VGDS: An Object-Oriented Framework for Distributed Scientific Computing

Jan-Jan Wu
Institute of Information Science
Academia Sinica
Taipei, Taiwan

Pangfeng Liu
Department of Computer Science
National Chung Cheng University
Chia-Yi, Taiwan

Marina Chen
Computer Science Department
Boston University
Boston MA, U.S.A

James Cowie
Cooperating Systems Corporation
Chestnut Hill, MA, U.S.A

Abstract *Two major engineering bottlenecks in the production pipeline for High Performance Computing software result from a shortage of adequate design tools and design theory. We propose one technology that can help eliminate the HPC software bottleneck: object-oriented construction of virtual global data structures (VGDS). In this paper, we give an overview of the VGDS framework. Our VGDS effort focuses on developing a general purpose, distributed environment that will allow fast prototyping of a diverse set of simulation problems in scientific and engineering domains, including regular, irregular, and adaptive problems. 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.*

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

1 Introduction

Two major engineering bottlenecks in the production pipeline for High Performance Computing software are the results of the shortage of adequate design tools. First, the expanding HPC software market requires analytical tools that can help identify tradeoffs between specificity and portability, and optimize applications' development cost and their performance. Secondly, the evolving HPC hardware market demands the ability to rapidly and flexibly synthesize new tools from old ones, to track the evolution of high performance targets. Eliminating design and implementation bottlenecks requires new technologies that can resolve both sides of this dichotomy between analysis and synthesis. We have identified and nurtured one such enabling technology for HPC: object-oriented construction of *virtual global data structures (VGDS)*.

Intelligent compilers and hand-coders exploit parallelism by

1. breaking problems into data-parallel, vectorizable inner loops supported by task-parallel top-level task dispatch and outer-loop synchronization,

2. laying out the data across the local memories of processors, while maintaining the original problem's global data coherence (either manually, or assisted by the system or hardware)

3. calling for help where necessary, either by using efficient parallel library routines, or by taking advantage of automatic program transformations supplied by the compiler.

Our approach proceeds from the observation that the identification of these *dichotomies* (task vs. data parallelism, processor-local vs. system-global views of data, and compiler vs. library-based optimization strategies) and the ability to exploit them to extract high performance, is a central challenge of HPC software. We have identified distributed data structures, implemented as object-oriented classes with explicit associated method interfaces, that form a key crossing-point for each of these dichotomies. Because they define their own data and methods, distributed data structure classes can have both data-parallel and task-parallel aspects. With appropriate interface design, they can be viewed from both global (where processors are invisible to the user) and local (within a processor) perspectives. Their access patterns can be automatically transformed by a compiler, or reduced to optimized library calls. We call distributed data structures that provide both global and local views virtual global data structures.

The benefit of data abstraction in object-oriented languages on parallel software development has been demonstrated by various efforts [8, 16]. Research efforts in providing suitable object-oriented parallel languages/libraries for certain classes of applications have also been abundant [3, 5, 2, 14, 20]. 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 a broad base of scientific, engineering, and commercial simulations. 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 such as computational fluid dynamics simulations and N-body simulations, 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.

In this paper, we report on the progress of our VGDS effort. We first describe the organization and functionality of the VGDS framework. We then discuss some implementation details of virtual global data structures in distributed-memory environments. Finally, we use the Array library to illustrate the core functionality of the VGDS framework.

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.

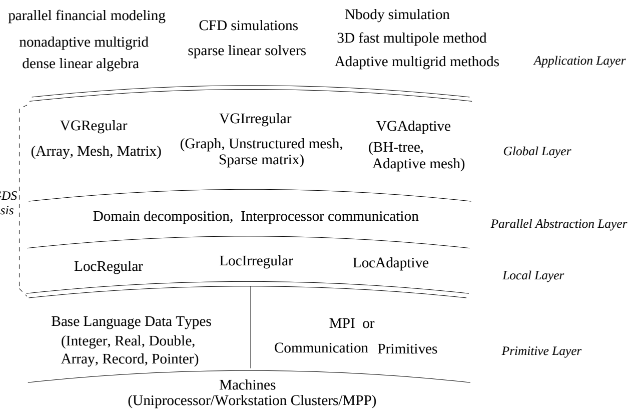

Figure 1: The VGDS Framework

The global and local layers together define a variety of virtual global operations on regular (such as arrays), irregular (such as graphs, unstructured meshes), and adaptive (such as adaptive trees, and adaptive meshes) 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. This layered approach provides a well defined development interface; application developers can use classes in the global layer without worrying about the implementation details of the layers below, while library developers can concentrate on optimizations without worrying about the applications to be implemented above. However, the library users can also go directly to the lower layers to use local computational kernels and communication classes in the framework. 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	VGRegular	Array Matrix	data-parallel operations
	VGIrregular	UMesh	
	VGAdaptive	BHTree AdaptMesh	
Local	LocRegular	LocArray LocMatrix	processor-local operations
	LocIrregular	LocUMesh	
	LocAdaptive	LocBHTree LocAdaptMesh	
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 (i.e. The framework adopts the Single Program Multiple Data model).

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.

The framework defines three base class libraries: *VGRegular* (*LocRegular*), *VGIrregular* (*LocIrregular*), and *VGAdaptive* (*LocAdaptive*), that capture common functionalities of data structures in regular, irregular, and adaptive applications, respectively. For instance, regular data structures can be characterized by their regular shapes (multi-dimensional index space), standard data mapping (block, cyclic distribution), uniform operations, and neighborhood communications. Concrete data structures can be derived from these base classes. Section 4 will elaborate on this using the *Array* class as an example.

2.2 Parallel Abstraction Layer

The parallel abstraction layer includes classes to enable data layout, interprocessor communication, and load balancing for virtual global data structures consisting of local data objects. 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 among 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 provides services 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, the framework supports a *BlockPartitioner* for regular data structures 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. To send data to another processor, first a *Message* object is created and the data to be sent are flattened and packed into the message object. Then the message object is sent to a *Communicator* object for delivery. When received by the destination, the message is unpacked into a data object of the matched type.

The *Communicator* class provides a generic interface to the message-passing primitives (or MPI implementation) supported by the underlying architecture. It provides routines for sending message objects to destination processor, and for receiving message objects from a sending processor. By encapsulating architecture dependent communication primitives into the *Communicator* class, the VGDS framework is portable across different architectures.

Figure 2 depicts the interactions between these classes.

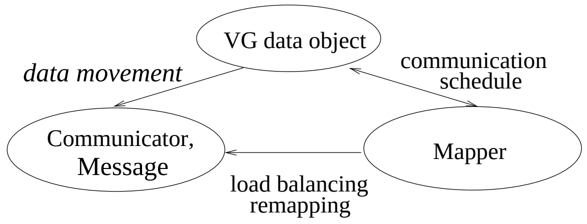

Figure 2: Interaction of Classes in the VGDS Framework

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 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.

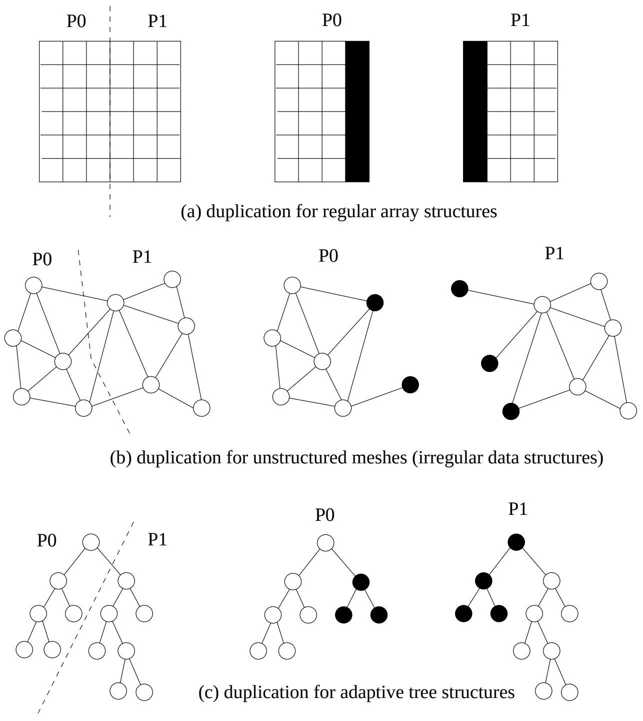
P0
P1
(a) duplication for regular array structures
(b) duplication for unstructured meshes (irregular data structures)
(c) duplication for adaptive tree structures

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

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. However, for efficiency reasons, it is desirable to reduce the number of global synchronizations between method invocations whenever possible. The VGDS caller can use *Synchronization Relaxation* wrappers around blocks of invocations which are known to be non-interfering. This will allow the user to relax synchronizations around noninterfering block computations in regular arrays, between distinct submeshes in an unstructured mesh, and distinct subtrees below some level of a distributed adaptive tree.

The master copies must be duplicated periodically. When the data dependency and distribution are static, e.g. static unstructured meshes, we only have to allocate storage for the duplicated data once during the entire execution, then update its value once the master copy is changed. However, when the data distribution or dependency is not static, the storage for duplicated data must be dynamically allocated, or even deallocated. Nevertheless, the principle of "read-only duplication, exactly one master copy" remains the same.

4 Case Study: Arrays

In this section, we use *VGRegular* and Array classes to illustrate the VGDS framework.

A regular data object is characterized by its uniform shape (most likely a Cartesian space), uniform data decomposition strategies (e.g. block, cyclic), and operations to be performed uniformly over its data elements. The VGDS Regular framework contains classes that work together to provide this functionality: *CartesianSpace* (that captures the size and shape of a Regular object), *VGRegTemplate* (that allows Regular objects to be aligned to each other and to share a common data distribution strategy), and *VGRegular* (that defines the creation of and the associated operations for a regular data structure). All of these classes are implemented as mirrored pairs of Global/Local classes. The class *Mapper<CartesianSpace>* supports standard block and cyclic distributions and user defined partitioning methods.

VGRegTemplate classes have functionality similar to the `TEMPLATE` directives in HPF. Regular data objects may be aligned to a common *VGRegTemplate* object, which in turn can be partitioned using either user defined or default partitioning strategy. A *VGRegTemplate* object does not allocate memory space; it only serves as an abstract index space that maintains interconnection geometry information (e.g. neighbor processors, size and shape of local substructure) of the data objects that align to it.

```
template<class ElTy,
         class Mapper<CartesianSpace>>
class VGRegTemplate : public LocRegTemplate<ElTy>
 {
  CartesianSpace *my_substructure;
  Processor_id  *my_neighbors;
  // dataspace is the template space,
  // processors is the processor space,
  // partitioner is the data decomposition class
  VGRegTemplate (CartesianSpace &dataspace,
              CartesianSpace &processors,
              Mapper<CartesianSpace> partitioner)
  { // determine local subgrids and
    // neighbor processors
   *my_substructure=partitioner.compute_my_role
                         (dataspace,processors);
   *my_neighbors=partitioner.compute_my_neighbors
                         (dataspace,processors);
  }}
```

VGRegular classes capture the allocation, deallocation and general behavior of regular data objects, which themselves are the aggregation of more primitive data types (denoted by ElTy), such as integer, single- and double- precision floating point, etc. When created, a *VGRegular* object is aligned to a given *VGRegTemplate* object. The constituent local Regular object is then created according to the substructure information cached in the *VGReg -Template* object. The *VGRegular* classes are also facilitated with a set of member functions for assignment, arithmetic, copying, printing, subscripted references, and data movement operations.

```
template<class ElTy>
class VGRegular : public LocRegular<ElTy>
{
   VGRegular(VGRegTemplate<ElTy> &template)
    : LocRegular<ElTy>
         (*(template.my_substructure)) {};
   // [subshape] operation for section references
   VGRegular<ElTy> &operator[](CartesianSpace *ss)
   // per-element operations are delegated
   // to LocRegular
      (e.g. +,-,*,/, logical ops, arith. ops, etc.)
   ... and many other member functions
}
```

The *Mapper<CartesianSpace>* **class** provides generic interface to data decomposition implementations for regular data structures. Two virtual functions are defined: *get_neighbors* (which collects the processors that are neighbors of the local substructure in the global interconnection geometry, and *subscript_send_recv*, which computes the send/receive pairs for subscripted assignment operations. The following shows a contiguous block partitioner class `CSBlock` that is derived from *Mapper<CartesianSpace>* class.

```
class CSBlock
    : public Mapper<CartesianSpace>
{
 public:
    int *get_neighbors
           (RegGeometry<CartesianSpace> &vrt)
    { ...return a list of processor ids };
    CartesianSpace **subscript_send_recv
      (RegGeometry<CartesianSpace> &lhs,
                   CartesianSpace **&recv_set,
       RegGeometry<CartesianSpace> &rhs,
                   CartesianSpace **&send_set)
}
```

Array Classes are constructed by instantiating *VGRegular* classes with another set of member functions that provide common functionality for HPF-like array-based computation. These include intrinsic array operations (e.g. reduction, spread, array transpose), and composite arithmetic operations (e.g. multiply-and-add) that can be performed efficiently on many HPC platforms.

The following code segment implements the Shallow Water Equations Solver using the *Array* class.

```
// size and shape declarations for
// the template (tspace) and processors (pspace)
CartesianSpace tspace (Intvl(1,m),Intvl(1,n));
CartesianSpace pspace (Intvl(1,p),Intvl(1,q));
// template is partitioned into blocks
VGRegTemplate<double> T (tspace, pspace, CSBlock);
// By aligning to T, all these arrays have
// the same shape and data distribution
Array<double> u(T),v(T),p(T),unew(T),vnew(T),
    pnew(T),uold(T), vold(T), pold(T),cu(T),
    cv(T),z(T),h(T),p_temp(T),u_temp(T),
    v_temp(T),T1(T),T2(T),T3(T);
... initializations
// start the periodic condition
// iterate itmax times
while (ncycle<itmax) {
// cshift array p at the second dimension by
// one element and assign the result to array cv
    cv.recv_cshift(&p,2,-1);
// arithmetic operations
    cv = .5*(p+cv)*v;
    p_temp.recv_cshift(&p,1,-1);
    p_temp += p;
    u_temp.recv_cshift(&u,1,1);
    v_temp.recv_cshift(&v,2,1);
    h = p + 0.25 *
        (u_temp*u_temp+u*u+v*v+v_temp*v_temp);
    ...
    unew.recv_cshift(&z,2,1);
    T2.recv_cshift(&v_temp,2,1);
    T3.recv_cshift(&h,1,-1);
    unew = uold +
         tdts8*(unew+z)*(v_temp+T2)-tdtsdx*(h-T3);

    vnew.recv_cshift(&z,1,1);
    T2.recv_cshift(&u_temp,2,-1);
    T3.recv_cshift(&h,2,-1);
    vnew = vold -
         tdts8*(vnew+z)*(u_temp+T2)-tdtsdy*(h-T3);

    T1.recv_cshift(&cu,1,1);
    T2.recv_cshift(&cv,2,1);
    pnew = pold-tdtsdx*(T1-cu)-tdtsdy*(T2-cv);
    ...
    u = unew; v = vnew;  p = pnew;

     };  // end while
```

5 Related Work

Virtual Shared Memory for HPC Systems The goal of our virtual global data structures approach, namely, making programming HPC systems easier, is similar to that of hardware [12, 19] or operating systems [13, 10] approaches to support virtual shared memory on top of physically distributed memory machines or distributed shared memory architectures. While underlying hardware mechanisms and operating systems techniques would be helpful and complementary to our programming level approach, VGDS focuses on exploiting the additional, domain-specific locality information which may not be obtained and exploited readily by the generic techniques employed by cache and virtual memory systems.

Object-Oriented Approaches for Parallelism The benefit of data abstraction in object-oriented languages on scientific code development has been demonstrated by various efforts [8, 16]. Particularly influential and relevant to our class-specific VGDS 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 VGDS approach has taken their class-specific philosophy further into the realm of runtime support for a diverse set of shared 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 VGDS approach. Examples of work along this line include the Paragon package [5], which supports a special class PARRAY for parallel programming, the A++/P++ Array class library [15], PC++ proposed by Lee and Gannon [11, 20], 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 [3] 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 [4], we report abstractions of adaptive load balancing mechanisms and complex, many-to-many communications as C++ classes for supporting HPC challenging applications. Otto [14] describes a light-weight sharing mechanism to support applications that use adaptive meshes. Instead of tackling one particular data structure such as arrays, VGDS proposes an integrated, layered object-oriented 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 VGDS effort has similar goals and approaches to the POOMA package [2] 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 VGDS does. Chaos++ is a general-purpose runtime library that supports pointer-based dynamic data structures through an inspector-executor-based runtime preprocessing technique. The VGDS framework focuses on a more specific class of data structures essential to scientific simulations and engineering computing; therefore, VGDS is able to exploit optimizations that would be difficult for a general preprocessing technique.

In addition to the above work on object-oriented parallelism which has influenced ours, a large body of work in the literature can be categorized as "object-parallelism," where *objects* are mapped to *processes* that are driven by *messages*. If a message is sent in between two processes residing on two different processors, this message will be implemented via inter-processor communication. Examples of parallel C++ projects using this paradigm include the Mentat Run-time System [9], Concurrent Aggregates (CA) [6] by Dally et al., and VDOM by [7]. Our use of object-orientation is for structuring the VGDS classes and their specializations for optimizations, debugging, profiling, etc., which is entirely distinct in philosophy from that of object-parallelism.

6 Conclusion

We have implemented a prototype of VGDS base libraries and a set of distributed data structures derived from this basis, including Array, Graph, and BHTree. The current prototype supports a set of predefined data partitioning strategies, including block partition for regular data structures and ORB partition for irregular and adaptive data structures. We plan to incorporate other commonly used data partitioning and user-defined partitioning strategies.

Our preliminary experiences with two application programs, the Shallow Water code developed using the Array class and a gravitational Nbody simulation code developed using the BHTree class, on a network of four Ultra Sparc workstations are quite encouraging. The VGDS class libraries significantly reduced the code sizes and development times of these two applications. The performance of the Shallow Water code developed using the Array class achieved a speedup factor of 3.5, and achieved more than 90% of the performance of a message-passing version hand-coded by experienced programmers. The Nbody code achieved a speedup factor of 3.4, where the uniprocessor time was measured by running the sequential program written by Barnes and Hut on a single workstation. The main sources of overhead in the libraries include dynamic memory allocation/deallocation for data object creation/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 and more efficient implementation of MPI becomes available, this overhead can be further reduced.

Acknowledgment

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

References

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

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] P. Keleher, A. Cox, S. Dwarkadas, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *1994 Winter USENIX Conference*, pages 115–132, 1994.

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

[12] Daniel E. Lenoski et al. The directory-based cache coherence protocol for the DASH multiprocessor. In *the 17th Annual International Symposium on Computer Architecture*, pages 148–159, 1990.

[13] Kai Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD thesis, Department of Computer Science, Yale University, December 1986.

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

[15] 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.

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

[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] J. P. Singh, W.-D.Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[20] 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.