*	*LocGrid* *LocMatrix*	processor-local operations
	LocGraph	*LocUMesh*	
	LocTree	*LocBHTree* *LocAdaptGrid*	
Parallel Abstraction	*Mapper*	BlockPartitioner ORBPartitioner	data layout management
	Communicator		inter-processor communication
	Message		message abstractions

2.1 Global and Local Layers

The VGDSs within the global layer provide a global view of the data, in which the data structure is treated as a monolithic whole, with operators that manipulate individual elements and implicitly iterate over substructures. In the local view (the local layer), each processor contains only a part of the whole, with operators acting only on the local data.

When a global data structure is instantiated, it creates a constituent local data substructure on each processor. Whenever a kernel operator associated with the data structure is invoked, the operation is carried out by first retrieving the handles to the local data, then delegating complete local computation to each local data substructure. If communication is required, it is performed through system objects in the parallel abstraction layer.

2.2 Parallel Abstraction Layer

The parallel abstraction layer defines classes for data layout, interprocessor communication, and load balancing for virtual global data structures. Classes in this layer are implemented as abstract classes and can be shared among various data structures. The key features of this layer are encapsulated into two groups of classes – data decomposition classes that are responsible for processor geometry, data partitioning and mapping, and load balancing, and communication classes that take care of data movement between processors.

Data Decomposition Classes

The global data structure are partitioned into local substructures on each processor according to the *Mapper* class. *Mapper* is an abstract class that define common service interface for finding the geometry of a VGDS, identifying global neighbor relations between their constituent local substructures, and deriving logical send- and receive-sets for a given global subscript resolved into the local substructure. Concrete mapping classes that are derived from *Mapper* provide domain specific information and functionality that can be tuned to the need of the specific data structure. For example, VGDS supports a *BlockPartitioner* for arrays and a *ORBPartitioner* (Orthogonal Recursive Bisection) for irregular and adaptive data structures. By instantiating the

Mapper class, the user can also construct customized data decomposition strategies.

Communication Classes

Two groups of classes are implemented to support portable, transparent message-passing communication on distributed-memory machines – *Message* and *Communicator*. The *Message* class is used to encapsulate data in a common format for easy data delivery and retrieval of different data structures. The *Communicator* is an abstract class that defines common service interfaces for buffer allocation, message delivery, and data handling related to communication. These services are encapsulated into three methods: `extract`, `communicate`, and `process`. Communicating data elements between processors are performed in three steps. First, the *Communicator* extracts data elements for sending by traversing the specified region in the VGDS data object and packing data elements into a *Message* object. Then the *Communicator* delivers(communicates) the *Message* object according to the given communication scheduling algorithm. When a *Message* object is received, the *Communicator* unpacks it and stores the data elements to the appropriate locations in the VGDS data object. The extraction and the restoring process requires interaction with the VGDS data object. The method `communicate` is implemented on top of MPI, to assure portability.

Figure 2 depicts the interactions between major classes in the VGDS framework. When a VGDS data object invokes a method that requires remote data accesses, the data object consults the *Mapper* object for the identifiers of the processors on which the global subscripts are mapped, and inserts them into a list of sends and receives (called communication schedule). The data object then requests the *Communicator* to carry out the planned data movement. During the course of computation, if the VGDS data object decides that a remapping is necessary (e.g. for load balancing purpose), it invokes the remap method in *Mapper*, which in turn redistributes the data structure incrementally.

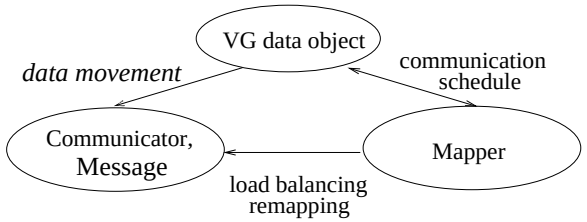

Figure 2: Interaction of Classes in the VGDS Framework

3 Data Coherence and Synchronization

Since a virtual global data structure is distributed over local memories of processors, in order to effect the same computation as in the global view, the local computations must be coordinated. We adopt the owner-computes rule, which distributes computations according to the mapping of data across processors. However, a local substructure may require information from other processors to complete the computation of data assigned to it. When communications mostly occur between neighboring processors and the same communication patterns may occur many times during program execution, it is more efficient to duplicate boundary

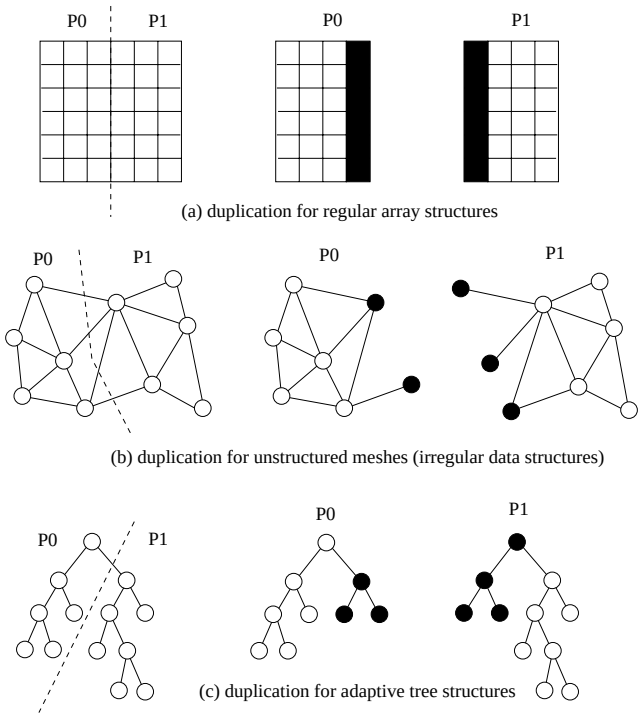

Figure 3: Duplication for distributed data structures. The duplicated data are indicated by solid black.

data elements on adjacent processors. For example, in an unstructured mesh computation where the new data value of a mesh node is a function of its neighbors, by duplicating boundary mesh nodes to the other side of partitioning lines, computations on the local submeshes on individual processors can all be performed locally without communication. In reality, data elements may be read or updated, which raises the issues of data coherence and synchronization. We describe our approach next.

We classify the data into two categories, *master* copy and *duplication*. A master copy is a data region in the original global structure that is mapped to a processor. A master copy can make copies of itself, called duplication, on other processors. That is, all the data elements that are essential to the computations of the local master copies will be fetched into the local substructure on the processor which owns the master copies. As far as each master copy is concerned, there is no distinction between global and local structures. Note that we do not have the notion of global pointers because all the pointers address a local memory address, be it a master copy or a duplication. The computations read and update the master copy only – the duplications only provide data and are read-only. Therefore, data coherence is guaranteed by allowing only the master copy to be updated, and only one master copy exists for one data element.

Figure 3 shows the duplication mechanism for a regular array, an unstructured mesh, and an adaptive Barnes-Hut tree for N-body algorithms. We assume that the computation of each element in the regular array and the unstructured mesh requires its neighbors, and the per-particle force computation of the Barnes-Hut algorithm requires a traversal on the adaptive Barnes-Hut tree.

To assure synchronization, data elements are duplicated before the actual computation is performed. After data are

partitioned, system objects in the parallel abstraction layer duplicate the data to the processors where they are essential to the computation. A barrier synchronization separates the duplication process from the computation, assuring that all the data are available and the computation can proceed without any further communication. This mechanism guarantees safety in a distributed environment.

4 Case Study: BH Tree

In this section, we use the `Tree` class (and a `BH Tree` class derived from `Tree`) as an example to illustrate the VGDS framework.

4.1 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. Fast algorithms have been reported in [2, 5, 10, 19]. 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. 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.

4.2 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. An example of such a recursive partition in two dimensions and the corresponding BH-tree are shown in Figure 4. 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.

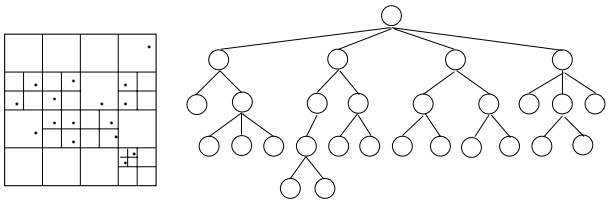

Figure 4: BH tree decomposition

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. 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.

4.3 Parallel Implementation

To make the paper self-contained, we briefly describe our parallel implementation of the BH algorithm, upon which the BH-Tree library is built.

4.3.1 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 and is used as a map to locate points in space to processors.

4.3.2 Building the BH-tree in parallel

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.

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.

4.3.3 Collecting essential data

Once the global BH-tree has been constructed it is possible to start calculating accelerations. 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 the area (called *influence area*) 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.

4.3.4 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.

VGDS employs a randomized protocol for all-to-some communication. 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 [12] 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 5 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 5: Outline of code structure

4.4 The Tree Framework

To eliminate duplicated programming investments in developing similar tree-based scientific codes, we have developed a VGDS tree frameowrk. The tree framework defines three layers of classes: *base tree layer*, *Barnes-Hut tree layer*, and *application layer*. Each latter layer is built on top of the former one. The *base tree layer* supports simple tree construction and manipulation methods. System programmers can build domain-specific tree libraries (e.g. Barnes-Hut Tree) using the classes in the *base tree layer* (Sec 4.4.2). Application programmers can write programs using classes in the *Barnes-Hut tree layer*, or any other special library developed from the *base tree layer*. Figure 6 depicts the hierarchy of tree classes and their associated methods.

4.4.1 Base tree layer

The *base tree layer* is the foundation of our framework from which complex tree structures can be derived. We define basic tree manipulation methods in the base tree layer, including inserting a new child from a leaf, deleting an existing leaf, and performing parallel tree reduction and traversal.

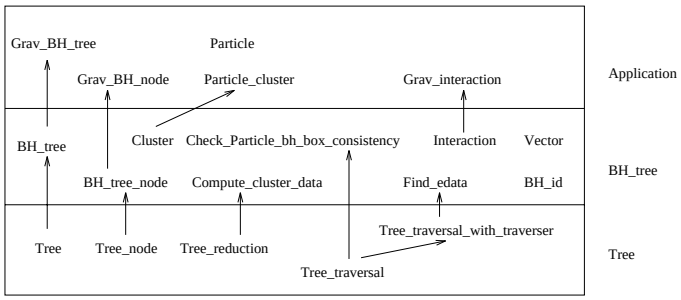

Figure 6: The class hierarchy in *base tree*, *Barnes-Hut tree*, and *application* layers.

```
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 7: Base tree layer classes.

`Tree_reduction` computes 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. `Tree_traversal` walks over the tree nodes and perform a user-defined operation (denoted as *per node function*) on each tree node (Figure 7).

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.

4.4.2 Barnes-Hut tree

The `BH_tree` 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. 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 8: 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.

Cluster data computation is implemented as a tree reduction (Figure 8). `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.

The essential data finding class `Find_edata` inherits `Tree_traversal_with_traverser` with two additional lists for essential clusters and bodies (Figure 9). 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.

After collecting the essential clusters and bodies, a body can start computing the interactions. The computation class `Interaction` (Figure 9) goes through the essential data list[1] and calls for functions to compute body-to-body and body-to-cluster interactions defined by the user of `Interaction`.

4.4.3 Application Layer

Various N-body applications can be built upon the `Bh_tree` layer. We briefly describe the implemenation of the gravitational N-body computation. 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 10). 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`.

[1] 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 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 9: Class for finding essential data and interaction computation.

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`.

4.4.4 Parallel Abstraction

Mapper class

The `Mapper` class defines a data partitioner (e.g. the ORB partitioner), a remapping method, and two associated geometry resolution functions: `data_to_processor` (that translates a data coordinate to a processor domain) and `dataset_to_processors` (that translates multiple data coordinates to a set of processor domains). In addition, it defines a simple data structure `MappingTable` to store the mapping information.

```
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 class

The `Communicator` class supports general purpose all-to-some communications. A `Communicator` class defines three methods: `extract` (that, when given a data pointer, constructs outgoing data and packs them into a `Message` object), `communicate` (that sends outgoing messages and receives incoming messages according to a randomized schedul-

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

ing algorithm), and `process` (that unpacks the received messages and performs appropriate action on the received data).

```
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 communicate();
   virtual DataPacket extract(Data*)=0;
   virtual process(DataPacket*)=0;
};
```

5 Experimental Results

The experiments were conducted on a cluster of four UltraSparc2 workstations located in the Institute of Information Science, Academia Sinica. The workstations are 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.

In the following, we report our preliminary experiences with a set of application programs, the Shallow Water code developed using the Array class, a airfoil simulation code developed using the preliminary Umesh class, and a gravitational Nbody simulation code and a vortex CFD code developed using the BHTree class. The VGDS class libraries significantly reduced the code sizes and development time of these applications (e.g. for each of the two tree codes, from over ten thousand lines down to a few hundred lines in code sizes and from over six months down to a few days in development time), compared with their message-passing C counterparts. Table 2 shows the performance of the VGDS codes. The Shallow Water code developed using the Array class achieved a speedup factor of 3.5. The Nbody code and the CFD code achieved a speedup factor of 3.2 and 3.5 respectively. In all these cases, speedup factor increases as problem size is increased. This is because communication overhead becomes less significant, compared with computation time, for large problem sizes.

Furthermore, the codes developed using the VGDS classes achieved more than 90% of the performance of their message-passing C version implementing the same algoritm. The main sources of overhead in the libraries include dynamic memory allocation/deallocation for data object creation and destruction, non-optimized computation kernel for long expressions, and additional overhead in support of portability of the library. We expect that as the project grows more mature, this overhead can be further reduced.

Shallow Water (10 iterations)

problem size	64^2	128^2	256^2	512^2	$1k^2$
seq time	0.28	1.22	4.96	21.32	88.59
parallel time	0.13	0.45	1.50	6.18	25.16
speedup	2.16	2.71	3.31	3.45	3.52

Gravitational N-body

problem size	48k	56k	64k	128k	256k
seq time	65.62	81.23	93.59	186.12	413.75
parallel time	21.03	26.17	29.17	58.78	125.78
speedup	3.12	3.12	3.17	3.12	3.23

Vortex CFD

problem size	48k	56k	64k	128k	256k
seq time	148.60	175.46	204.18	404.34	801.81
parallel time	43.00	51.37	59.05	117.20	231.07
speedup	3.42	3.42	3.46	3.45	3.47

Table 2: Execution time of the VGDS codes. Time units are seconds

6 Related Work

The benefit of data abstraction in object-oriented languages on scientific code development has been demonstrated by various efforts [9, 15]. Particularly influential and relevant to our approach are the work reported by Angus [1] and Shart and Otto[18] where class-specific compiler optimizations are introduced into a compiler written in an object-oriented fashion. Our approach has taken their class-specific philosophy further into the realm of runtime support for a diverse set of parallel and distributed data structures (beyond simply array classes) on high performance platforms.

Another line of work uses objects to define data structures with built-in data distribution capabilities. This again relates directly to our approach. Examples of work along this line include the Paragon package [8], which supports a special class PARRAY for parallel programming, the P++ Array class library [14], PC++ proposed by Lee and Gannon [11, 21], 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 [4] 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. In a related work by ourselves [6], we report abstractions of adaptive load balancing mechanisms and complex, many-to-many communications as C++ classes for supporting HPC challenging applications. Instead of tackling one particular data structure such as arrays or matrics, we propose an integrated design framework for a diverse set of distributed data structures, where data distribution, data sharing, data coherence, and synchronization between data references are mediated by the runtime system.

Our data structuring framework has similar goals and approaches to the POOMA package [3] and the Chaos++ library [17]. 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 we do. Chaos++ is a general-purpose runtime library that supports pointer-based dynamic data structures through an inspector-executor-based runtime preprocessing technique. Our framework focuses on a more specific class of data structures essential to scientific simulations and engineering computation; therefore, we are able to exploit optimizations that would be difficult for a general preprocessing technique.

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*. Our use of object-orientation is for structuring the VGDS classes and their specializations for optimizations, which is entirely distinct in philosophy from that of object-parallelism.

7 Conclusion

In this paper, we have presented the VGDS framework for scientific applications. We have implemented a prototype of VGDS base libraries and a set of distributed data structures derived from this basis, including array, unstructured mesh, and BH tree. We reported our experimental results on a workstation cluster. We demonstrated that the VGDS class libraries significantly reduced application development cost, at the expense of slight performance penalty due to object orientation. We are currently investigating possible approaches to reduing such overhead.

We hope that the basic scientific results and concrete classes and templates libraries from our VGDS effort will encourage *value-added* development of mapping and optimizing methods for classes of parallel applications.

Acknowledgment

Support for this work is provided by National Science Council of Taiwan under grant 86-2213-E-001-009.

References

[1] Ian G. Angus. Applications demand class-specific optimizations: The c++ compiler can do more. In *Proceedings of the First Annual Object-Oriented Numerics conference*, 1993.

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

[3] 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.

[4] 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.

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

[6] S. Bhatt, M. Chen, C.-Y. Lin, and P. Liu. Abstractions for parallel n-body simulations. In *Scalable High Performance Computing Conference SHPCC-92*, pages 38 – 45, Williamsburg, VA, April 1992.

[7] L.S. Blackford, J. Choi, A. Cleary, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D.W. Walker, and R.C. Whaley. SCALAPACK: A portable linear algebra library for distributed memory computers - design issues and performance. In *Supercomputing'96*, 1996.

[8] 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.

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

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

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

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

[13] Steve W. Otto. Parallel array classes and lightweight sharing mechanisms. In *Proceedings of the First Annual Object-Oriented Numerics conference*, 1993.

[14] 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.

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

[16] Petsc 2.0 for mpi. Argonne National Laboratory, Mathematics and Computer Science Division, URL: http://www.mcs.anl.gov/petsc/petsc.html, 1997.

[17] 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.

[18] Michael D. Sharp and Steve W. Otto. A class specific optimizing compiler. In *Proceedings of the First Annual Object-Oriented Numerics conference*, 1993.

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

[20] A.E. Trefethen, V.S. Menon, C.C. Chang, G.J. Czajkowski, C. Myers, and L.N. Trefethen. MultiMATLAB:MATLAB on multiple processors. Technical Report 96-239, Cornell Theory Center, 1996.

[21] 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.