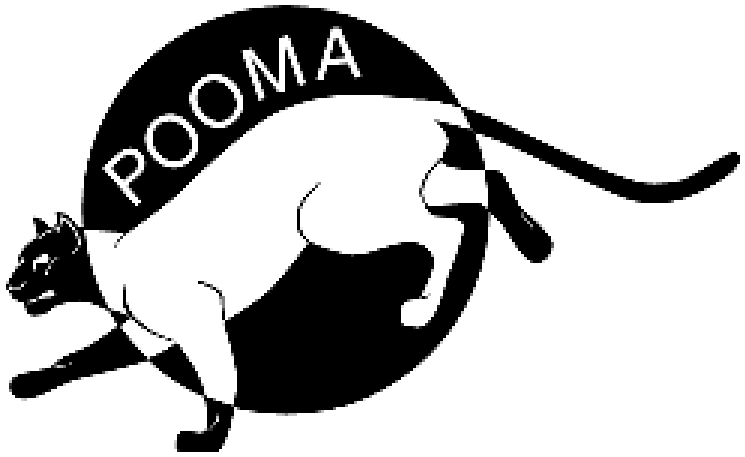
POOMA: A Framework for Scientific Simulation on Parallel Architectures

John V. W. Reynders
Los Alamos National Laboratory, Los Alamos, NM
and
Paul J. Hinker
Dakota Software Systems, Inc., Rapid City, SD
and
Julian C. Cummings,
Los Alamos National Laboratory, Los Alamos, NM
and
Susan R. Atlas,
Parallel Solutions, Inc., Santa Fe, NM
and
Subhankar Banerjee,
New Mexico State University, Las Cruces, NM
and
William F. Humphrey,
Univeristy of Illinois at Urbana-Champaign, IL
and
Steve R. Karmesin,
California Institute of Technology, Pasadena, CA
and
Katarzyna Keahey,
Indiana University, Bloomington, IN
and
M. Srikant,
New Mexico State University, Las Cruces, NM
and
Marydell Tholburn,
Los Alamos National Laboratory, Los Alamos, NM



1 Introduction

The Parallel Object-Oriented Methods and Applications (POOMA) FrameWork is a C++class library designed to provide a flexible environment for data-parallel programming of scientific applications. The FrameWork defines an interface in which the users, who need not be familiar with object-oriented programming, express the fundamental scientific content and/or numerical methods of their problem (optionally with hints as to how to best decompose it across processors). Objects within the POOMA FrameWork perform the necessary data decomposition and communications.

The POOMA FrameWork is constructed in a layered fashion, in order to exploit the efficient implementations in the lower levels of the FrameWork, while preserving an interface germane to the application problem domain at the highest level. Thus, it is possible to alter underlying implementations with no changes to the high-level interface. This is our approach to the encapsulation of parallelism within an object-oriented programming system. For current information on the status of the POOMA FrameWork, we have provided a POOMA Home Page on the World Wide Web. Our URL is `http://www.acl.lanl.gov/PoomaFramework`.

This work has been supported primarily by the Department of Energy, Office of Scientific Computing. Results were obtained by utilizing resources at the Advanced Computing Laboratory in Los Alamos, the National Energy Research Supercomputer Center, and the Maui High-Performance Computer Center. We would like to thank David Forslund, Dan Quinlan, Jeff Saltzman, and David Kilman for many helpful discussions during the development of the POOMA FrameWork.

2 History and Philosophy

The POOMA FrameWork was inspired by the Numerical Tokamak community's need to resolve the Parallel Platform Paradox, which states:

> The average time required to implement a moderate-sized application on a parallel computer architecture is equivalent to the half-life of the latest parallel supercomputer.

Although a strict definition of "half-life" could be argued, no computational physicist in the fusion community would dispute the fact that most of the time spent in implementing parallel simulations was focused on code maintenance, rather than on exploring new physics. Architectures, software environments, and parallel languages came and went, leaving the investment in a new physics code buried with the demise of the latest supercomputer. There had to be a way to preserve that investment.

The POOMA FrameWork grew out of the Object-Oriented Particle Simulation (OOPS) class library [Reynders *et al.* 1994, Reynders *et al.* 1995] developed at Los Alamos specifically for particle-in-cell (PIC) simulations [Birdsall & Langdon 1985] of fusion plasmas using gyrokinetic methods [Lee 1987]. Performing efficient PIC simulations is notoriously difficult on parallel architectures [Decyk 1995]. PIC codes written with objects from the OOPS class libraries, however, allowed PIC simulations to move between parallel architectures with no change to the source code. Furthermore, the high-level, data-parallel representation of particle aggregates with OOPS objects provided several performance enhancements over previous object-oriented PIC simulations [Forslund *et al.* 1990].

The POOMA FrameWork extended the ideas of the OOPS classes to include a variety of high-level, parallel data types and greater functionality. The main goals of the POOMA FrameWork include:

1. Code portability across serial, distributed, and parallel architectures with no change to source code
2. Development of reusable, cross-problem-domain components to enable rapid application development
3. Code efficiency for kernels and components relevant to scientific simulation
4. FrameWork design and development driven by applications from a diverse set of scientific problem domains
5. Shorter time from problem inception to working parallel simulations

2.1 Why Data Parallel?

When using explicit message passing, programmers must manage both the details of data layout across processor memories and the movement of data between them. In a data-parallel programming system [Hillis & Steele 1986], responsibility is delegated to a run-time system or a layer of objects responsible for parallel abstractions. Data-parallel systems encourage programmers to develop algorithms appropriate for the large data sets, which are the usual target of parallel scientific applications. Attempting to parallelize a serial code is much easier if the programmer considers a large number of processors at the outset.

Another strong argument for data-parallel programming is that encapsulation at the data level is typically equivalent to encapsulation at the mathematical level. Our experience is that data parallelism exposes the natural mathematical structure of a code, extracting it from layers of `do` loops. Operations on data-parallel objects can be encapsulated as parallel operators. Structuring a code in terms of data-parallel objects and operators dramatically increases code readability and correspondence with the original equations. It also facilitates identification of computational primitives suitable for optimization. The overhead incurred by use of an object-oriented framework to provide data parallelism can be offset by the efficiency gained in its ability to chain together mathematical operations.

2.2 Why a Framework?

A framework provides an integrated, layered system of objects. Each object in the framework is composed of or utilizes objects from lower layers. In the POOMA FrameWork, the upper layers contain global data objects that are abstractions of scientific problem domains (i.e., particles, fields, and matrices) and typical methods performed on these objects, such as binary operations, Fourier transforms, or Krylov solvers. Objects lower in the FrameWork capture the abstractions relevant to parallelism and efficient node-level simulation, including communication, domain decomposition, chained-expression optimization, and load balancing. The higher-level objects in the FrameWork are principally bookkeepers that delegate computational tasks to these lower layers.

This layered approach to object-oriented analysis and design provides a natural breakdown of responsibility in application development. Computer scientists and algorithm specialists can focus on the lower realms of the FrameWork, optimizing computational kernels and message-passing techniques without having to consider the application being constructed. Meanwhile, application scientists can construct numerical models with objects in the upper levels of the FrameWork, without knowing their implementation details. This clear separation of duties is made possible by the encapsulation of parallelism and application science in POOMA, which helps the programmer avoid interspersing message-passing commands and computational algorithms in the application code.

3 Implementation

3.1 Framework Layer Description

The POOMA FrameWork is composed of classes written in C++. POOMA does not utilize language extensions; rather, parallelism is captured through a hierarchical layering of classes. Furthermore, no preprocessors or interpreters are invoked, thus enabling source-level debugging. The FrameWork consists of the following five layers of classes:

- Application Layer
- Component Layer
- Global Layer
- Parallel Abstraction Layer
- Local Layer

As described earlier, the classes higher in the FrameWork represent abstractions directly relevant to application domains, whereas classes lower in the FrameWork represent the abstractions of parallelism and efficient computational kernels. The Global and Local Layers work together to define Global Data Types (GDTs) that perform matrix, field, and particle operations. The interactions between the Global and Local classes are mediated by objects from the Parallel Abstraction Layer (PAL), which is responsible for capturing the key abstractions of parallelism, such as interprocessor communication, domain decomposition, and load balancing. The Component Layer, which is built upon the Global Layer, contains a rich set of objects directly relevant to scientific simulation (such as interpolaters, FFTs, and Krylov solvers). Objects in the Component Layer are generic and reusable across problem domains, whereas objects in the Application Layer represent a configuration of Component and Global objects interspersed with application-specific objects. These highest-level objects are complete physics simulations that serve as archetypes for the process of constructing applications with the POOMA FrameWork. Applications currently under investigation include Numerical Tokamak, Molecular Dynamics, high-speed multimaterial CFD, and rheological flow simulations.

Code written with objects from within or above the Global Layer are capable of running with no source code changes on serial, distributed, and parallel architectures. We currently support most Unix-based workstations, Cray vector architectures, and MPI/PVM clusters of workstations (COWs). The supported parallel architectures include the IBM SP2, the Cray T3D, the SGI Power Challenge, and the Meiko CS2. As discussed in Section 3.6, we are also researching the extension of our Parallel Abstraction Layer to enable portability to heterogeneous MPP clusters.

A principal feature of this layered framework approach is its extensibility. The Parallel Abstraction Layer is designed for easy addition of new user-defined GDTs. If classes within the POOMA FrameWork contain functionality relevant to the target problem domain, a user may also exploit polymorphism to obtain the requisite behavior. The FrameWork provides further functionality by allowing penetration to any level. Thus, one may access and modify lower-level objects, including overloading their member functions with user-defined behavior. The FrameWork layers and their interaction are described in the sections that follow.

3.2 Global Data Types

The GDTs within the FrameWork provide the user with a data-parallel representation for a variety of data types, including fields, matrices, and particles. The design of these high-level objects has been driven by applications, and hence they have matured with a rich set of member functions directly relevant to high-performance science and engineering simulation.

Because many scientific programmers are new to object-oriented programming, the interfaces to the FrameWork's GDT objects have been designed, where possible, to seem similar to a familiar procedural, data-parallel language syntax, such as that of Fortran-90. However, this does not preclude the use of inheritance and polymorphism to create new classes that map directly to problem domains of interest. With this in mind, the FrameWork has been designed and implemented with a very shallow inheritance structure.

Each Global Data Type object is comprised of several Local Data Type objects that, as an aggregate, constitute the Global Data Type object. These Global and Local classes interact to provide simple I/O and data visualization capabilities. At compile time, architecture-dependent I/O libraries may be linked in, as well as the Advanced Computing Laboratory's portable Generic Display Library (GDL), to extend these capabilities.

Although a lot of ground can be covered by using the GDTs that are already in place and specializing their behavior through inheritance, there are hooks within the FrameWork to enable further extensibility through the explicit installation of other user-defined GDTs. The process of taking a serial class library and making it parallel is made easier through the reuse of objects from the FrameWork that encapsulate key parallel abstractions such as domain decomposition, load balancing, and communication.

3.2.1 Field Classes

The `Field` classes are N-dimensional arrays of floats, doubles, or integers. Data-parallel representation allows looped expressions, such as

```
for (int i=0; i<Nx; i++) {
  for (int j=0; j<Ny; j++) {
```

```
    for (int k=0; k<Nz; k++) {
      A[i][j][k] = B[i][j][k] + C[i][j][k] + D[i][j][k];
    }
  }
}
```

to be replaced with the single line of code

```
A[I][J][K] = B[I][J][K] + C[I][J][K] + D[I][J][K];
```

Here `I`, `J`, and `K` are `Index` objects that describe how the data-parallel array is traversed with ranges, strides, and offsets. In this simple case, with each `Field` utilizing the entire index range and a stride of one with no offset, the index notation is optional, since this is the default behavior. Both the `Field` and `Index` classes have overloaded-operator member functions to enable expressions such as

```
B[I][J][K] = A[I+1][J][K] + A[I-1][J][K] +
             A[I][J+1][K] + A[I][J-1][K] - 4.0 * A[I][J][K];
```

Here we assume that `A` and `B` are conforming `Fields` (their sizes in each dimension are identical) and the `Index` objects `I`, `J`, and `K` traverse each `Field` entirely. Code written in this data-parallel fashion is compact, easy to debug, and provides a close computational analog to the mathematical expression under investigation.

A rich set of helper objects is available to the `Field` class. For example, a `Boundary` class helps `Field` objects resolve behavior at computational boundaries. Consider the Laplacian stencil defined above: if the `Index` objects span each `Field` entirely, what happens at a `Field` border when the `Index` has an offset such as `I+1`? In this case, the `Boundary` object contained in the `Field` object is invoked and determines the boundary condition at the border. The default behavior for all `Field` classes is periodic; however, a `Field` can have any combination of periodic, Dirichlet, Neumann, or mixed boundary conditions. This provides a much cleaner representation of boundary behavior than the elaborate combinations of `cshift` and `eoshift` operations required in other data-parallel languages.

The `Field` class is enabled by a comprehensive set of mathematical and data-parallel functions. These include both parallel versions of standard mathematical functions and data-parallel operations that reduce, spread, transpose, scatter, and gather data. There are also functions that allow data-parallel relational operators to select and de-select portions of the data within a `Field` object that will be subject to manipulations within a specified scope. This functionality is similar to the `where` construct or "masking" provided by HPF and other data-parallel languages.

The `Field` class has functionality complementary to that of the A++/P++ serial and parallel array class libraries [Lemke & Quinlan 1992] developed at Los Alamos. We are currently working to merge these two classes into a single, highly-tuned array class library.

3.2.2 Banded Matrix and Vector Classes

The `Vector` class provides optimized vector operations, such as binary operations, dot products, and norms. The `NDiagMatrix` class utilizes a striped, row-compressed data format to represent banded, structured matrices, and it includes typical operations like transposing a matrix or multiplying two matrices. These classes interact to provide useful matrix-vector operations.

No single data-decompostion strategy can simultaneously be optimal for sparse, banded structured, and full matrices. Optimization of banded, structured matrix storage and operations was motivated by the current POOMA applications under investigation (Numerical Tokamak, CFD, and rheological flow), which require banded-matrix operations for their elliptic and hyperbolic equation solvers. The necessary interoperability with other GDTs in the FrameWork is enabled by member functions within the `NDiagMatrix` and `Vector` classes that convert data to and from vector or banded, structured matrix format. Work is now underway to develop separate full and sparse matrix classes for future application areas.

The responsibilities of the Global and Local class member functions performing vector and matrix operations split clearly, with global operators handling communications and storage management, and local operators performing the actual computation. For example, element access, addition, and many other operations can be performed on the data in place; hence, the global member functions are simple. Furthermore, the global and local parts of an operator vary widely in the algorithms they utilize. For example, matrix-vector multiplication, squaring a matrix, and matrix transpose use different algorithms for the optimized global communication patterns and for the optimized local computations.

3.2.3 Particle Classes

In particle simulation programs, particles are free to move about a given domain while interacting with a fixed grid. However, if each particle is not continually repositioned upon the processor holding the cells of the grid nearest to the particle position, then the simulation will become dominated by interprocessor communication. The particle classes within the POOMA FrameWork provide the message-passing capability needed to address this problem, while maintaining an expressive data-parallel syntax for coding the numerical algorithms for particle motion and interaction.

A `DPField` (Double-Precision Particle Field) object represents a particle attribute such as a position, velocity, or electronic charge. A `Particles` object, which represents a distribution of particles, contains a set of `DPField` objects that describe the particles' attributes. Both the `Particles` and `DPField` objects point to the same data, but they provide the user with two different views.

A `DPField` object's operators and member functions allow the user to operate

on separate attributes of the particles. For example, a particle position update of the `DPField` objects `x0` (old position), `x1` (new position), and `v` (velocity) with the double `delta_t` (time step) would be performed as shown below:

```
x1 = x0 + delta_t * v;
```

The `Particles` class has a member function called `swap` that is responsible for load balancing the particles as they move throughout the simulation domain. Given particle coordinate `DPFields` and drawing upon data layout information provided by objects in the Parallel Abstraction Layer of the FrameWork, the `swap` function determines which particles need to move to other processors and performs the particle swapping and memory management. Due to the expense and complexity of particle swapping, this operation has been made a high-level member function available to the user. This will aid in the understanding of program algorithms and will improve code performance.

Other member functions are provided in the `Particles` class to facilitate the calculation of forces on particles in a simulation. If one is computing forces or potentials on a grid (i.e., a particle-in-cell method), `Gather` and `Scatter` member functions are available to interpolate a field quantity to particle positions or accumulate a particle quantity onto the grid. These functions invoke an `Interpolate` class that is equipped with several common interpolation methods, and this class can query the `Field` involved for critical information, such as whether the `Field` is cell-centered or vertex-centered. For particle-particle interaction forces, POOMA provides a member function called `Interact` to accumulate forces due to particles within a given radius of each particle. This function circulates particles amongst the neighboring nodes in a pre-defined pattern, performing interprocessor communication as required and accumulating interparticle forces at each stop using a force defined in the POOMA library or a user-defined force function. By choosing the dimensions of the subdomain on each virtual node (see Section 3.3.1) to be slightly larger than the cutoff radius for the interaction, the communications pattern can be restricted to nearest-neighbor nodes only, without any loss of generality. The data structures and communication pattern that `Interact` implements correspond to the Cell Method[Beazley *et al.* 1994].

3.3 Parallel Abstraction Layer

Objects within the Parallel Abstraction Layer are responsible for capturing key features of parallel programming, such as interprocessor communication and domain decomposition, in the context of supporting GDTs composed of local (node-level) data objects. The global/local programming paradigm extends the data-parallel programming model by allowing a user to drop down to the message-passing level from within an active data-parallel program. This is accomplished through access to run-time system information on GDT object data layouts and geometries. Once the details of a data layout are known, it is

possible to establish the correspondence between individual pieces of data and the processors in a run-time partition and to manipulate that data manually, whether on-node or as arguments of explicit sends and receives. A data-parallel array descriptor provides the essential link between local and global array contents. A GDT object can thus be modified by manipulating its individual pieces at the local level.

This global/local paradigm precisely describes how POOMA implements its data-parallel interface. Using stored array-descriptor geometry information, POOMA performs an explicit conversion of data-parallel code to nodal C++ with message passing. The key abstractions of global/local parallelism are encapsulated into three groups of classes within the PAL: Global-Local Interaction classes (those managing the interactions of GDT objects with their local constituents), DataLayout classes (those responsible for data layout and processor geometry), and Communication classes (those responsible for moving data between nodes).

3.3.1 Global/Local Interaction classes

Every GDT within the FrameWork consists of a mirrored pair of global and local classes. For each instantiation of a GDT object, the Global/Local Interaction classes help to instantiate an appropriate number of type-corresponding local objects on each of the available processors. In an application code, the user calculates only with GDT objects. In turn, each member function of a global object works with PAL objects to locate the constituent local objects and invoke the corresponding local member function. Thus, the primary role of the GDT object is to act as a bookkeeper, while the calculations actually are performed by the constituent local objects.

Every Global and Local class employs a letter/envelope paradigm [Coplien 1992] and inherits from an abstract base class, which is responsible for registering and interacting with objects in the Parallel Abstraction Layer. Furthermore, a virtual constructor technique [Gamma *et al.* 1995] is employed to enable other objects within the FrameWork (such as the Chained-Expression Object described in Section 3.4) to perform high-level operations on GDT objects without having to distinguish their types.

A key abstraction within the FrameWork is that of the "virtual" node. There can be several virtual nodes on a physical processor, and a map of the virtual node IDs and corresponding physical node IDs is maintained by a PAL object called the `VnodeManager`. When a global object is instantiated, it actually spreads its constituent local objects across virtual nodes rather than physical nodes. Thus, the global and local objects know nothing about the actual number of physical processors available.

This abstraction allows us to move entire virtual nodes between physical nodes without changing properties of the global or local objects contained in a virtual node. The data configuration is made consistent by simply updating

the virtual-to-physical node map. Thus, we exploit two levels of load balancing: balancing local data between virtual nodes and balancing virtual nodes between physical nodes. Modification of the virtual-to-physical node ratio also enables the user to choose data sizes that complement machine-specific cache sizes. Code efficiency on many of the architectures on which the FrameWork runs is a highly sensitive function of data alignment in the cache.

Another advantage to storing data in a virtual node form is the simplified approach to parallel I/O. Most parallel computer vendors do not have tools for saving data from a given system partition and then reading the same data back into a system partition of a different size. By maintaining data in a virtual node form (where a number of virtual nodes greater than the maximum partition size is used), the FrameWork need only update the virtual-to-physical node map as the data is read back in.

3.3.2 DataLayout classes

The DataLayout classes are responsible for defining the geometry of the GDTs and the interconnection of their constituent local objects. An abstract base class provides the necessary hooks into the PAL, while inherited classes provide information and functionality pertinent to each GDT in the FrameWork (i.e., there is a `FieldLayout` for `Field` objects, a `NDiagLayout` for `NDiagMatrix` objects, etc.). Thus, each GDT has a layout object tuned to its needs.

DataLayout objects are composed of smaller objects such as `Neighbors`, `Offsets`, and `Sizes`. This encapsulation of DataLayout abstractions enables code reuse when DataLayouts for other data types are required for new GDTs integrated into the FrameWork.

The DataLayouts accompanying the `Field`, `NDiagMatrix`, and `Particles` classes provide a variety of archetypal layout strategies, as well as default behavior when no data layout is specified. For example, given an N-dimensional `Field` object, one is able to specify which of its dimensions are to be distributed in parallel and which are to remain on-node. Once the subset of parallel axes are known, one is then able to choose from several parallel domain decomposition archetypes (hypercubes, hyperplanes, pencils, etc.) to complete the formation of the DataLayout.

Over the course of a simulation, DataLayout objects are constructed during the construction of GDT objects. The DataLayouts mediate object operations that require interprocessor communication, and they are used to determine when two GDT objects can be used together in an expression. In the cases where the DataLayouts are the same, the objects are considered fully conforming. In cases where the global sizes of the objects are similar, but the layout across the virtual nodes is different, the objects are considered partially conforming. In this case, the DataLayout object interacts with PAL objects to generate a temporary GDT object that is fully conforming, and the data is moved to this layout before the operation is performed. When the global sizes are not the same, the objects do

not conform, and the operation cannot be performed.

3.3.3 Communication classes

A wide variety of parallel computer architectures are available today, as well as several different communication libraries for interprocessor communication and process control. To provide a portable, transparent mechanism for supporting message-passing communications on distributed, parallel, and clustered parallel architectures (discussed in Section 3.6), two classes have been implemented and used throughout the Framework: `Communicate` and `Message`.

The `Message` class is used to encapsulate data in a format that allows for easy construction and data retrieval. Each `Message` consists of `N` items, where each item is a scalar or vector of any defined data type. Routines are provided by the `Message` class to query for information on the number, size, type, and contents of the items in a `Message`. In essence, the `Message` class provides an arbitrary run-time structure that is used to hold data to be sent to or received from another node. `Message` objects can be concatenated together, copied, forwarded to other nodes, and written to or read from a file.

The `Communicate` class is a generic interface to the specific parallel communication library to be used. It is responsible for making sure the necessary processes are running on the parallel processors, and it contains the code to send and receive data between these processes. Each process is assigned a unique integer ID, from `0` to `N-1` (where `N` is the total number of parallel processes), and there may be any number of processes on each physical processor. To use the `Communicate` class to send data to another process, first a `Message` object is created and filled with the data to be sent, and then this `Message` object is given to the `Communicate` object for delivery. `Communicate` provides routines for sending `Message` objects to a destination process, and for receiving `Message` objects from a sending process. Each `Message` is sent with a user-specified identification code (tag). `Message` objects may be received in any order, and options are provided to search for pending data from a specific process or pending data with a specific tag. `Message` objects that have the same source and destination node are passed to a message queue and are never introduced to the network. This approach also makes it possible to simulate some aspects of message-passing algorithms on serial architectures through reads and writes to local memory.

Through the use of this parallel communication abstraction, the work required to port an application to a new parallel communication library is greatly reduced. Development of a new version of the `Communicate` class (for a new base communications library or a new architecture-specific system) requires only four routines to be rewritten: initialization and termination of the parallel communication environment, and sending and receiving routines for the `Message` objects. At present, versions of `Communicate` have been developed for use with PVM and MPI clusters of Unix-based workstations, SMP architectures, and the

Cray T3D, IBM SP2, and Meiko CS2 parallel supercomputers.

3.4 Chained-Expression Object

One of the most powerful features of the C++ language is the ability to overload operators for user-defined types. Unfortunately, this feature is also one of the major reasons that the performance of C++ code does not compare well to that of C or Fortran.

A simple statement that demonstrates the problem is:

```
D = A + B + C;
```

Here `A`, `B`, `C`, and `D` are user-defined classes that have a large amount of data associated with them (such as vectors or arrays). The act of performing the binary operation `+` or `=` on two objects involves a call to an overloaded, binary-operator function. Thus, the above expression involves three separate calls that are "chained" together. In our example, `A + B` is evaluated, and the result is added to `C`. The result of that is then assigned to `D`. The POOMA FrameWork views expressions like these as "chained expressions", in which a series of overloaded function calls are chained together to arrive at the answer.

In the case of elemental types (`int`, `float`, `double`, etc.), it has become possible for compilers to optimize chained expressions. C++ overloaded binary operations inhibit such optimizations by breaking chained expressions into their binary elements. For large data sets, this can can severely reduce overall performance.

Another source of performance degradation comes from the creation of temporary variables during expression evaluation. If users do not take steps to optimize default usage of copy constructors through reference counting methods and shallow copies [Coplien 1992], a temporary object is created for each binary operation performed. This is not only time-consuming, but, for large user-defined types, can cause the code to run out of physical memory (due to the temporary creation) and can lead to virtual memory swapping.

These performance penalties prompted the creation of the Chained-Expression Object (`CEO`) in the POOMA FrameWork. The main goal of the `CEO` is to recognize complex expressions and bypass multiple function calls normally required in the implementation of overloaded operators. This is accomplished by modifying the way overloaded operators are used. Instead of performing the operation specified by a given function, the code updates the `CEO` with information concerning objects involved in the operation, and the `CEO` builds an expression stack for the statement at run time. When the assignment statement is finally encountered, the `CEO` matches the expression stack with a library of tuned expression kernels. If a match is found, the expression can be evaluated directly. If not, the `CEO` "deconstructs" the expression stack into the largest possible registered kernels to complete the evaluation.

For example, consider the following code:

```
Vector A(100);   // constructs a vector with 100 items
Vector B(100);
Vector C(100);
Vector D(100);
A = 1.0;         // assigns every element of the vector to
B = 2.0;         // the double on the rhs
C = 3.0;
D = A + B + C;   // perform two + ops and the = op
```

A naïve implementation would perform the following operations:

```
Vector::Vector:Constructing vector 'A' with length of 100
Vector::Vector:Constructing vector 'B' with length of 100
Vector::Vector:Constructing vector 'C' with length of 100
Vector::Vector:Constructing vector 'D' with length of 100
Performing A = 1.000000
Performing B = 2.000000
Performing C = 3.000000
Vector::Vector:Constructing vector 'TMP1' with length of 100
Performing TMP1 = A + B
Vector::Vector:Constructing vector 'TMP2' with length of 100
Performing TMP2 = TMP1 + C
Performing D = TMP2
Destructing TMP2
Destructing TMP1
Destructing D
Destructing C
Destructing B
Destructing A
```

In contrast, by using the CEO, the POOMA FrameWork would execute the same code in a more efficient manner:

```
Vector::Vector:Constructing vector 'A' with length of 100
CEO::registering expression kernel FLDFLDadd
CEO::registering expression kernel FLDFLDaddFLDadd
Vector::Vector:Constructing vector 'B' with length of 100
Vector::Vector:Constructing vector 'C' with length of 100
Vector::Vector:Constructing vector 'D' with length of 100
Performing A = 1.000000
Performing B = 2.000000
Performing C = 3.000000
Placing token FLD on expression stack
Placing token FLD on expression stack
Placing token add on expression stack
Placing token FLD on expression stack
Placing token add on expression stack
Performing D = FLDFLDaddFLDadd
Destructing D
```

```
Destructing C
Destructing B
Destructing A
```

Each operation registers itself with the `CEO`. When `FLDFLDaddFLDadd` is found during execution of the assignment, the `Vector` pointers are passed in, and all the operations occur inside a single function call. This allows the registered functions to place several operations on a single line in a `for` loop and reap the added performance over repeated binary operations. Furthermore, no temporaries are constructed, and no extra copies are performed. If an object on the right-hand side of the expression also were to appear on the left-hand side, a temporary would be created to store the intermediate value before assignment.

3.4.1 CEO in Parallel Scientific Simulation

The `CEO` provides a powerful expression-by-expression optimization capability. The speedups gained in cache performance, CPU speedup, and reduced memory utilization are significant, while the expressive features of the GDTs are preserved. We have extended previous work on expression grouping [Parson & Quinlan 1994] to explore parallelism and chained-expression operations between heterogeneous data types. The FrameWork's use of virtual constructor techniques for its Global Data Types (described in Section 3.3.1) enables optimization of inter-type operations, providing efficient implementations of expressions such as particle gather/scatter operations on fields and stencil/field interactions.

The `CEO` plays a further role in parallel architectures by coordinating interprocessor communications and managing temporary border information, as is shown in a two-dimensional diffusion example in Section 4.1. The `CEO` defers all message passing to the `=` operator, where it then determines which `Field` objects in the expression stack require updated border information. Current research is exploring inter-expression border reuse techniques to reduce the overall message-passing requirement.

An important insight gained from parallel application development efforts is the recurrence of stencil patterns (e.g., elliptic and hyperbolic stencil operations) in a variety of problem domains. This eases the effort in maintaining a comprehensive, tuned kernel library targeted to the application domain. If exact stencil matches are not found, the expression is deconstructed into the largest possible sub-expressions contained in the expression kernel library. Even in the pathological case where the largest pattern match reduces to a set of binary operations, a substantial savings is still gained through the avoidance of unnecessary temporary creation.

This structure allows developers to "freeze" a working, high-level description of the numerical algorithms they are using, and then extensively tune the performance by writing optimized kernels for critical sections of the code. The process of adding new kernels is simple, due to the straightforward structure of

the expression kernels and their interaction with the FrameWork. The `CEO` and its supporting cast of optimized kernels are the key to how POOMA efficiently evaluates general, data-parallel statements.

Currently, we are exploiting the design of the `CEO` to implement a run-time, code-generation capability for the POOMA FrameWork. In this manner, a simulation can be run in a "debug, code-generation" mode, in which expression stubs for non-matching kernels are interpreted into code modules. Thus, after a recompilation of the target class, the code can run in an optimized mode with no expression deconstructions or intermediate temporaries.

3.5 Component Layer

One of the most time-consuming aspects of scientific code development on parallel architectures is the process of rewriting code to match native scientific libraries as one moves to a new parallel architecture. The changes can be small: a different ordering of parameters; medium: communications are required to fit the requisite parallel data layout; or large: the routine your code depended upon on architecture A does not exist on architecture B, thus requiring you to write your own.

Objects from the Component Layer encapsulate many useful routines typically found in scientific libraries and some that provide functionality unique to the FrameWork. The objects are built upon GDT objects; thus, they run with no changes on serial, distributed, and parallel architectures.

3.5.1 FFT

POOMA provides a Fast Fourier Transform (`FFT`) class for Fourier analysis of a `Field` of any dimension. The library currently includes Radix-2 Complex-to-Complex, Real-to-Complex, and Complex-to-Real FFT components. An object of this class accepts `Field` data, moves the data into "pencils" (1-dimensional arrays of data contained on a single processor) aligned along the `Field` dimension that is to be Fourier transformed, performs the FFT, and returns the data to its original layout. The data layout is altered prior to the FFT in order to minimize the interprocessor communication and increase efficiency. In order to perform an FFT on a `Field` object, an `FFT` object must be instantiated for that `Field`. This `FFT` object will know the lengths of the `Field` in each dimension and its data layout across the parallel processors, and it will pre-allocate pencils of memory for FFTs in each dimension of the `Field`. Any `Field` object that fully conforms with this `Field` (has the same dimensions, sizes, and data layout) can be Fourier transformed with the same `FFT` object.

When an FFT is requested, the user provides `Field` objects with real and imaginary components of the data, asks for a forward or inverse Fourier transform along a particular dimension of the field, and can provide a factor by

which to scale the result (in order to renormalize the resulting field data). After the data is transposed into a layout of pencils, each processor will perform 1-dimensional FFTs on all its pencils. A subroutine written in C is provided to perform these FFTs, but one may substitute other optimized routines if desired.

The code below instantiates two `Field` objects and constructs an `FFT` object. Note that we need a `Field` object to construct the `FFT` object (this construction sets up work arrays for use in FFTs).

```
Index I(64), J(64);
Field real(I,J);
Field imag(I,J);
FFT fftObject(real); // instantiate with a Field
fftObject.FFT_CC(real, imag, 1, 1, 1.0/sqrt(64)); // FFT in X-dir
fftObject.FFT_CC(real, imag, 2, 1, 1.0/sqrt(64)); // FFT in Y-dir
```

Table 1 shows performance results for 25 iterations of a 2D Complex-to-Complex FFT and inverse FFT on a 256×256 grid running on the Cray T3D.

Nodes	Seconds	MFlops	MFlops/Node
2	41.40	12.66	6.33
4	21.43	24.46	6.12
8	14.62	35.86	4.48
16	6.51	80.54	5.03

Table 1: FFT performance

3.5.2 Krylov Solvers

POOMA has a collection of scalable, preconditioned conjugate-gradient (PCG) solvers, and provides user-friendly facilities for selection of solution methods and algorithms. The iterative Krylov `Solver` object is based on a conceptual decomposition of the general task sequence that needs to be performed in PCG algorithms. The basic idea is to separate the initialization phase from the main-loop phase. The initialization phase includes the preconditioned factorization, if there is one. In the main-loop phase, the optional preconditioner solve is isolated from the rest of the computations. The FrameWork allows us to plug in different preconditioners with a conjugate-gradient algorithm without repeating any CG code.

POOMA provides a collection of CG algorithms (CG, BiCG, CGS, and BiCGStab) with a collection of preconditioners (Diagonal, Incomplete Cholesky, and Incomplete LU). A CG algorithm or a combination of a CG algorithm with a preconditioner is considered a specific solver "strategy" [Gamma *et al.* 1995]. All these strategies share a common structure for the `Solver` member function

`Solve` (initialization followed by solve), and this is defined in an abstract class called `LSStrategy`. In a PCG solve, a strategy subclass is responsible for the factorization, as well as the initialization. Efficiency is gained since various factorization schemes are kernel operators of the `NDiagMatrix` and `Vector` classes. We employ a certain class of PCGs [Gupta *et al.* 1995, Kumar *et al.* 1993] that substitute the preconditioner solve within a PCG loop by a sequence of matrix-vector multiplications, thereby enhancing scalability. After the initialization (and factorization), the strategy class interacts with the specific CG class to perform the solve.

The `Solver` class provides a user-friendly interface for setting up a solver context (e.g., selection of CG solve and preconditioner). Once a context is set up, it interacts with the specific strategy class for the chosen CG solve. The following code shows an example of using the `Solver` class for an ICCG solve.

```
int vnodes=16;
int rank=256;
// Fill up the matrix A using the discretizer object (see next subsection)
⋮
// Fill up the source vector b with appropriate values
Vector b(rank, vnodes);
⋮
// Allocate solution vector x
Vector x(rank, vnodes);
Solver L;
L.SetPrecond(IC);  // Choose preconditioner
L.SetCG(CG);          // Choose CG solver
x = L.Solve(A, b);   // Get solution vector x
```

3.5.3 Elliptic Discretization

Finite-difference solution of elliptic, partial differential equations involves two steps: first, the continuous system is discretized over a finite-difference grid, resulting in a linear matrix system, and then this linear system is solved using matrix computations. The POOMA `Discretizer` class discretizes general, second-order, elliptic, partial differential equations on a finite-difference grid in 1D, 2D, and 3D. It can handle equations with constant coefficients (e.g., the Poisson equation on a physically rectangular grid in Cartesian coordinates) or variable coefficients (e.g., the Poisson equation in a non-Cartesian coordinate system, and other general, second-order equations). Both cell-centered and vertex-centered grid discretizations are provided. Using a `Boundary` object, various boundary conditions (Dirichlet, Neumann, periodic, etc.) can be specified at different edges or faces.

A `Discretizer` object has both global and local components. The global component is used to specify the grid (dimension, number of points in each dimension, and the geometry of the grid), discretization stencil (e.g., 5 points, 7

points, etc.), and coefficients of the equation (e.g., constant coefficients and/or variable coefficients). Once the grid, the stencil, and the coefficients are specified, the global component delegates the computing job to the local components. Local components are responsible for computing the stencil weights at each grid point. The `Discretizer` generates a block N-diagonal matrix as the output. Generating the matrix requires transformation of data from one DataLayout (`FieldLayout`) to another (`MatrixLayout`). This transformation of data from one layout to another is done utilizing the PAL `Communicate` object. Such patterns of communication to transform objects from one data layout to another are very useful during a simulation (e.g., to transform a field object to a vector object and vice-versa, for an elliptic field solver).

The code below is an example of using the `Discretizer` class to discretize the Poisson equation on a square grid in 2D. The `Discretizer` generates a block N-diagonal matrix. Note that after this is done, the `Solver` described in Section 3.5.2 might then be used to solve this system.

```
int vnodes = 64;
int stencilPoints = 5;
FIndex I(256), J(256);
ArchType gridGeometry = Squares2D;
Discretizer D(I, J, gridGeometry, vnodes, stencilPoints);
Boundary *BC;
// Set proper boundary conditions on each side here
D.set_boundary(BC);
// Give coefficients of terms in partial differential equation here
D.set_coefficients(1.0, 1.0);
NDiagMatrix A(D.discretize());
```

3.5.4 Stencil Objects

The `Stencil` class provides the user with a shorthand for long expressions with a fixed set of index offsets into an array. For example, in our `Field` and `Index` notation, a three-dimensional Laplacian operation may be represented as shown below:

```
B[I][J][K] = A[I+1][J][K] + A[I-1][J][K] +
             A[I][J+1][K] + A[I][J-1][K] +
             A[I][J][K+1] + A[I][J][K-1] - 6.0 * A[I][J][K];
```

If this operation occurs several times throughout a simulation, however, the application code becomes obscured by the long expression and there is an increased chance of coding error. Furthermore, in an N-dimensional simulation on a non-orthogonal mesh, the number of stencil points required for a second-order expression is 3^N – a potentially huge expression. The `Stencil` class alleviates these concerns by storing the coefficient and offset information in a single object.

Thus, expressions such as the one above may be written in the following compact notation:

```
B[I][J][K] = Laplacian ( A[I][J][K] ) ;
```

where `Laplacian` is our 3D Laplacian `Stencil` object. This object can be reused throughout the simulation on any `Field` and is integrated with the `CEO` to optimize the inner-loop calculation and determine the appropriate interprocessor communications based on the `Stencil` offsets.

The construction of the `Stencil` is generalized to enable persistence of source code when changing the dimensionality of a simulation. Thus, a scientist is able to simulate one-dimensional behavior on a workstation, and then move the code to a parallel architecture to explore three-dimensional behavior with no changes to the structure of the source code.

The `Stencil` class includes overloaded operations that enable the construction of complex `Stencil` objects from simpler constituent `Stencil` objects. Given the differentiation `Stencil` objects `DX`, `DY`, and `DZ`, and the identity `Stencil` `I`, one can construct a `Stencil` for the Helmholtz operator and perform the operation on a `Field` with the following two lines of code:

```
Helmholtz = (DX*DX + DY*DY + DZ*DZ) + I;
Result[I][J][K]  =  Helmholtz( Source[I][J][K] );
```

As these examples show, the `Stencil` class provides a powerful mechanism for direct representation of mathematical abstraction in source code.

3.6 POOMA Simulations on Clustered Parallel Architectures

The POOMA team has been investigating the possibility of developing a distributed, parallel-programming environment that would enable spreading computation over many parallel resources at the same time and integrating parallel applications written using different tools or platforms. It is our hope that such an environment also would enable more reuse of parallel code and facilitate the implementation of distributed parallel code. Similar systems, such as the Common Object Request Broker Architecture (CORBA) [OMG 1993], have already been proposed and developed in the distributed-systems community. Unfortunately, these systems do not provide a sufficient means of describing and implementing parallel objects or data sharing between different objects, and they do not address issues important to parallel processing, such as load balancing. Our first experiment leading to the development of a system adequate for clustered parallel architectures was undertaken in the summer of 1995. This initial design is meant to be a preliminary "proof of principle" of the feasibility of such systems.

In order to enable communication between many massively parallel processing computers (MPPs), the POOMA FrameWork has been extended to include a hierarchical communication model. The `Communicate` class has been replaced with a virtual communicate class that determines the location of the receiving

virtual node with respect to the sending virtual node, and then invokes either the communication services provided by the architecture or a combination of these services and network transport to deliver the message to its destination. The virtual nodes themselves are not aware of each other's location; the assignment of virtual nodes to particular machines has been left to the programmer, who configures the virtual machine over which computation is spread. A virtual machine is composed of clusters of virtual nodes, where every physical machine corresponds to the notion of a cluster of virtual nodes.

The system interfaces with CORBA in such a way that, given servers realizing any particular application, the clients can spread their computation over the chosen set of machines, specify what resources of any particular machine they want to use (e.g., the partition size), how they want to balance their load (e.g., how many virtual nodes they want to put on that particular machine), and other initialization information. The client then uses CORBA IDL one-way functions to initiate the computation on all of these servers simultaneously. It is also possible to initialize computation without using CORBA — through an initialization object that reads in the necessary data from a file or obtains it in some other, previously agreed upon way.

Reliance on CORBA services to implement communication between clusters makes this initial system inefficient and limits the set of platforms programmers can use. The network transport will soon be replaced by a more efficient method, which will enable us to gather performance data, help identify useful scenarios, highlight problems, and offer a better understanding of the conditions and mechanisms necessary to couple distributed and parallel computing effectively. In additon to the current reliance on CORBA, the initial design rests on the assumption that all its pieces are implemented using the POOMA library. Providing full interoperability means designing an interface through which other libraries and parallel language compilers could become parts of a distributed system. Therefore, our further plans include formulating a Parallel Interface Definition Language, which would make integrating objects written in parallel languages into the system feasible. We also intend to address the issues of reliable security mechanisms and efficient parallel processing in the final version of the system.

4 POOMA Appearance

In this section, we provide the flavor of codes written with objects from the POOMA FrameWork by discussing two codes in detail. The first is a simple two-dimensional diffusion simulation, while the second is a 2D electrostatic gyrokinetic particle simulation.

4.1 Simple 2D Diffusion

The diffusion code in Program 1.1 demonstrates some of the capabilities of the `Field`, `CEO`, and `Timer` classes provided in the POOMA FrameWork. The command-line input is assumed to contain the system size and the number of iterations to be performed. The problem the code solves is quite simple: starting with a two-dimensional field of double-precision values, deposit a non-zero value near the center of this field (simulating an initial density), and then for a number of iterations, have each element update its value with the average of itself and its eight neighbors. This operation (depicted in Figure 1) is referred to as a nine-point stencil.

```
1     // Simple two-dimensional diffusion simulation
2     #include <stream.h>
3     #include <unistd.h>
4     #include "POOMA.h"  // collection of header files for POOMA
5
6     #define MILLION 1000000.0
7
8     int main(int argc, char *argv[])
9     {
10      double mFlops, startVal, endVal, seconds, relativeError;
11      int i, n, iter;
12      PoomaInfo* myinfo = new PoomaInfo(argc,argv);
13      Timer cpuTime;
14
15      sscanf(argv[1], "%d", &n);
16      sscanf(argv[2], "%d", &iter);
17
18
19      int centern = n/2;
20      long ops = 9 * n * n * iter;
21      Index I(n), J(n);
22      Field a(I,J);
23
24      startVal = 1000.0;
25      cpuTime.clear();
26      cpuTime.start();                          // start timer
27      a.atom_set(startVal, centern, centern);  // put value in center
28    // loop on stencil operation
```

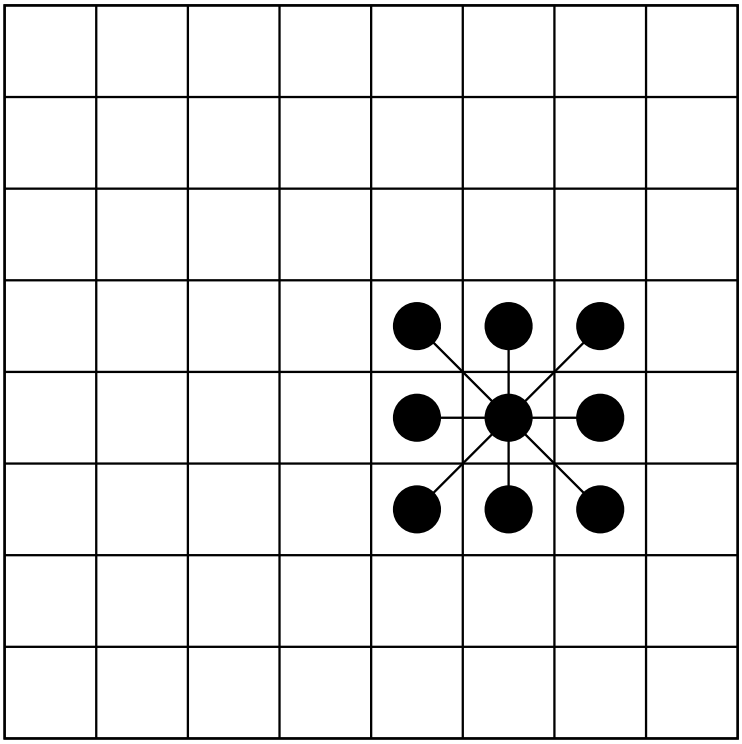


Figure 1: Two-dimensional 9-point diffusion stencil

```
29      for (i=0; i<iter; i++)
30        a = (a[I-1][J-1] + a[I-1][J] + a[I-1][J+1] +
31             a[I][J-1] + a[I][J] + a[I][J+1] +
32             a[I+1][J-1] + a[I+1][J] + a[I+1][J+1]) / 9.0;
33
34      cpuTime.stop();                          // stop timer
35      seconds = cpuTime.cpu_time();            // get time
36      endVal = a.sum();                        // get sum of elements
37    // compare final sum to original
38      relativeError = ABS(startVal - endVal) / ABS(startVal);
39
40      if ((myinfo->get_comm()->this_node())==0)
41      {
42        mFlops = ops / (seconds * MILLION);
43        cout << "Total floating point operations = " << ops << endl;
44        cout << "Total cpu time to solution = " << seconds << endl;
45        cout << "Performance in mFlops = " << mFlops << endl;
46        cout << "Relative Error = " << relativeError<<endl;
```

```
47        }
48        return 1;
49      }
```

Program 1.1: 2D diffusion code written with POOMA

Lines 1–11 are straightforward inclusions and declarations. Line 12 constructs a `PoomaInfo` object, which in turn constructs a `Communicate` object and a `VnodeManager` object. The `Communicate` object is responsible for interprocessor communication, whereas the `VnodeManager` is responsible for determining upon which physical node each virtual node resides. The `PoomaInfo` object has pointers to these two important objects, giving the user the capability to obtain information about the virtual node setup within the application code.

Line 13 constructs a `Timer` object to measure CPU time used by the code. This `Timer` object is an extension of a utility developed at the University of Illinois to perform on a variety of serial, distributed, and parallel architectures. Lines 15–16 initialize the size of the field and the number of iterations using command-line arguments. Line 19 approximates the center of the field and assigns this value to `centern`, while Line 20 calculates the number of operations that will be performed by the diffusion code.

Line 21 constructs two `Index` objects of length `n`. The `Index` objects control the pattern by which data is accessed within a `Field` object. Length, offset, and stride are all configurable. Line 22 constructs a two-dimensional `Field` of doubles. The default layout assumes a domain decomposition that minimizes the surface-to-area ratio of the rectilinear sub-domains on each virtual node, although other layouts can be specified. Since this is the first `Field` object instantiated in this simulation, the `Field` also registers itself and all its expression kernels with a `CEO`.

Line 24 assigns the initial value to be deposited near the center of the field. Lines 25–26 clear and start the `Timer` object. Line 27 assigns the value `startVal` to the element in `Field` a at coordinates `(centern, centern)`. The main loop begins at line 29; lines 30–32 perform the 9-point diffusion stencil. Once the loop is completed, lines 34–35 stop the timer and get the amount of CPU time spent executing the loop.

Line 36 uses the `Field::sum()` member function to do a global sum on the elements of the field to verify the conservation of mass. Line 38 calculates the relative error between the starting and ending values. Line 40 uses the `PoomaInfo` object (via `Communicate`) to obtain the physical node number. Node 0 then prints out the results in lines 42–46. Finally, in lines 47–49, destructors for the `Field`, `Timer`, `Index`, and `PoomaInfo` objects are called, since they are now out of scope. This also causes the call of destructors for `Communicate`, `CEO`, and `VnodeManager`.

A brief explanation of how `CEO` manipulates `Field` objects is in order at this point. The `Field` class is actually a pair of letter/envelope [Coplien 1992]

classes. The letter class contains all of the field data, while the envelope class contains lightweight objects that maintain a pointer to the letter object and the indices and offsets gathered by the `Field::operator[]` during the traversal of a `Field` in an expression. Thus, different offsets can apply to the same object. Each `Field` on the right-hand side of an expression is accessed with a set of `Index` objects. This instantiates an envelope object that places the ID for the contained letter object onto the expression stack (represented by the token "FLD") in the `CEO`, with the appropriate offset information garnered from the `Index` objects. Constants are placed on the stack as tokens labeled "CON". Operators such as + are overloaded in a non-standard way. Instead of performing the addition of two terms, the `Field::operator+(Field&)` function describes the requested operation to the `CEO`. In the case of `a(I-1,J-1) + a(I-1,J)`, the operator token "add" is placed on the expression stack in the `CEO`; no additions are performed. This process of placing tokens onto the `CEO` expression stack continues until the assignment operator is reached (`Field::operator=(Field&)`). At this point, the `CEO` is instructed to evaluate the entire expression stack, which by this time has become

FLDFLDaddFLDaddFLDaddFLDaddFLDaddFLDaddFLDaddFLDaddCONdiv

For the diffusion example, a kernel performing a 9-point stencil operation is registered with the `CEO`. Thus, the `CEO` makes a single function call that performs the entire 9-point stencil. Before the kernel is called, however, the `CEO` checks to see if any of the `Field` references are "off-node". `Field` objects are constructed either with or without "guard" cells (sometimes also called "boundary" or "ghost" cells). If a `Field` has guard cells and the expression references an off-node element of that `Field`, then the `CEO` updates these guard cells by using the `Communicate` object (see Figure 2).

If the `Field` was constructed without guard cells, then a temporary `Field` is constructed with the appropriate number of guard cells, and those guard cells are updated using `Communicate`. The `CEO` maintains a list of the temporaries created during an expression and deletes them at the conclusion of the `Field::operator=` call. Thus, if one knows that such off-node references will occur often in an application, one should construct `Field` objects with the necessary borders at the outset.

Table 2 and Table 3 present breakdowns of time spent in two slightly different versions of the diffusion code. In the first case, the result of the stencil operation is put into a second `Field` object called `b` and then explicitly copied back to `a` on a separate line of code. The second case is that shown in Program 1.1, where the `CEO` handles the data dependency as explained in Section 3.4. All cases were run on a single processor.

As can be seen in both tables, there is very little overhead incurred by employing the `CEO`. Most of the time is spent in the inner `for` loops of the 9-point

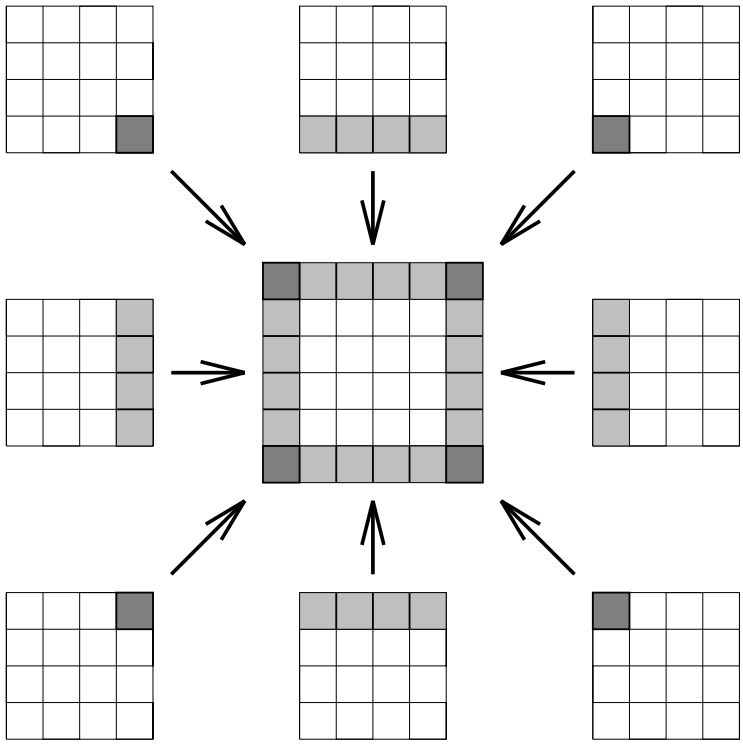


Figure 2: Communication pattern for border update during two-dimensional, 9-point diffusion stencil

stencil kernel operator. Furthermore, allowing the CEO to handle data dependency, rather than performing an explicit copy, avoids an expensive memcpy call and makes the computation even more efficient.

Registered kernels can be highly tuned assembly, Fortran, C or C++ code that performs a specific task. When the kernel is called, it is passed all the information required to describe the objects and operations involved in the expression. This includes getting pointers to the actual field data, and getting offset and axis length information. Once this information is obtained, the expression is evaluated as a whole. Program 1.2 shows an example of such a kernel.

```
double *out, *f[8];
out = A.data_address();
for (int i=0; i<8; i++)
  f[i] = F[i].data_address();
for (int y=0; y<ay; y++)
  for (int x=0; x<ax; x++)
```

```
*out++ = (*f[0]++ + *f[1]++ + *f[2]++ +
          *f[3]++ + *f[4]++ + *f[5]++ +
          *f[6]++ + *f[7]++ + *f[8]++) / constant;
```

Program 1.2: Kernel for nine-point stencil operation

Routines	SGI Crimson	Sun HyperSPARC	IBM RS6000
stencil	74.0	72.2	83.5
memcpy	23.7	25.6	13.7
allocate memory	0.9	0.7	1.2
get_data	0.4	0.3	0.2
set_data	0.4	0.3	0.6
sum	0.2	0.2	0.3
other	0.4	0.7	0.5

Table 2: Breakdown of the percentage of time spent in each routine on various architectures (with explicit copying)

Routines	SGI Crimson	Sun HyperSPARC	IBM RS6000
stencil	97.1	97.2	96.4
allocate memory	1.1	0.9	1.4
get_data	0.6	0.4	0.3
set_data	0.4	0.3	0.5
sum	0.3	0.3	0.2
other	0.5	0.9	0.3

Table 3: Breakdown of the percentage of time spent in each routine on various architectures (without explicit copying)

Performance results for the 2D diffusion code running on the Cray T3D and the IBM SP2 are presented in Table 4 and Table 5.

4.2 Gyrokinetic Simulation

Our gyrokinetic (GK) code performs a simple 2D, electrostatic, gyrokinetic particle simulation of a fusion plasma. It is comprised of a single source file and header files that declare the constants and subroutines used in the main code. The program reads an input file to obtain basic parameters describing the desired simulation, such as the number of particles, number of grid cells along

Nodes	Problem Size	MFlops	MFlops/Node
2	1024x1024	26.71	13.36
4	1024x1024	48.87	12.22
8	1024x1024	101.44	12.68
16	1024x1024	178.83	11.18
32	2048x2048	396.58	12.39
64	2048x2048	633.03	9.89
64	4096x4096	758.14	11.85

Table 4: Performance of 2D diffusion code on Cray T3D (with readahead on)

Nodes	Problem Size	MFlops	MFlops/Node
1	1024x1024	27.35	27.35
2	1024x1024	47.24	23.62
4	1024x1024	83.97	20.99
8	1024x1024	136.65	17.08
16	1024x1024	176.28	11.02
16	2048x2048	209.10	13.07
16	4096x4096	262.82	16.43

Table 5: Performance of 2D diffusion code on IBM SP2

each axis of the simulation domain, number and size of timesteps, mass and charge of the plasma constituents, and the scale lengths of the equilibrium density and temperature profiles of the plasma. This code can be used to examine low-frequency, long-wavelength plasma instabilities, such as electron drift waves and ion-temperature-gradient (ITG) modes [Connor 1986], in a simplified geometry. To begin this particle simulation, a `Particles` object is set up for each different particle species in the plasma, as shown in Program 1.3.

```
//  construct particle coordinate and data field objects
//  Max = maximum # of particles per virtual node
DPField xe1(Max), xe2(Max);    // old, new x position
DPField ye1(Max), ye2(Max);    // old, new y position
DPField ve1(Max), ve2(Max);    // old, new v'parallel
DPField we1(Max), we2(Max);    // old, new particle weights
DPField pex1(Max), pex2(Max); // old, new particle E'x
DPField pey1(Max), pey2(Max); // old, new particle E'y
DPField w(Max);                // normalized weight
```

Program 1.3: Code for construction of particle attributes

A `DPField` is constructed for each particle coordinate and for the other desired particle attributes, which in this case include a velocity, electric field values at the particle's position, and a "weight" to represent this particle's contribution to the deviation from plasma equilibrium. The weight is needed because a "δf" scheme [Parker & Lee 1992] is being employed in this code for improved signal-to-noise properties. POOMA offers the user various easy-to-use tools for initializing `DPField` data, including setting all `DPField` values to a constant, inserting random values, using pseudo-random, bit-reversed numbers [Halton 1960], and fitting a probability distribution (e.g., a Maxwellian distribution). Other initialization techniques can be added to the `DPField` class easily. Once the `Particles` object is instantiated, the DPFields are hooked into the `Particles` object (see Program 1.4) and initialized. After the `Particles` object is initialized and `Field` objects have been constructed to hold the charge density, the electrostatic potential, the electric field components, etc., we can begin the timestep loop.

```
// construct electron distribution object
// ncx and ncy are the input system box-size
int TimeLevels = 2;    // predictor-corrector needs two copies of position
int nDPFields = 9;      // number of dbl prec.fields
                                        // attached to each particle
//construct Particles object
Particles electrons(TimeLevels, ncx, ncy, nDPFields);
//Now attach the DPField objects to the Particles object
electrons.set_coord(xe1, 0, 0).set_coord(xe2, 0, 1);
electrons.set_coord(ye1, 1, 0).set_coord(ye2, 1, 1);
electrons.set_dpfield(ve1, 0).set_dpfield(ve2, 1);
electrons.set_dpfield(we1, 2).set_dpfield(we2, 3);
electrons.set_dpfield(pex1, 4).set_dpfield(pex2, 5);
electrons.set_dpfield(pey1, 6).set_dpfield(pey2, 7);
electrons.set_dpfield(w, 8);
```

Program 1.4: Code for attachment of particle attributes to Particles object

We use a two-step, predictor–corrector scheme to advance in time. In each timestep, we start with "old" (from the last timestep) and "current" (from this timestep) values for the particle positions and velocities. In the predictor step, we use the current velocities and electric field to advance the old positions and velocities to the "new" time level. Then, in the corrector step, we average the current and new values of the velocities and electric field, and use these averages to advance the current positions and velocities to the new time level. Both steps are performed in a time-centered manner.

Each step requires us to "scatter" the particles' charge density to the simulation grid, solve Poisson's equation to get the electrostatic potential, take finite

differences to get the components of the electric field, "gather" this electric field to the particles' positions, and advance the particle positions, velocities, and weights. An example of these portions of the code is shown in Program 1.5.

```
// accumulate electron charge density
// onto grid using positions (xe2,ye2)
gphi2 = 0.0; // clear grid
// scatter weights w onto gphi2
electrons.ScatterNGP(gphi2, w, xe2, ye2);
//  FFT-based field solver
// Fourier transform charge density
fftob.FFT_CC(gphi, gphi_im, 1, 1, 1.0/sqrt(ancx));
fftob.FFT_CC(gphi, gphi_im, 2, 1, 1.0/sqrt(ancy));
// apply form factors
// CEO recognizes these operations and optimizes
gphi    = gphi*fmpo;
gphi_im = gphi_im*fmpo;
gphi    = gphi*fmax;
gphi_im = gphi_im*fmax;
// inverse transform on potential with scale factor for normalization
fftob.FFT_CC(gphi, gphi_im, 2, -1, 1.0/sqrt(ancy));
fftob.FFT_CC(gphi, gphi_im, 1, -1, 1.0/sqrt(ancx));
// compute finite-difference Ex and Ey electric field components
// CEO recognizes these operations, updates border
// information and optimizes
ex2 = 0.5 * rhos * (gphi2[I+1,J] - gphi2[I-1,J]);
ey2 = 0.5 * rhos * (gphi2[I,J+1] - gphi2[I,J-1]);
// gather electric fields onto particle positions (xe2,ye2)
electrons.GatherNGP(ex2, pex2, xe2, ye2);
electrons.GatherNGP(ey2, pey2, xe2, ye2);
// advance particle positions, velocities, and weights using velocity
// and E field.  CEO recognizes these operations and optimizes.
xe1 = xe1 - 2.0 * dt * rhos * pey2;
ye1 = ye1 + 2.0 * dt * rhos * (theta*ve2 + pex2);
we1 = we1 - 2.0 * dt * rhos * swh * theta * pey2 * ve2 / (vtxe*vtxe);
ve1 = ve1 - 2.0 * dt * rhos * swh * theta * pey2 / vtxe;
// swap particles to be on same processor as local grid data
// swap using predicted coordinates
electrons.swap(xe1, ye1);
```

Program 1.5: Timestep of gyrokinetic particle code

The `GatherNGP` and `ScatterNGP` procedures are gather and scatter functions that utilize a nearest-grid-point (NGP) `Interpolate` object to perform the interpolation between particle positions and grid points. Border-cell information is automatically updated on each `LocalField` object, so that particles have access to the grid cells they need. The GK code has a 2-dimensional domain that

is periodic in both directions, so the Poisson equation solve is handled using Fast Fourier Transforms and the application of form factors in Fourier space. The finite differencing of the electrostatic potential to obtain the electric field is done by applying an optimized stencil operation to the `Field` containing the potential. Overloaded operators in the `DPField` objects allow the advance of particle positions, velocities, and weights to proceed with data-parallel array syntax. The `Particles` member function `swap` is passed a set of particle coordinates to use in checking which particles have moved out of the local subdomain and need to be passed to a neighboring processor.

The GK code includes a simple diagnostic routine to compute the total kinetic energy of the particles and the field energy of the potential, in order to check energy conservation and monitor instabilities. In addition, GK has been equipped with calls to the Generic Display Library (GDL), a portable graphics package written at the Advanced Computing Laboratory. GDL provides simple function calls that take a 2-dimensional array of data and convert it into a pixel map, which graphically represents a color contour of the data set. The color contour plot is displayed on the terminal screen in real time during the simulation, and it can be updated at any time. This allows one to monitor the electrostatic potential, for example, as a plasma instability develops and is nonlinearly saturated. It is a powerful tool for both debugging and analysis of production runs. Program 1.6 illustrates just how easy it is to utilize GDL within the POOMA FrameWork.

```
// construct serial version of Field containing
// electrostatic field potential for graphical display
// all data resides in contiguous memory on Node 0
SField iophi(gphi2);
// initialize X display for GDL.
// Node 0 controls graphics display.
if (myNode==0) {
  int dummy = GDL_OpenDisplay(X11FB);  // open display in X11 mode
  GDL_SetColormap(RAINBOW_BLUE, NULL); // choose standard color map
}
⋮
// later... during time-stepping loop
// output values of electrostatic potential
iophi.update();  // update values in Serial Field from parallel Field
// X display of SField using GDL
// display data in SField
if (myNode==0) GDL_Display_double(iophi.get_data(),
               iophi.get_length(0), iophi.get_length(1));
```

Program 1.6: Utilizing GDL in the gyrokinetic code

The code shown in Program 1.6 is designed for debugging moderate-sized,

two-dimensional simulations. Current research is focused on embedding GDL into a Graphical User Interface with Tcl and the Tk Toolkit [Ousterhout 1994] and merging objects in the POOMA FrameWork with parallel rendering techniques.

5 The Polygon Overlay Problem

The POOMA implementation of the polygon overlay problem is almost identical to the ANSI C base code, for a couple of reasons. First, one of the central themes in the design of the POOMA FrameWork is to ease migration from known programming environments (e.g., ANSI C) to the new programming environment of the FrameWork. In addition, it is more meaningful to compare performance numbers between the Framework implementation and the ANSI implementation if the methods used are the same.

Two different versions of each of the sequential methods of polygon overlay were implemented and benchmarked on a Cray YMP, a Cray T3D, a PVM cluster of RS6K workstations, an SGI Crimson, an IBM SP2, and the Meiko CS2.

5.1 POOMA Framework Implementation Details

In the first implementation of the polygon overlay code (see Program 5.1), almost no change to the ANSI source code was required. The I/O routines were modified so that the entire left vector was read into each processor, while the right vector was divided between the processors.

```
#include "defs.h"
#include "Index.h"
#include "IField.h"
#include "PoomaInfo.h"

static int Nodes, MyNodes;

int
main(
  int argc, /* argument count */
  char_p argv[] /* argument vector */
){
  ···definitions from sequential code···
  PoomaInfo MyInfo(argc, argv);

  Nodes  = MyInfo.get_comm()->nodes();
  MyNode = MyInfo.get_comm()->this_node();

  ···handle arguments···
  ···read inputs···
  ···perform overlay···

  // Gather results from pnodes to pnode 0
  Index I(Nodes);
  IField Polys(I);
```

```
  int    OutSecSize, TotalPolys, i, nodes;
  poly_p pp;

  Polys.local_set(outVec->len, 0);
  OutSecSize = Polys.max();
  TotalPolys = Polys.sum();

  Index OP(OutSecSize * 4 * Nodes);
  IField OutVec(OP);

  pp = outVec->vec;
  for (i = 0 ; i < outVec->len * 4 ; i += 4){
     OutVec.local_set(pp->xl, i    );
     OutVec.local_set(pp->yl, i + 1);
     OutVec.local_set(pp->xh, i + 2);
     OutVec.local_set(pp->yh, i + 3);
     pp++;
  }

  free(leftVec->vec);
  free(rightVec->vec);
  free(outVec->vec);
  ALLOC(outVec->vec, Polys.sum(), poly_t);

  pp = outVec->vec;
  for (nodes = 0 ; nodes < Nodes ; nodes++){
    for (i = 0 ; i < Polys.atom_get(nodes) * 4 ; i += 4){
       pp->xl = OutVec.atom_get(nodes * OutSecSize + i    );
       pp->yl = OutVec.atom_get(nodes * OutSecSize + i + 1);
       pp->xh = OutVec.atom_get(nodes * OutSecSize + i + 2);
       pp->yh = OutVec.atom_get(nodes * OutSecSize + i + 3);
       pp++;
    }
  }
  outVec->len = TotalPolys;

  // Only Pnode 0 will do the sorting and output
  if (!MyNode){
    ···sort using qsort···
    ···output···
  }

  // finish
  return 0;
}
```

Program 1.7: POOMA polygon overlay code

Two static, global, integer variables were added to hold the number of processors and the current physical node (pnode). A `PoomaInfo` object was instantiated in the main declaration area, since data concerning the number of physical nodes present was required before the instantiation of a Global Data Type object. The `PoomaInfo` object checks to see if there is currently a `Communicate` object running. If not, it will create one. The `Communicate` object's tasks are to obtain the number of physical nodes and to start running a copy of the executable on each physical node that is discovered. Next, the `PoomaInfo` object checks to see if there is a `VnodeManager` running. Once again, if not, it will create one. `VnodeManager` is responsible for keeping track of the virtual nodes on each physical node. The default behavior is to provide only one virtual node per physical node, but a larger number may be requested. Once the `PoomaInfo` object is constructed, the variables `Nodes` and `MyNode` can be initialized.

Execution continues with the command-line processing being handled as usual. The I/O source code (not shown) reads the first polygon file in serially. The second file is read by dividing the size of the file by the number of physical nodes present in the current machine. Each copy of the executable running then reads in only that portion of the second polygon file that it is responsible for.

When the input files have been read, each processor generates the polygon list for its input sets. Although an integer `Field` is employed during the simulation, the embarrassingly parallel nature of the polygon overlay problem does not exploit any of the communication-hiding features of the `Field` class (like indexing operations). However, it does take advantage of the data-parallel `max` and `sum` operations.

When the output routines are called, the reverse of the input procedure takes place. The total number of polygons created is discovered by each processor setting its `Field` variable to the number of polygons generated for its subset. The `Field` is summed to get the overall total. When the output routine is called, a `Field` variable large enough to hold the entire polygon list is constructed. Each processor fills its portion of this `Field` with generated polygons, and then pnode 0 writes out the entire solution set.

As seen in Table 6 and Table 7, the naive implemention of the code performs much as expected, since once the data is read in, there is no interprocessor communication. (Note: the RS6K optimizations included -qarch=pwr2 -qtune=pwr2 flags and the Crimson optimizations included the -mips2 flag.) The speedup is nearly linear as the number of processors increases. The surprising results appear in the parallel versions of the more sophisticated methods, as seen in Table 8 and Table 9. The List-Ordered and List versions run more slowly when more processors are added. This is due to the way these methods are implemented. The entire left vector is loaded onto each processor, but the right vector is divided between all available physical processors. When a polygon is culled from the list of active polygons, it is culled from the right vector. This implies that some processors may end up culling their entire right vector while others do not cull any polygons at all. The additional checks involved in

Arch	Opt	Nodes	Naive	Ordered	Area-Ord
Cray YMP	-O3	1	0.722	0.521	0.331
Cray T3D	-O2	1	0.500	0.250	0.270
Cray T3D	-O2	2	0.380	0.210	0.220
Cray T3D	-O2	4	0.260	0.200	0.210
SGI Crimson	-O2	1	0.390	0.260	0.110
SGI Crimson	-O2	2	0.210	0.270	0.070
RS6K	-O3	1	0.270	0.200	0.110
RS6K	-O3	2	0.130	0.150	0.080
RS6K	-O3	4	0.090	0.100	0.090
RS6K	-O3	8	0.170	0.210	0.080

Table 6: Polygon overlay code timings (in seconds) on map.00 and map.01 data sets for the Naive, Ordered, and Area-Ordered methods

the List-Ordered and List methods are executed for all elements in both the left vector and right sub-vectors, causing the performance to decrease as seen.

5.2 Improved Polygon Overlay Implementation

The polygon overlay problem appears to be embarassingly parallel in nature, but due to uneven loading caused by the way the data files are produced, adding processors can reduce performance in many cases. A second, improved implementation (given in Program 5.2) remedies this imbalance by dividing the left and right vectors between all the processors. Local versions of the left sub-vectors and right sub-vectors are compared, and a partial output list is created. The code iterates by shifting the right sub-vector to the neighboring processor and repeating the comparison. This continues until each right sub-vector has visited each physical node. By doing this, a crude form of load balancing is achieved, as is evident in the performance numbers given in Table 10 and Table 11.

```
#include "defs.h"
#include "Index.h"
#include "IField.h"
#include "PoomaInfo.h"

static int Nodes, MyNodes;

int
main(
  int argc, /* argument count */
  char_p argv[] /* argument vector */
```

Arch	Opt	Nodes	Area	List-Ord	List
Cray YMP	-O3	1	0.241	0.082	0.081
Cray T3D	-O2	1	0.110	0.050	0.050
Cray T3D	-O2	2	0.190	0.180	0.180
Cray T3D	-O2	4	0.190	0.200	0.210
SGI Crimson	-O2	1	0.070	0.030	0.020
SGI Crimson	-O2	2	0.150	0.030	0.170
RS6K	-O3	1	0.090	0.020	0.030
RS6K	-O3	2	0.200	0.050	0.190
RS6K	-O3	4	0.140	0.070	0.040
RS6K	-O3	8	0.120	0.070	0.120

Table 7: Polygon overlay code timings (in seconds) on map.00 and map.01 data sets for the Area, List-Ordered, and List methods

Arch	Opt	Nodes	Naive	Ordered	Area-Ord
Cray YMP	-O3	1	2287.267	1590.815	959.756
Cray T3D	-O2	1	1402.280	822.140	783.520
Cray T3D	-O2	2	787.070	607.520	600.210
RS6K	-O3	1	844.28	659.37	335.36
RS6K	-O3	2	429.50	491.86	249.36
RS6K	-O3	4	276.63	279.51	141.43
RS6K	-O3	8	173.28	162.86	144.01

Table 8: Polygon overlay code timings (in seconds) on K100.00 and K100.01 data sets for the Naive, Ordered, and Area-Ordered methods

```
){
  ···definitions from sequential code···
  PoomaInfo MyInfo(argc, argv);

  Nodes  = MyInfo.get_comm()->nodes();
  MyNode = MyInfo.get_comm()->this_node();

  ···handle arguments···

  // Load Left Vector into shift buffer
  Index I(Nodes);
  IField Polys(I);
  int PolysPerNode, OutSecSize, TotalPolys, i, eachnode, node, GlobalTotal=0;
```

Arch	Opt	Nodes	Area	List-Ord	List
Cray YMP	-O3	1	662.867	34.652	34.669
Cray T3D	-O2	1	288.580	18.080	16.510
Cray T3D	-O2	2	345.190	21.100	30.250
RS6K	-O3	1	226.58	9.11	8.09
RS6K	-O3	2	225.30	25.17	61.43
RS6K	-O3	4	234.60	42.90	111.59
RS6K	-O3	8	205.23	4.22	70.99

Table 9: Polygon overlay code timings (in seconds) on K100.00 and K100.01 data sets for the Area, List-Ordered, and List methods

```
Polys.local_set(leftVec->len, 0);
PolysPerNode = Polys.atom_get(0);
TotalPolys   = PolysPerNode * Nodes;
Index SHIFTI(TotalPolys * 4);
IField ShiftBuf(SHIFTI);
poly_p pp;

pp = leftVec->vec;
for (i = 0 ; i < leftVec->len * 4 ; i+=4){
  ShiftBuf.local_set(pp->xl, i    );
  ShiftBuf.local_set(pp->yl, i + 1);
  ShiftBuf.local_set(pp->xh, i + 2);
  ShiftBuf.local_set(pp->yh, i + 3);
  pp++;
}

for (int eachnode = 0 ; eachnode < Nodes ; eachnode++){

   ...perform overlay...

   ShiftBuf = ShiftBuf(SHIFTI + (PolysPerNode * 4));

//reload the left vector with the contents in ShiftBuf

   pp = leftVec->vec;
   for (i = 0 ; i < leftVec->len * 4 ; i++)
   {
      pp->xl = ShiftBuf.local_get(i    );
      pp->yl = ShiftBuf.local_get(i + 1);
      pp->xh = ShiftBuf.local_get(i + 2);
      pp->yh = ShiftBuf.local_get(i + 3);
```

```
        pp++;
      }
    }
  //Gather results from pnodes to pnode 0
    Polys.local_set(outVec->len, 0);
    OutSecSize = Polys.max();
    TotalPolys = Polys.sum();
    GlobalTotal += TotalPolys;

    Index OP(OutSecSize * 4 * Nodes);
    IField OutVec(OP);

    pp = outVec->vec;
    for (i = 0 ; i < outVec->len * 4 ; i += 4){
      OutVec.local_set(pp->xl, i    );
      OutVec.local_set(pp->yl, i + 1);
      OutVec.local_set(pp->xh, i + 2);
      OutVec.local_set(pp->yh, i + 3);
      pp++;
    }

    free(outVec->vec);
    ALLOC(outVec->vec, Polys.sum(), poly_t);

    pp = outVec->vec;
    for (node = 0 ; node < Nodes ; node++){
      for (i = 0 ; i < Polys.atom_get(node) * 4 ; i += 4){
        pp->xl = OutVec.atom_get(node * OutSecSize + i    );
        pp->yl = OutVec.atom_get(node * OutSecSize + i + 1);
        pp->xh = OutVec.atom_get(node * OutSecSize + i + 2);
        pp->yh = OutVec.atom_get(node * OutSecSize + i + 3);
        pp++;
      }

      outVec->len = TotalPolys;

      // Only Pnode 0 will do the sorting and output
      if (!MyNode){
        ···sort using qsort···
        ···output···
      }
    }

    // finish
    return 0;
  }
```

Program 1.8: Improved POOMA polygon overlay code

Arch	Opt	Nodes	Naive	Ordered	Area-Ord
Cray T3D	-O2	8	187.070	28.820	28.970
Cray T3D	-O2	16	102.790	21.012	20.150
Cray T3D	-O2	32	59.920	18.020	17.960
RS6K	-O3	1	844.28	659.37	335.36
RS6K	-O3	2	415.01	171.23	90.58
RS6K	-O3	4	212.83	58.65	37.78
RS6K	-O3	8	113.50	27.02	21.60

Table 10: Improved polygon overlay code timings (in seconds) on K100.00 and K100.01 data sets for the Naive, Ordered, and Area-Ordered methods

Arch	Opt	Nodes	Area	List-Ord	List
Cray T3D	-O2	8	23.900	18.309	17.940
Cray T3D	-O2	16	18.570	17.270	13.810
Cray T3D	-O2	32	17.860	17.220	14.130
RS6K	-O3	1	226.58	9.11	8.09
RS6K	-O3	2	62.67	10.67	9.64
RS6K	-O3	4	31.20	18.65	18.73
RS6K	-O3	8	19.21	17.02	17.24

Table 11: Improved polygon overlay code timings (in seconds) on K100.00 and K100.01 data sets for the Area, List-Ordered, and List methods

In Table 10 and Table 11, an increase in code scalabilty is evident. When higher numbers of processors are used, the performance begins to level off due to the higher amount of data movement on the network. The added advantage of this second implementation is its ability to handle much larger data sets, since it is not necessary to load the entire left sub-vector onto each physical processor.

A further enhancement to this second scheme would be to retain the shortened right sub-vectors between iterations. As it stands, each new iteration considers all the right sub-vector polygons active, regardless of whether or not they were removed from the linked list in one of the previous iterations.

6 Critique

Although there are many advantages to our approach to parallelism as described above, there are some limitations. Even though codes can move with no changes between parallel and serial environments, the motivation for this was to leverage serial environments for development against parallel environments for production. The POOMA FrameWork is tuned for solving large problems on parallel supercomputers efficiently. The techniques utilized to provide these performance gains on parallel architectures provide minimal gains on serial architectures.

The target users for the POOMA FrameWork are science and engineering application developers who, as a community, prefer programming in Fortran and other procedural languages. Although C++ is a powerful object-oriented language with which the FrameWork mimics a procedural language, encapsulates parallelism, and chains expression for efficiency, there is still a perceivable lag in C++ compiler technology. Thus, it will take some time before the scientific and engineering communities accept C++ as being as efficient a language as Fortran, and this will inhibit the transition to systems like POOMA.

In addition, because we have chosen to drive our system with scientific applications, the scope of POOMA is somewhat limited. This is not a general-purpose programming system; rather, it is specifically focused on facilitating the use of numerical tools common to scientific application codes on parallel computer architectures. This design decision will undoubtedly exclude other types of computer codes from utilizing tools in the POOMA FrameWork.

Finally, the FrameWork currently focuses on a SPMD approach to parallelism. Although a data-parallel approach captures a majority of scientific application domains and provides an intuitive casting of mathematical expressions directly into objects, there are those who argue that some algorithms are better cast in a task-parallel language. In the future, we hope to bridge this gap by combining objects from the POOMA FrameWork with task-parallel, run-time systems such as Chapter **??**.

References

[Beazley *et al.* 1994] D. M. Beazley, P. S. Lomdahl, N. Grønbech-Jensen, and P. Tamayo. High performance communication and memory caching scheme for molecular dynamics on the cm-5. In *Proc. of 8th International Parallel Processing Symposium*, 1994.

[Birdsall & Langdon 1985] C. K. Birdsall and A. B. Langdon. *Plasma Physics Via Computer Simulation.* McGraw-Hill, New York, 1985.

[Connor 1986] J. Connor. Transport Due to Ion Pressure Gradient Turbulence. *Nuclear Fusion*, 26:193, 1986.

[Coplien 1992] J. O. Coplien. *Advanced C++: Programming Styles and Idioms.* Addison-Wesley, 1992.

[Decyk 1995] V. Decyk. Skeleton PIC codes for parallel computers. *Computer Physics Communications*, 87:87–94, 1995.

[Forslund *et al.* 1990] D. Forslund, C. Wingate, P. Ford, J. Junkins, J. Jackson, and S. Pope. Experiences in writing a distributed particle simulation code in c++. In *1990 USENIX C++ Conference Proceedings*, 1990.

[Gamma *et al.* 1995] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[Gupta *et al.* 1995] A. Gupta, V. Kumar, and A. Sameh. Performance and Scalability of Preconditioned Conjugate Gradient Methods on Parallel Computers. *To appear in IEEE Transactions on Parallel and Distributed Systems*, 1995.

[Halton 1960] J. Halton. On the efficienty of certain quasi-random sequences of points in evaluating multi-dimensional integrals. *Numerische Mathematik*, 2:84–90, 1960.

[Hillis & Steele 1986] W. Daniel Hillis and Guy L. Steele, Jr. Data Parallel Algorithms. *Communications of the ACM*, 29(12):1170–1183, December 1986.

[Kumar *et al.* 1993] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms.* Benjamin-Cummings, 1994.

[Lee 1987] W. W. Lee. Gyrokinetic Particle Simulation Model. *Journal of Computational Physics*, 72:243, 1987.

[Lemke & Quinlan 1992] M. Lemke and D. Quinlan. P++, a C++ Virtual Shared Grids Based Programming Environment for Architecture-Independent Development of Structured Grid Applications. In *In Arbeitspapiere der GMD, No. 611*, 1992.

[OMG 1993] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, Version 1.2 edition, December 1993.

[Ousterhout 1994] John Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[Parker & Lee 1992] S. E. Parker and W. W. Lee. A Fully Nonlinear Characteristic Method for Gyokinetic Simulation. *Physics of Fluids B*, 5:77–86, 1992.

[Parson & Quinlan 1994] 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.

[Reynders *et al.* 1994] J. V. W. Reynders, D. W. Forslund, P. J. Hinker, M. Tholburn, D. G. Kilman, and W. F. Humphrey. Object-Oriented Particle Simulation on Parallel Computers. In *OON-SKI '94 Proceedings ??? ???*, 1994.

[Reynders *et al.* 1995] J. V. W. Reynders, D. W. Forslund, P. J. Hinker, M. Tholburn, D. G. Kilman, and W. F. Humphrey. OOPS: An Object-Oriented Particle Simulation Class Library for Distributed Architectures. *Computational Physics Communications*, to appear.