VGDS: A Distributed Data Structure Framework for Scientific Computation

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

Jan-Jan Wu
Institute of Information Science
Academia Sinica
Taipei, Taiwan 11529, R.O.C.
wuj@iis.sinica.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 irregular scientific and engineering domains, focusing on computations with irregular and adaptive structures. The framework defines two base libraries: unstructured mesh and adaptive tree, that capture major data structures involved in irregular 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 network of workstations is reported.*

Keywords: parallel and distributed data structures, scientific computation, object-oriented class libraries, unstructured mesh, adaptive tree

1 Introduction

Numerous on-going efforts attempt to provide a parallel version of C++ for High Performance Computing. In recent years, these efforts have emerged to a High Performance C++ (HPC++) standard [15]. HPC++ supports various constructs to develop both task and data parallel applications. It also supports the standard high-level mechanism of transparent remote method invocation that is essential to supporting object parallelism. Despite of these sophisticated language features and library support, HPC++ has not been widely used by computational scientists in the HPC community. A key reason is that the HPC++ community has not developed sufficient scalable technology to support rich and complex data structures expressible in C++. On the other hand, 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 of Fortran's restrictions and the well understood principles of mapping arrays and regular computation to HPC platforms.

In our opinion, the crucial and timely task for parallel C++ is to develop scalable technology for supporting rich and complex distributed *non-array* data structures. The idea is to present a virtual global data structure to the user where data distribution and communication are hidden much like an HPF array. The goal is to allow the same code to be portable from uniprocessors to shared memory processors, massively parallel processors, or network of workstations (NOW).

Tree structures and unstructured meshes are two important data structures that have gained much attention in scientific computing community. For example, tree structures are oftenly used as algorithmic approximation structures in numerically intensive applications, enabling a scale of computation not attainable before. In recent years the so-called "tree codes" for N-body simulations have received wide attention. Our recent work with Rutgers University and Bellcore has led to new tree codes for multi-filament simulations for vortex dynamics of turbulent flows [7, 11].

Unstructured meshes are oftenly used in Computational Fluid Dynamics computations. The use of CFD to predict internal and external flows has risen dramatically in the past decades. In recent years, the availability of affordable high performance computers and efficient solution algorithms have led to a upsurge of interest in CFD, and indeed the technique spans a wide range of industrial and non-industrial application areas.

Having discussed the significance of tree structures and unstructured meshes, there remains the question whether these data structure classes can be made more-or-less application independent and be useful for a wide range of applications. We observed that all tree codes must resolve common issues of building, traversing, updating and load-balancing large trees in distributed memory, and all unstructured mesh codes follow similar patterns of neighborhood (adjacent edges or vertices within small number of hops) data referencing and updating and load-balancing large meshes. This motivated our effort in designing a general data structure framework for scientific computation.

We have designed a data structure framework called Virtual Global Data Structures (VGDS) for this purpose. The idea has been to provide high-level programming tools by presenting a virtual global data structure to the user where data distribution and communication are hidden much like an HPF array. The VGDS framework defines two base libraries: Graph, and Tree. The framework implements virtual global data structures as object-oriented classes with explicit associated method interfaces. Application-specific data structures and their optimizations can be derived from the base libraries through class inheritance. Currently we have constructed an unstructured mesh library from the Graph class and a Barnes-Hut tree library from the Tree class. Both data structures are useful for various scientific simulation problems.

The idea of abstracting certain scientific code into libraries has been demonstrated by various efforts [3, 4, 8, 21, 32]. Our VGDS framework distinguishes itself from others in the following aspects: (1) VGDS incorporates distributed, adaptive tree structures, which, to our knowledge, has not been attempted before. (2) Most of the existing works focus on a specific data structure. Instead of tackling one particular data structure, VGDS proposes a general data structure framework from which a diverse set of application-specific data structures can be derived. (3) We use layered object-oriented design and analysis in the construction of the VGDS base libraries. This layered approach allows easy construction and optimization of application specific data structures. It also provides a natural breakdown of responsibility in designing a complete HPC system.

In addition to programming support, the performance issue also needs be addressed as it is the key point of exploiting HPC in the first place. The challenges for virtual global data structure classes can be summarized as follows:

- The data can be irregularly structured and dynamic; as the data structure evolves, a good mapping must change adaptively.
- The data access patterns can be irregular and dynamic; the overhead of gathering needed data at runtime can be prohibitive unless done carefully.

- The number of floating point operations needed to update the value of an element can vary tremendously between elements. Therefore, good load balancing is crucial. This is a tricky issue since it involves critical tradeoffs between sometimes conflicting requirements of data locality and balance of workload.

To address these issues satisfactorily requiers novel algorithmic strategies, load balancing techniques, data coherent strategies and innovative object-oriented class design methodology. Our approach to these issues can be summarized as follows : (1) exploiting physical locality by novel partitioning strategies, (2) reducing cost of gathering needed data by prefetching, (3) maintaining data coherence by low-overhead duplication and synchronization, (4) minimizing cost of dynamic structural change by incremental remapping, (5) reducing communication overhead using aggregate communication, and (6) optimizing all-to-some collective communication to exploit characteristics of underlying machine.

The rest of the paper is organized as follows. Section 2 describes the organization and functionality of the VGDS framework. Section 3 discusses our approach to the performance issue. Section 4 uses the Tree library as an example to illustrates the core functionality of the VGDS framework. Section 5 reports our experimental results on a network of workstations. Section 6 describes related works. Section 7 concludes.

2 The VGDS Framework

The first step towards an abstraction of application codes using complex data structures is to separate data structures from their application context. A suitable representation must be chosen so that a global data structure abstraction can be seen by the user while being implemented by an actual collection of distributed parts. The VGDS framework defines three layers of C++ classes for this purpose: the global layer, the parallel abstraction layer, and the local layer. Figure 1 depicts the structure of the VGDS framework.

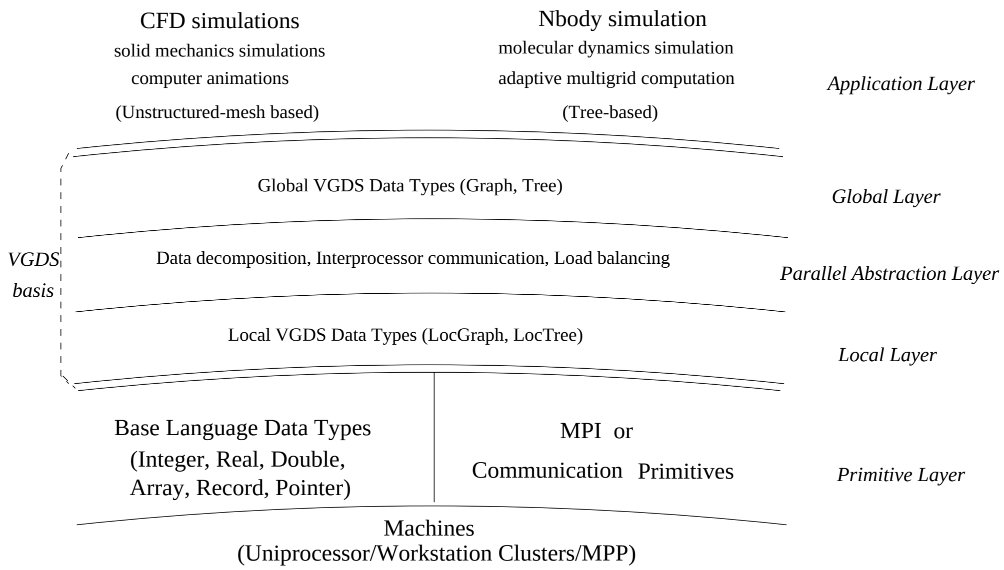

Figure 1: The VGDS framework

The global and local layers together define various virtual global operations on graph-based (such as unstructured meshes) and tree-based (such as Barnes-Hut trees) 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 implements the abstraction of parallelism, including data decomposition, interprocessor communication, and load balancing.

2.1 VGDS Classes

Base VGDS Classes. Currently, the VGDS framework defines two basic non-array data structure classes: `Graph` (for equation-solution based computation), and `Tree` (for tree-based particle simulations). We call them the Base VGDS. The Base VGDS layer is generic to multiple machines; it contains the usual constructors and a destructor for storage allocation, and methods for inserting/deleting data structure elements (e.g. tree nodes, graph nodes), building the data structure in a manner to be specified, performing computation on the data structure elements as directed by the user, and travering the data structures in various ways.

Derived VGDS Classes. Application-specific VGDS data structures can be derived from the Base VGDS classes via class inheritance. For instance, an unstructured mesh class can be derived from the `Graph` class by inheriting the `Graph` class and defining additional methods essential in unstructured mesh computation (finite element method, finite volume method, etc.). This derived VGDS layer represents specialization of Base VGDS to different target machines and different classes of algorithmic components (data coherent strategies, load balancing techniques). The goal of the Derived VGDS has been to optimize the performance of the VGDS library classes for different classes of applications on a wide range of machines while shielding its complexity from the application users.

Table 1 outlines the classes and functionality of each layer, details of which are described in the following sections.

Table 1: VGDS classes and functionalities

Layers	Classes		Functionality
	Base	Derived	
Global	*Graph*	*UMesh*	data-parallel operations
	Tree	*BHTree*	
		AdaptGrid	
Local	*LocGraph*	*LocUMesh*	processor-local operations
	LocTree	*LocBHTree*	
		LocAdaptGrid	
Parallel Abstraction	*Mapper*	*BlockPartitioner* *k-wayPartitioner* *ORBPartitioner*	data distribution and load balancing management
	Communicator	*RandomizedGather*	inter-processor communication
	Message	*FlattenMessage*	message abstractions

2.2 Global and Local Layers

A VGDS data structure at 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.

A VGDS data structure is implemented as mirrored pairs of global/local classes. When a global data object is instantiated, it creates a constituent local data object on each processor. Whenever a kernel operator associated with the data object 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 object. If communication is required, it is performed through system objects in the parallel abstraction layer.

2.3 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 METIS's k-way partitioning strategy for graph-based data structures and a *ORBPartitioner* (Orthogonal Recursive Bisection) for adaptive tree structures.

The `Mapper` class (Figure 2) defines a data partitioner (e.g. the ORB partitioner), a dynamic load balancing (remap) 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. By instantiating the *Mapper* class, the user can also construct customized data decomposition strategies. In Section 4 we will use the Tree class to show the implementation of the ORB mapper class.

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. Currently VGDS supports a message abstraction that flattens the attributes and contents of a data structure section to be communicated into a liner buffer on the sending side,

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

Figure 2: Definition of the Mapper class

and can quickly recovers the contents of the data structure on the receiving side. 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` (Figure 3). 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.

```
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;
};
```

Figure 3: Definition of the Communicator class

Figure 4 depicts the interactions between data classes and systems 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 detects 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.

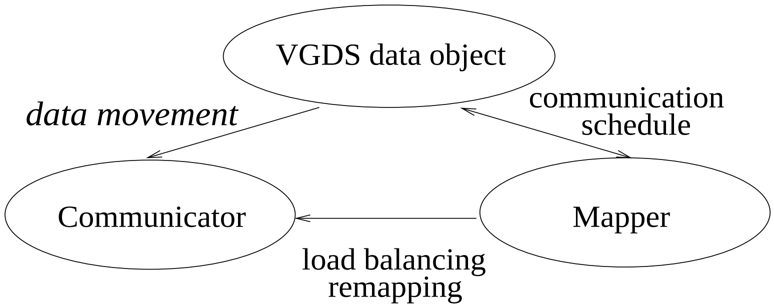

Figure 4: Interaction of classes in the VGDS framework

3 Implementation issues

This section discusses our implementation strategies for the VGDS framework. The issues include our strategy for maintaining a virtual global data structure over distributed memories and a number of optimizations that we have taken to improve the performance of the libraries.

3.1 Maintaining a virtual global data structure (data coherence)

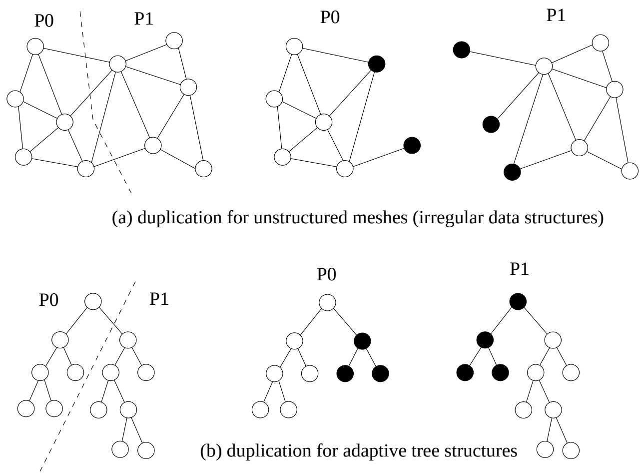

Figure 5: Duplication for distributed data structures. The duplicated data are indicated by solid black.

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 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 5 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 data coherence, 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.

Figure 6 summarizes the sequence of VGDS operations for irregular and/or adaptive scientific simulations. Note that for adaptive structures, update on the values on the master copies may change the shape of local substructures, and the data mapping may need incremental adjustments to improve load balancing.

1. Assign data to processors and build local substructures.
2. For each iteration do:
   (a) Duplicate data via communication.
   (b) Update data on the master copies.
   (c) Do computation and update local substructures.
   (d) Update data mapping if load imbalance occurs.

Figure 6: VGDS operations for irregular and adaptive scientific simulations.

3.2 Optimizations for Good Performance

Four principles are essential to achieving good performance on complex, non-array data structures: (1) exploiting data locality, (2) minimizing the cost for dynamic structural change, (3) using aggregate communication and computation, and (4) using optimized communication functions.

3.2.1 Exploiting Data Locality

Block distribution is oftenly used in array-based computation in that it keeps data that are physically close in the same processor. For complex data strutures, we need to stick to the same princple. VGDS provides a number of locality-preserving distribution methods for unstructured meshes and trees. For example, the tree library provides a ORB distribution that maps elements in a spatial domain to processors. At each recursive step, the separating hyperplan is oriented to lie along the smallest dimension so as to reduce the surface-to-volume ratio. The ORB distribution can be represented by a binary tree, a copy of which is stored in every processor and used as a map to quickly locate data elements in the procesor space.

VGDS also allow other distribution methods such as clustering techniques. Domain specialists might come up with various distribution strategies and make them available to the user as "swappable" components.

3.2.2 Minimizing Cost of Structural Change

The very nature of tree structures and unstructured meshes is dynamic and the applications using them invariably require dynamic storage management and load balancing. Instead of doing global change of data distribution to balance the work load among processors, VGDS employs incremental remapping to reduce the overhead. For example, the ORB decomposition can be maintained incrementally over the tree structure in parallel as described below.

Each ORB tree node represents a set of processors corresponding to the leaves of its subtree. We call these processors the *descendant processors* of the ORB node. Each ORB node is assigned a weight equal to the total number of operations performed in updating the states of elements over all its descendant procerssors. An ORB node is said to be overloaded if its weight exceeds the average weight of nodes at its level by a fixed percentage.

We identify ORB nodes which are not overloaded but one of whose children is overloaded; call each such node an initiator. Only the processors within the corresponding subtree participate in balancing the load for the region of space associated with the initiator. The subtrees for different initiators are disjoint so that non-overlapping regions can be balanced in parallel. At each steop of the load-balancing process it is necessary to move bodies from the overloaded child to th enon-overloaded child. This involves computing a new separating hyperplane; a binary search combined with a tree traveral on the processor's local tree determines the total weight and the new location of the hyperplane.

3.2.3 Using Aggregate Computation and Communication

The main idea of using aggregate computation and communication is that we want a working set of data to be reused by many operations before significant updates are required in the cache or local memory. VGDS provides two mechanisms to help the users to program their applications in this style. First, a set of communication methods are provided which combine multiple messages for the same destination. Second, users are given aggregate data structure traveral methods designed to do computation for multiple data elements at once. For example, a tree traveral method starting at the root of a subtree processes the nodes of the subtree at once.

3.2.4 Using Optimized Communication Functions

The communication style in computation using complex data structures can be characterized by "all-to-some" communication, in which each processor sends a set of messages to dynamically determined destination processors. There, the communication pattern is irregular and dynamically changing. Communication operations such as gather, scatter, and all-to-all/all-to-some personalized communication that are frequently used in tree codes and unstructured mesh codes all fall into this category. Efficiency of communication is a key factor to the performance of this class of applications.

VGDS employs a randomized scheduling for all-to-some communication. The scheduling algorithm 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 [19] 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.

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, 13, 28]. 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 7. 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.

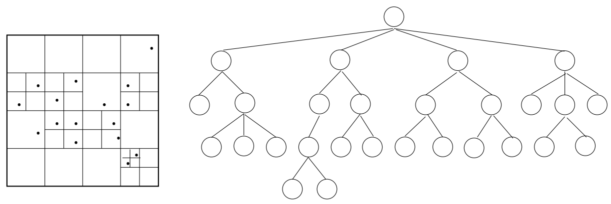

Figure 7: BH tree decomposition

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.

Figure 8 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 8: 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 9 depicts the hierarchy of tree classes and their associated methods.

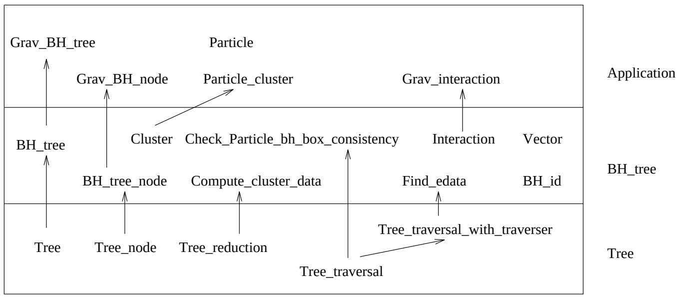

Figure 9: The class hierarchy in *base tree*, *Barnes-Hut tree*, and *application* layers.

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.

```
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 10: 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 10).

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 11: 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 11). `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 12). 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.

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

The computation class `Interaction` (Figure 12) 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 13). Next, in the `Particle_cluster` class we define the methods for computing/combining center of mass and the methods for testing essential data.

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

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

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

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

4.4.4 Parallel Abstraction

ORBPartitioner class

The `ORBPartitioner` class 15 inherits the *Mapper* class and defines functions that are specialized for the orthogoal recursive bisection partitioning method. The two associated geometry resolution functions: `data_to_processor` translates a particle coordinate to a processor) and `dataset_to_processors` translates the coordinates of a box of particles to a set of processors.

RandomizedGather class

The communication phases in tree codes and unstructured mesh codes can 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. The `RandomizedGather` class is implemented by inheriting the *Communicator* class and specializing it with a randomized scheduling algorithm for sending/receiving messages.

5 Experimental Results

The experiments were conducted on a cluster of four UltraSparc2 workstations. 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 experimental results with a set of application programs: a

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

Figure 14: One simulation step of gravitational N-body

airfoil simulation code developed using our unstructured mesh class library, a gravitational force simulation code and a vortex CFD code developed using our BHTree class library. The VGDS class libraries significantly reduced the code sizes (e.g. for each of the two tree codes, from over ten thousand lines down to a few hundred lines) and the development time (from over six months down to a few days) of these applications, compared with their message-passing C counterparts. Table 2 shows the performance of the VGDS codes. The gravitational force simulation 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 increases. 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 uniprocessor 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.

6 Related Work

The benefit of data abstraction in object-oriented languages on scientific code development has been demonstrated by various efforts [12, 23]. Particularly influential and relevant to our approach are the work reported by Angus [1] and Shart and Otto[26] 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 [22], PC++ proposed by Lee and Gannon [18, 32], which consists of a set

```
class ORBPartitioner : public Mapper<class Data, class DataSet, class ProcessorDomain,class MappingTable>
{
  int root_index;
  int P;                        // number of processors
  Bisector bisector[MAXP];
  void box2pset_helper(int, Box*, ProcessorSet);
  void dataset_to_processors(DataSet* box, ProcessorDomain ps) {
     for (int p = 0; p < MAXP; p++)
         ps[p] = false;
     box2pset_helper(root_index, box, ps);
   }
   ....
};
void ORB::box2pset_helper(int bisector_index, DataSet* box, ProcessorDomain ps)
{
  int dim;
  Real pos;
  Bisector *b;
  if (bisector_index <= 0) {    // leaf of ORB tree
    ps[-bisector_index] = true;
  } else {
    b = &bisector[bisector_index];
    dim = b->dim;
    pos = b->pos;
    if (box->start.cord[dim] > pos)
        box2pset_helper(b->larger_side, box, ps);
    else if (box->end.cord[dim] < pos)
        box2pset_helper(b->smaller_side, box, ps);
    else {
       DataSet smaller_side_box, larger_side_box;
       smaller_side_box = larger_side_box = *box;
       smaller_side_box.end.cord[dim] = larger_side_box.start.cord[dim] = pos;
       box2pset_helper(b->smaller_side, &smaller_side_box, ps);
       box2pset_helper(b->larger_side, &larger_side_box, ps);
    }
  }
}

ProcessorDomain ORBPartitioner::data_to_processor(Data* p)
{
  Bisector *b;
  int dim, b_index = root_index;
  while (b_index > 0) {
      assert(b_index < P);
      b = &bisector[b_index];
      dim = b->dim;
      if (p->cord[dim] >= b->pos)
          b_index = b->larger_side;
      else
          b_index = b->smaller_side;
  }
  assert(-b_index < P);
  return(-b_index);
}
 ...
}
```

Figure 15: The ORBPartitioner class

Airfoil Simulation

problem size	36864	65536	147456	262144	589824
sequential time (C)	1.08	1.77	4.17	7.15	16.59
parallel time (VGDS)	0.38	0.61	1.41	2.38	5.33
speedup	2.84	2.90	2.93	3.00	3.09

Gravitational N-body

problem size	48k	56k	64k	128k	256k
sequential time (C)	65.62	81.23	93.59	186.12	413.75
parallel time (VGDS)	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
sequential time (C)	148.60	175.46	204.18	404.34	801.81
parallel time (VGDS)	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

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 a general 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, 17] and the Chaos++ library [25]. The POOMA framework defines a 5 layer class structure: application layer, application component layer, global layer (field, particle,matrix), local layer (lfield,lparticle,lmatrix), and parallel abstract layer (comm, i/o, vnode, vnodemanager, data layout). A global data structure is mapped onto Vnodes via a Data Layout class mapping. Then, one or more Vnodes are mapped onto a Pnode via a VnodeManager mapping. This looks very much like the structure of the Nexus run-time system which provides support for CC++ [] where multiple threads are mapped onto a context (Vnode in POOMA) and multiple contexts are mapped onto a physical node. Up to this time, POOMA has focused mostly on array-based computations that exhibit regular reference patterns.

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.

The only tree code implementation efforts related directly to ours are by Salmon and Warren at Caltech. The two approaches to implementing distributed data structures explore different points in a space-time tradeoff. While we use a controlled amount of redundant storage to save time, they minimize the storage requirements, at the expense of efficiency and programmability. For multi-filament vortex dynamics applications it is unclear whether their approach can be made to work because of subtleties in maintaining the closed filament structures, and also because the number of particles is time-dependent.

Several parallel implementations of the algorithms mentioned above have been developed. Salmon [24] implemented the Barnes-Hut algorithm on the NCUBE and Intel iPSC, Warren and Salmon [29] reported experiments on the 512-node Intel Touchstone Delta, and later developed hashed implementations of a global tree structure which they report in [30]. Board and Leathrum [16] have implemented the 3D adaptive fast multipole method on shared memory machines including the KSR [16], Zhao and Johnsson [33] describe a non-adaptive 3D version of Greengard's algorithm on the Connection Machine CM-2, and Singh etal. [27] have implemented the 2-dimensional adaptive method on the Stanford DASH. Nyland, Prins and Reif [20] describe a data-parallel implementation of the 3d adaptive fast multipole method using the Proteus prototyping system. Finally, Hu and Johnsson [31] have implemented the 3D non-adaptive version of Anderson's method onthe CM-5E.

In contrast, our VGDS Tree library allows rapid development of high-performance applications code for a variety of tree codes. Indeed, our hand-written prototypes substantially outperform all the efforts mentioned, while allowing greater generality.

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*. Examples of parallel C++ projects using this paradigm include the Mentat Run-time System [14] and Concurrent Aggregates (CA) [9, 10] by Dally et al.. 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 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 time and the program sizes. The penalty due to object orientation is about 10% compared with its C-only implementation. We are currently investigating possible approaches to reduing such overhead.

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

Acknowledgement The authors would like to thank Marina Chen for initiating the idea of the VGDS framework, Sandeep Bhatt for valuable comments on the adaptive tree library.

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

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

[10] A. A. Chien and W. J. Dally. Experience with concurrent aggregates (CA): Implementation and programming. In *The 5th Distributed Memory Computing Conference, Vol. II*, pages 1040–1049, Charleston, SC, April 1990. IEEE.

[11] 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.

[12] 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.

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

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

[15] E. Johnson and D. Gannon. *HPC++ Reference Manual.* Department of Computer Science, Indiana University, 1994.

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

[17] S. Karmesin, J. Crotinger, J. Cummings, S. Haney, W. Humphrey, J. Reynders, S. Smith, and T.J. Williams. Array design and expression evaluation in POOMA. In *ISCOPE 98*, 1998.

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

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

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

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

[22] 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.

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

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

[25] 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.

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

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

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

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

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

[31] S. Johnsson Y. Hu and S.H. Tseng. High performance fortran for highly irregular problems. In *ACM Conference on Principles and Practice of Parallel Programming*, 1997.

[32] 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.

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