Experiences in Writing a Distributed Particle Simulation Code in C++

David W. Forslund Charles Wingate
Peter Ford J. Stephen Junkins
Jeffrey Jackson Stephen C. Pope

CRPC-TR90060
July, 1990

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

Experiences in Writing a Distributed Particle Simulation Code in C++

David W. Forslund, Charles Wingate, Peter Ford, J. Stephen Junkins, Jeffrey Jackson, Stephen C. Pope

Los Alamos National Laboratory

Los Alamos, NM 87545

dwf@lanl.gov

Abstract

Although C++ has been successfully used in a variety of computer science applications, it has not yet become of widespread use in scientific applications. We believe that the object-oriented properties of C++ lend themselves well to scientific computations by making maintenance of the code easier, by making the code easier to understand, and by providing a better paradigm for distributed memory parallel codes. We describe here our experiences using C++ to write a particle plasma simulation code using object-oriented techniques for use in a distributed computing environment. We initially designed and implemented the code for serial computation and are in the process of making it work on top of the distributed programming toolkit ISIS. In this connection we describe some of the difficulties presented by using C++ for doing parallel and scientific computation. In the spirit of most C++ papers, we advocate some changes in the language, although we remain devotees of C++ despite the shortcomings cited.

Introduction

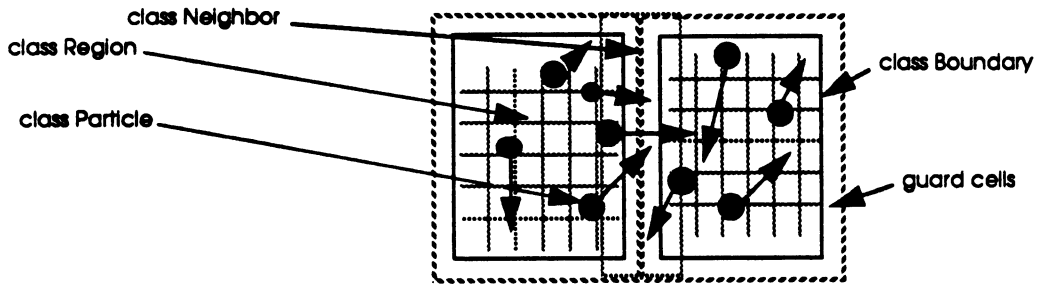
Plasma particle simulation involves the modeling of the behavior of an ionized gas in its self-consistent and externally imposed electromagnetic fields by advancing a large collection of particles with discretized Newton's laws on a grid and the solution of discrete electromagnetic field equations using particle quantities interpolated to the grid. WAVE [Forslund 1985] is a moderately large (20,000+ lines) plasma simulation code written in Fortran which is in wide spread use at plasma research sites around the world. The code can consume enormous amounts of computer time in the process of modeling problems ranging from the interaction of intense laser light with ionized gases to the behavior of the interstellar medium. Although the model is quite simple, it re-

quires vast computer resources and is reasonably well suited to running on a massively parallel MIMD computer. There are many physics problems being studied atLos Alamos National Laboratory which will eventually require harnessing several Cray class computers together over high speed channels. Although the channels will be very high speed they will be high latency relative to the clock cycle of the machines. This is similar to a network of RISC based machines running on a ethernet. The goal of this project is to write WAVE++ in an object-oriented manner to simplify and accelerate its use on large parallel computers and and simultaneously match the numerical model more closely to the physical world. This also provides a good platform for evaluating the usefulness of C++ for scientific computing and observing some ways in which the language might be improved.

Design of WAVE++

C++ enables us to design the code in a more modular fashion than is possible to do with Fortran and allows for much more flexibility in the type of data structures used. The most compelling reason to use C++, however, is the natural decomposition of the problem for parallelization. The data and the methods are kept together providing all the information for advancing the particles and fields in the local grids. The grid domain in WAVE++ is currently decomposed with a quadtree in 2 dimensions and an octtree in 3 dimensions. Each leaf (a minimum size grid region) has all the methods and data (particles and fields) needed to solve the problem locally. Communication between the regions is constrained to a single layer of cells surrounding each region. The electromagnetic field equations involve a coupled set of second order partial differential equations. Although in the Fortran version the field equations are solved by block diagonal linear algebra methods, in order to minimize communication in WAVE++, we solve the field equations by means of a local iterative solve with only a small amount of communication across the boundary layers (Fig 1.).

FIGURE 1. Grid and Particles in the Wave++ Model.



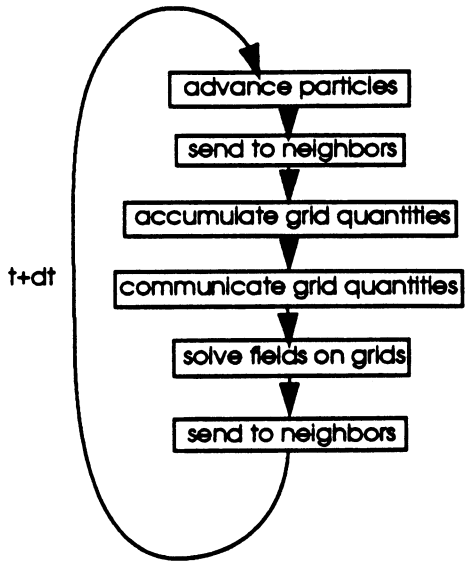
Although inheritance is not heavily used in this application, it is quite useful for several of the container classes used to manipulate the data. The basic classes involved in WAVE++ are illustrated in Table I. The two most important are Particle, which contains the methods for advancing the particles in response to the fields, and Region, which is a restricted domain of the grid which contains the fields and particles.

TABLE 1. Class Structure for Wave++.

Class	Functionality
Array	Holds grid variables, provides guard cell references
Boundary	Encapsulates geometry of a region
List	Linked list class
Neighbor	Region guard cells and communication
Particle	Physics of particles, how to accelerate, etc.
Region	Basic physics (contains multiple arrays of fields and currents, knows neighbors, pushes particles, advances fields, low level diagnostics)
Species	Defines properties of groups of particles
Wave	Master for all regions, initializes and collects data

A Region is intended to reside entirely on a single processor of some parallel computer. As the particles move around on the grid, they pass from one region to another, and the fields communicate across the boundaries of the various regions.The class Neighbor, which represents the overlap between adjoining regions, is designed to provide the basic communication buffering between regions. This is done in a memory efficient manner and in such a way as to minimize the number of communication events between regions. Data is accumulated by these buffering classes and is transmitted in one command once the data has been fully assembled. By defining reference Arrays in Neighbor, the update of data in the internal Arrays of Region automatically updates the Array buffers in Neighbor. The particles crossing the region boundaries are also buffered for a single transmission to the appropriate neighboring region. This buffering facilitates the functioning of the code in a distributed, high-bandwidth, high latency environment. We note that the design decisions which were made for the sake of parallelism actually clarified the overall design and increased the speed of the serial implementation. To better illustrate how the physics is related to the communication process, we show the cyclic process of advancing the fields and particles in Fig. 2.

FIGURE 2. Flow Diagram of Simulation Time Step.



With the proven capability of C++ in handling graphics, we have designed into the code a modern graphical user interface to assist in the setup of problems and in the interpretation of data generated. This is complicated by the desire to run the code in a distributed parallel environment. We have chosen to use the XView toolkit because of our familiarity with Sunview.

The design of the particle and region classes allows for relatively small effort in changing the code from 2D to 3D (a change in the tree structure and in the boundary class) and in enabling the grid to adaptively reshape itself during the computation to track the physics of the problem. This will allow the code to use a variety of popular simulation methods.

Serial Wave++ Implementation

WAVE++ was first built in a serial manner on a Sun workstation, in which the leaf regions are explicitly advanced using the generating quadtree as the computational harness. This allowed for a simple debugging environment and basic optimization issues to be addressed and solved. As with most first time C++ programmers, it took a few stabs into dark alleys to discover that the key to successful programming in C++ requires an object oriented design in the first place.

We use gprof++ as provided by Sun Microsystem's C++ 2.0 environment as the performance profiler. In Table 2 we show a comparison of Fortran (F77) and C++ performance times for several categories of a computation with 15,000 particles running 133 time steps. In the Fortran version "Communication" refers to the functions required to bring blocks of particles into the particle mover. In C++ it is the time spent in the routines which send the data between regions and in the link list functions. Note that the Particles and Fields combine to be more than 70% of the computation time in both cases but that the Fortran is about 1.7 times faster than the C++ code. Some of this is due to the much finer granularity of the C++ classes than the Fortran. In the Fortran, one subroutine call pushes 256 particles. In C++ there are multiple methods invoked for each particle. The difference is even greater in the optimized code (factor of 2.2) because of the greater opportunity for optimization in the larger blocks of code and the small function call overhead. Use of inlining would probably help.We are continuing to look at the class hierarchy in an effort to allow greater optimization in C++ without the sacrifice of the object-oriented character. This issue of being able to optimize the C++ code may continue to be a problem unless the compilers become much smarter. In a vector machine such as the Cray, the Fortran is substantially faster because most of the inner loops are fully vectorized. Vectorization issues in C++ are discussed later in this paper.

TABLE 2. Performance Comparisons between Fortran and C++

	C++ -g	F77 -g	C++ -O	F77 -O
Function	**Time (Seconds)**	**Time (Seconds)**	**Time(Seconds)**	**Time(seconds)**
Particles	374	223	338	153
Fields	18	31	12	17
Communication	50	56	40	15
Diagnostics	29	5	25	4
Other	79	10	70	1

Parallelization Efforts

The granularity of the parallelism can be readily controlled by design, and in general, a number of parallel threads run on a single processor simultaneously. This is primarily done in order to simplify the problem of load balancing between processors by combining together on one node both lightly and heavily loaded threads. At this time we have not attempted to perform dynamic load balancing, although the structure of the code will permit dynamic reconfiguration of the grid based on particle count, which would be a step in this direction.

One of the primary reasons for redesigning and recoding Wave using C++ was to use an object oriented design (OOD) methodology. By using OOD we felt it would be significantly easier to intro-

duce parallelism into the code. The Presto [Bershad et al. 1988] model of computation allows the programmer to explicitly introduce parallelism by creating separate threads of control and then direct each thread to execute an object's method. Using this model allowed us to develop the code in a serial fashion and then add parallelism where we felt it was needed, without disturbing the original code and class hierarchy.

The Presto model of programming in C++ works with a Thread class defined as:

```
class Thread {
private: // internal representation of thread
public:
   Thread();
   call(POBJany,PFany,...);// execute method on object
}
```

The basic idea is that the user can have the thread execute any method of any object:

```
class Region {
public:
       Region ();
       int update(int, float); // ...
}
Thread *t;
t = new Thread; // create a thread, on standby
Region o;
t.call(o, o.update, 2, 5.4);
```

This example illustrates there are many problems with the C++ implementation of the Presto model, most of which are due to abuse of C++. The current implementation of Presto under C++ can only use methods which are not overloaded, return values are ignored and the argument list is not coerced to the types of the formal arguments of method Obj::update. C++ does not have a way of specifying the linkage between the object/method/argument list which would allow proper type checking and coercion. As it stands, Thread::call is type unsafe. Note that Thread::call is in spirit a polymorphic function, which takes an object of any type TY, and a method of TY which has a signature to which the argument list can be coerced. Currently C++ does not have semantics (or for that matter syntax) which allow the programmer to express anything about method invocation (or function calling) beyond actually doing the invocation. A bit more formally [Cardelli 1985], let TY be a type, TY:M a method M of type TY, and ARGLIST a cartesian product over the types in the language. The signature of Thread::call is TY*TY:M*ARGLIST where ARGLIST conforms to the signature of TY:M. The result type of Thread::call is the result value of TY:M. In operational terms, do the standard C++ type check, select the method, generate the correct argument list (coercion), BUT do not invoke the method. In current C++ and the proposed Template extensions by Stroustrup [Stroustrup 1989], the type checking is done only if method invocation syntax is used. What if

the underlying system is written in C as most thread systems are (SunOS, Taos, uSystem, etc.)? We might need an interface into something like:

```
thread_invoke(o.update, nargs, <ptr to obj>, <arglist>);
```

At no point in the template is C++ invoking the method and thus a type check is not being done. While it is common for programming languages to ignore issues like these, C++ should take the lead in the Unix community for better software engineering practices. Perhaps it may require the introduction of meta-notation similar to the old lint method of annotating code fragments (an undesirable result)

C++'s model of computation assumes a single thread of execution, compilation rather than interpretation and a shared address space. Many of the problems we have encountered can be traced to these assumptions. An issue which comes up in all distributed systems is how to actually do the remote invocation and argument binding for functions, which has no corresponding analog in the C++ model of computing. Having a first class abstraction for invocation and argument lists built into the language would give the programmer a type safe interface to these objects, and facilitate the process of delegating the invocation to remote agents and marshalling arguments "across the wire". The ability to get a handle on a thread invocation would allow for the implementation of several styles of concurrent programming. Let us assume that C++ is extended with Stroustrup's proposal for parametric types [Stroustrup 1989], and we have some way in C++ for t.call to return a handle to an invocation:

```
// assume INVOKE_HANDLE contains a handle on the thread id
template <TY>
class future {

        INVOKE_HANDLE<int>  fh;

    public:
                operator TY() {
                // if function is done return value else
                // do a thread join(IH.thread_id)
                };
                operator=(INVOKE_HANDLE<int>);
};


class Thread {
        INVOKE_HANDLE<typeof obj::*> call(obj, obj::*, ARGLIST)
}                                       // * is universally quantified
```

```
{

   future<int> futi;
   futi = t.call(o, o.m, args);   // o.m returns an int

   // many clock ticks later ...
   int a = futi;                  // future<int>::operator TY()
                                  // coercion will wait if necessary
}
```

There have been many extensions made to C++ to support distributed computation. Some have proposed new language constructs such as Michael Tiemann's wrappers [Tiemann 1988] which are specifically intended to aid in building distributed programs. We argue that the proliferation of different concurrent computational models will severely stress any language that can not be easily extended, and that the semantics of method invocation should not be fixed in stone as in current implementations of C++ and other sequential programming languages. Programs should be able to get handles on the fundamental building blocks of invocations so that they can manipulate invocations with semantics differing from the current semantics inherited from simple procedure calls. Concurrent object oriented programming supporting classical rpc, futures, asynchronous send/receive, and passive vs. active objects should be expressible in one programming language. Static typing as in C++ should be preserved so programmers understand what they are writing and compilers have sufficient information to do a good job of optimization.

Distributed Computing with ISIS

In order to run Wave++ in a distributed environment, we use ISIS, a distributed programming toolkit designed and implemented at Cornell [Birman et al. 1987; Birman et al. 1989]. This choice was made to limit the amount of code we need to write, and to take advantage of features such as a simple communication interface which can support broadcast and replicated data, a lightweight task system running within each process, support for replicated service which supports fault tolerance, the ability to perform high speed state transfer upon node failure and recovery, and a uniform name space. ISIS is written in C and has interfaces for C, Fortran, and Lisp. We are currently in the process of porting Wave++ to ISIS and developing a C++ interface into the fundamental structure of the ISIS system.

The layout of distributed WAVE++ under ISIS is to run with one master process which does the problem setup, such as geometry layout, and perform any interaction with the user. Each region in WAVE++is distributed to a separate process running under ISIS. The methods of a region are mapped to ISIS entries, which allow for concurrent execution within the ISIS process by the use of a lightweight process system. Since passing particles between regions is a method invocation, we can map this into ISIS entries which allow multiple neighboring regions to update the same region concurrently. ISIS supports locks which may be needed if simultaneous method invocations need

to modify a single member data structure. In many real problems there will be more regions than processors, so there will be multiple processes running on each processor. In our current implementation, the design is asymmetric with different process bodies, one for the master process and many of the same kind for the region processes. The constructor of a region will be invoked in the master process which will in turn fire up the ISIS process for a region somewhere on the network. This region process is currently a kludge, since we are having difficulty in mapping the notion of a C++ object to an ISIS entity. The next release of ISIS (2.0) should eliminate the problems in the current ISIS interface which inhibit using ISIS directly from a C++ program. A single region will be declared as a global in each process and each public method will be indirectly called by a C function which references the global object. It is clear that at this time we are lacking language level support for distributing objects.

```
class Region {
   public:
       Region();
       init(geom);
       m1();
       m2();
       m3();
};

Region R;

initialize() {
       Geometry geom; // process message args, stuff into geom
       R.init(geom);
}

method1()
{
   R.m1();
}  // same for 2 & 3m

main() {
   isis_entry(initialize,...);
   isis_entry(method1,...);
   isis_entry(method2,...);
   isis_entry(method3,...);
   isis_start_main();
}
```

We are trying to stay within the confines of C++, and we are not language implementors (although we are certainly language design kibitzers), so the process of generating this code is done by hand. Fortunately, the object structure which resulted in sequential Wave++ maps well into this form. In

the long term, we will need to create an distributed object class where the master's version of an object is simply a handle to the object which resides on the worker process.

Scientific Programming with C++:

We have discussed several issues with respect to integrating concurrent programming models into the sequential model presented by C++. In the process of developing this and other scientific codes we have encountered some fundamental limitations which hamper the acceptance of C++ by the scientific programming community. Specific issues include optimization, storage management, and the implementation of aggregate arithmetic types (Matrices). We expect that the resolution of these issues will require changes to the language, and that proposed extensions to C++, such as parameterized types, could significantly further the acceptance of C++ as a scientific programming language.

Optimization

The use of class objects to represent individual particles results in a completely scalar calculation of the particle motion. Particles are handled very efficiently in a vectorized fashion by the Fortran WAVE code. In order for WAVE++ to favorably compete with WAVE, some amount of vectorization and optimization must be regained in the object oriented implementation. Currently vectorization over arrays of objects is difficult, as object methods tend to hide the underlying vectors spread across a linear array of objects. Additionally, the space-wise encapsulation of data members within an object spreads out components over strides large enough to impede some vector engines. Similarly, machines with caches will suffer lower cache hit rates.Ideally, we would like to stick with the C++ view of an array of objects, but have the underlying object data stored with one contiguous vector for each data member. The current semantics of the `const` modifier(which, with some help from inlining, is the only available channel for communicating data dependency information to the compiler) are not sufficiently strong to enable even the best of compilers to successfully perform some simple procedural integration and loop manipulation which would otherwise enable vectorization.

Storage Management

Vectorization is but one part of the performance issue. Because of the enormous size of problems to be solved, efficient management of the data's memory store is also a critical factor. We discuss below some significant problems encountered in developing aggregate arithmetic types with respect to data storage and unnecessary data duplication. Problems currently solved in Fortran on our Crays often utilize the full memory store. As these are non-virtual memory machines, it is essential that C++ implementations do not waste the store gratuitously, or they will never be able to handle sufficiently large problems to be of interest.

Aggregate Arithmetic Types in C++

The overloading of operators as object methods is one of the great attractions of C++ for the community of scientific programmers. A primary goal of a matrix/linear algebra package is to enable programmers with strong mathematical and physics backgrounds, but cursory programming skills, to develop algorithms by focusing on the science rather than on implementation and coding details (the goal of all C++ class library builders). The primary constraint in this context is to develop a set of tools that are as dependably efficient as they are notationally familiar. However, anyone who has made a concerted attempt to implement a usable matrix/vector package has no doubt faced a series of difficulties and disappointments with the current C++ language. Bjarne Stroustrup himself has stated [Stroustrup 1988] that he couldn't think of a worse undertaking for a first foray into C++ programming. To our knowledge, there has been no substantial published discussion of the class of problems that arise in this context [Lea 1989], although much of it has become general lore.

Consider an initial naive implementation of a Matrix class. We'll start out with only a simple row-major, dense representation (Warning: not recommended for use in the home):

```
class Matrix {
   private:
               int       r,c;
               double*   data;
   public:
                         Matrix(int m, int n)
                             :r(m), c(n), data(new double[m*n]) {}
                         Matrix(Matrix&);
                         ~Matrix() { delete data; };
               int       rows() const { return r; };
               int       cols() const { return c; };
               double&   operator () (int i, int j)
                             { return data[i*c+j]; };
               double    elem(int i, int j) const
                             { return data[i*c+j]; };
               Matrix    operator + (Matrix&) const;
               Matrix&   operator = (Matrix&);
};

Matrix::Matrix(Matrix& m) {
   r = m.rows(); c = m.cols();
   data = new double[r*c];
   for(int i = 0; i < rows(); i++ )
       for(int j = 0; j < cols(); j++ )
           (*this)(i,j) = m(i,j);
}
```

```
Matrix Matrix::operator + (Matrix& m) const {
   ensure_conformance(*this, m);
   Matrix t(rows(), cols());
   for(int i = 0; i < rows(); i++ )
       for(int j = 0; j < cols(); j++ )
          t(i,j) = elem(i,j) + m.elem(i,j);
   return t;
}

Matrix& Matrix::operator = (Matrix& m) {
   if (&m == this) return *this;
   delete data;
   r = m.rows(); c = m.cols();
   data = new double[r*c];
   for(int i = 0; i < rows(); i++ )
       for(int j = 0; j < cols(); j++ )
           (*this)(i,j) = m(i,j);
   return *this;
}
```

Now consider the following code fragment:

```
Matrix a(M,N), b(M,N),c(M,N),d(M,N);
 // assign values to a, b, and c here ...
d = a + b + c;
```

Internally C++ converts the last statement above into the following sequence of statements (or something similar):

```
Matrix temp1 = a.operator+(b);
Matrix temp2 = temp1.operator+(c);
d.operator=(temp2);
```

Note that two temporaries have been created by the compiler, and that each invocation of `Matrix::Matrix(Matrix&)` or `Matrix:operator=` involves copying the entire data array from the source to the destination Matrix. For large matrices, the space requirements for the two temporaries may well exceed the available memory. Instrumenting and testing this example under both the AT&T 2.0 and GNU 1.37 compilers indicates that three copies of the data array are performed, two by the `Matrix(Matrix&)` constructor and one by the assignment operator. As one could hand code the expression above with no copying of data or temporary creation, these copies and temporaries are unnecessary. As noted previously, this wasteful management of memory resources can make all the difference between a usable and an unusable scientific code.

The problem faced with temporary management is analogous to temporary register management in the compilation of expressions of primitive types such as Real and Int. An optimizing compiler knows when it can recycle a temporary register and generates the appropriate code. For non-primitive types the compiler does not perform this storage management function, thus we must mimic this function at run time. We have identified several techniques for implementing run-time temporary control for aggregate objects (the methods we are describing are applicable to any object implementing a pointer to heap-allocated store). The first technique is for each object to maintain internal state reflecting whether this instance is bound or temporary. This state is accessed and modified by the constructors, assignment operators, and a handful of manipulator and accessor functions. A second technique is to implement a helper temporary class, so that an object's type can supply the necessary state information. This technique can be expanded to include a complete run-time expression evaluator for each aggregate type. A third technique is to fall back on reference counting and copy-on-write semantics for the object's store, thereby avoiding the need to know anything of an object's temporary status. A fourth and less desirable technique is to circumvent the entire issue by avoiding constructive operators and function calls completely, relying on a procedural implementation much like that one would use in C. As there are many pitfalls to these techniques, we have provided a more detailed synopsis of the problem of temporary control in an appendix.

The lesson to be learned is that it is very difficult to implement aggregate arithmetic types in C++. We hesitate to place our finger on exactly *why* it is so difficult, but the problem seems to involve the inability to specify sufficient semantic information about a given object type to the compiler.

Squeezing Performance out of Inheritance Hierarchies

In an implementation of a matrix package, the class Matrix would be an *abstract base class* [Lea 1990], defining only the semantics of a matrix and the generic interface to a Matrix object through the use of pure virtual member functions. All implementation details would be reserved for various publicly derived types (such as DiagMat and SymmetricMat), frequently implementing the virtual methods as inline code. The code fragment below will function properly for arguments of all types derived from Matrix, but will require three virtual function invocations on each pass through the inner loop.

```
void add(const Matrix& a, const Matrix& b, Matrix& result) {
   ensure_conformance(a, b, r);
   for( int i = 0; i < r.rows(); i++ )
       for( int j = 0; j < r.cols(); j++ )
          r(i,j) = a(i,j) + b(i,j);
}
```

These invocations are far more costly than the actual addition of elements. It is desirable to have a version of this `add()` routine for all possible combinations of subtypes of class Matrix. These versions of `add()` are identical except for the types of the formal parameters and the use of the corresponding explicit subtype method invocations. This permits the compiler to do inline expansions of what would otherwise be virtual function calls, and to perform the usual optimizations.

At present, the best one can do is to generate all the necessary or appropriate function definitions by hand. Intelligent editors and awk or perl scripts can be trained to do the source code generation from a template. However, for a three-parameter function like `add()` above, and with ten matrix subtypes (not an unreasonable number), there are one thousand possible combinations of parameter types (an unreasonable number).

We need a mechanism for folding function definitions with inheritance hierarchies, which we believe goes beyond the capabilities of parameterized function definitions. Such a facility would allow the programmer to specify a prototypical function or method based upon the semantics and interface of an abstract base class, and the compiler would assume responsibility for generating more refined and integrated instances of the method whenever more highly refined (wrt derivation) argument lists are known at compile time. In these proceedings Douglas Lea proposes a set of "customization" extensions to the C++ language to support such a notion of inheritance-based parameterization.

Parameterized Types

Adaptive grids, kd trees, quadtrees, and octtrees are all parameterized types which we could use immediately in our code. The parameterization of arithmetic and algebraic *classes* such as matrices and vectors would greatly simplify future work in WAVE++. Beyond the obvious utility of matrices of integral, floating point, and complex elements implemented via parameterization lies the ability to *nest* parameterization, e.g. matrices of matrices. The representation of sparse-block or block diagonal systems as matrices where each element is itself a matrix of floating point values would simplify the implementation and parallelization of block parallel algorithms for solving the electromagnetic field equations, especially for implicit particle simulations.

Conclusions

From our experiences with the development of WAVE++ and basic arithmetic and algebraic classes, we have found that C++ provides a useful and powerful paradigm for building modular and easily extended physics codes. Data abstraction techniques can be used to closely, but not perfectly, match the mathematical representations to the physical objects and also provides a nice paradigm for writing parallel codes. The limitations of C++ that we have encountered involve difficulties in providing the proper information to a compiler for significant optimization and vec-

torization, and the lack of a good syntax for expressing parallelism and referencing distributed computing objects.

Appendix

Temporary Management with Aggregate Types

The critical issues in temporary management are unnecessary data copying and data store utilization. By implementing aggregate types which allocate storage from the heap one can, under many circumstances, replace element-wise copying of data with an exchange of storage block references. Using this technique, temporaries become nothing more than convenient short-term stewards for storage blocks of data generated and passed around during expression evaluation. The problem becomes one of developing a sufficient semantics of temporary aggregate objects that can be implemented under the constraints of the current language.

To paraphrase the AT&T C++ Language System Reference Manual [A.T.&T. 1989], there are only two things which may be done with a temporary object once constructed. Its value can be fetched once for use in another expression, in which case the temporary may be destroyed immediately; or the temporary may be bound to a reference, in which case it cannot be destroyed until the reference goes out of scope. Any time an object is created or assigned from another like object which can be identified as a temporary, it is possible to exchange "ownership" of the data store from the temporary to the target object, providing that the possibility of binding a temporary to a reference is eliminated. All that is needed is a mechanism for determining whether an object is bound or temporary. This may be accomplished through manipulation of an object's internal state or through type information. Since temporaries occur as the result of expression evaluation, it is sufficient to ensure that all methods/functions returning aggregate objects by value set the state (either by manipulating internal state or by typing) of the returned value to indicate temporary status.

There are several techniques whereby the determination of bound state and the transfer of data storage can be implemented. One technique is to include some state in the object itself describing who owns the data storage and whether the object is temporary or bound. This state can be manipulated by the various constructors and assignment operators, as well as some carefully designed manipulator methods. While relatively easy to implement, the drawbacks of this method are rather severe. In order to ensure that a temporary is never bound to a reference, it suffices to follow two cardinal rules: always pass into functions by value (except under special circumstances), and never bind an expression or a function return value to a reference explicitly. This implies that function arguments must be passed by value, which is reasonable as long as the actual parameters are always temporaries. This is most certainly an undesirable misfeature, and a major performance penalty is imposed. (This problem may be partially circumvented by combining this technique with the indirected approach discussed shortly.) Also, it inevitably imposes

upon the end user the use of certain awkward constructs in order to manipulate the state information. For example, consider the following function which returns a Matrix by value using this technique:

```
Matrix IdentMatrix(int n) {
Matrix t(n,n);
for( int i = 0; i < n; i++ )
    t(i,i) = 1.0;
t.release_data(); // mark storage for release
return t;
}
```

It is necessary to mark the storage held by the local object `t` as available to be stolen by the constructor of the return value. In fact, `Matrix::release_data()` must mark the data storage as twice-exchangeable: once to pass the storage from the local variable to the temporary return value, and once again to pass from the temporary return value to whatever the next or final destination may be. For this reason, it is necessary to count the passes through various constructors and assignment operators to determine proper bound state.

Herein lies a serious problem of this technique. C++ compilers may also occasionally take various short-cuts. Although they frequently create temporaries, the number of these temporaries can vary from compiler to compiler and optimized compilations may even eliminate the construction of useless temporaries, under the implicit assumption that constructors perform no useful computation by side effect. There is no way to consistently ensure how many constructors are invoked in passing out of a function returning an aggregate by value to a constructor of a like typed destination. This uncertainty renders it difficult ensure that a bound object *always* owns its data (except when explicitly released) and that a temporary object never thinks itself bound. Although we have succeeded in using a matrix implementation similar to what we describe here in a large code, it was at the expense of banishing the `X x = f()` syntactic form from our vocabulary!

A second possible solution to the problem of temporary control is that of using a special helping class to hold temporary object values. By carefully avoiding the binding of an instance of the temporary helping class to a reference, we can freely treat the ownership and contents of a temporary object as we wish. In the context of our Matrix example, we can implement our matrices by creating a tmpMatrix class in parallel with class Matrix. We ensure that *all* constructive functions and methods return a tmpMatrix object, and that a tmpMatrix is never bound. By providing `Matrix::Matrix(tmpMatrix&)` and `Matrix::operator=(tmpMatrix&)` methods which always steal the data and data ownership from the tmpMatrix, we can ensure that all bound Matrices own their storage.

This technique does have some of the same notational convention problems as the first proposed technique, but it is not susceptible to the problems of constructor elision en-

countered with the earlier technique. In fact, when combined with the indirected reference-counting scheme discussed below, many of the notational inconveniences can be avoided. Furthermore, the code can be structured in such a way as to always guarantee correctness under all constructor and expression syntactic variations save one - explicitly declaring a bound tmpMatrix. Failure to follow certain cliches may introduce some runtime inefficiencies, but the code will always function correctly.

A third possible solution to the problem of temporary control and excessive copying is to implement the object class as an "intelligent pointer" to a reference counting representation class. Construction and assignment of the object class can simply pass around a handle to a representation object which keeps track of all live references and implements copy-on-write semantics. In fact, this technique may be used to advantage when combined with either the previous techniques.

One significant problem, however, is that the language specification for temporary destruction requires only that a temporary be destroyed before control exits the most closely encompassing scope. This means that after assigning from a temporary to a bound object, a compiler may keep the temporary and its associated reference to the underlying representation object alive long enough to force an otherwise unnecessary copy-on-write. Additionally, reference counting and the associated indirection can carry a significant performance costs. For small objects and on machines not very adept at indirection, this cost may well be prohibitive. However, on modern day RISC architectures and with large aggregate objects (our work often involves 100 x 100 or 10,000 x 10,000 matrices), the gains in copy avoidance and reduced storage requirements can far outweigh the costs of reference counting and indirection.

Of course, there is a fourth technique available, which is to simply avoid the use of *any* constructive functional forms. Rather than utilizing operators, one can employ the sort of procedural technique that would ordinarily be used in C, passing in by reference the operands and a pre-constructed object to receive the results of the procedure call. Although occasionally appropriate, this technique violates our sense of what C++ and object oriented design should allow us to do with aggregate arithmetic types.

A root of the difficulties in implementing aggregate arithmetic types, as Doug Lea has pointed out [Lea 1988; Lea 1989], is that C++ makes no assumptions about the semantics of overloaded operators. This renders any higher-level compile-time optimization of expressions containing constructive operator invocations impossible. It also compels one to consider the possibility of implementing expression evaluation and optimization at runtime, by use of expression-constructing object operator methods. Done on a class-by-class basis, knowledge of the full semantics of an arithmetic type can be exploited. One particular advantage in the Matrix context is the exploitation of the commutativity of matrix multiplication to reduce the operation count of multi-factor products of non-square matrices.

References

A.T.& T. C++ Language System Reference Manual. 1989.

B. N. Bershad, E. D. Lazowska, H. M. Levy and D. B. Wagner, An Open Environment for Building Parallel Programming Systems, *SIGPLAN PPEALS Conference 1988*, 1988.

K. Birman and T. Joseph, Exploiting virtual synchrony in distributed systems, *Proc. 11th ACM Symposium on Operating Systems Principle*, 1987.

K. Birman, R. Cooper, T. Joseph, K. Kane and F. Schmuck, *The ISIS Systems Manual*, Version 1.2, 1989.

L. Cardelli and P. Wegner, On Understanding Types, Data Abstraction, and Polymorphism, *ACM Computing Surveys*, vol. 17 no. 4, 1985.

D. W. Forslund, Plasma Simulation Techniques, *Space Science Reviews*, 1985.

D. Lea, Customization in C++, *USENIX C++ Conference Proceedings*, 1990.

D. Lea, Lecture on Embedded Languages in C++, July 1989.

D. Lea, Users Guide to the Gnu C++ class library, Free Software Foundation, 1988.

B. Stroustrup, Private Communication, Oct 1988.

B. Stroustrup, Parameterized Types of C++, *USENIX Computing Systems*, vol 2, no. 1, 1989.

M. Tiemann, "Wrappers:" Solving the RPC Problem in GNU C++, *USENIX C++ Conference Proceedings*, 1988.