Object-Oriented Programming for Scientific Codes. I: Thoughts and Concepts

By T. J. Ross,[1] Member, ASCE, L. R. Wagner,[2] and G. F. Luger[3]

Abstract: This paper is an introduction to object-oriented programming (OOP) as it relates to the development of large-scale scientific codes. It is the first of two papers describing OOP issues for scientific-code development. Since object-oriented methods provide for the encapsulation of data and methods, scientific codes can be written in terms of the underlying physics of the problem with less regard for computer-science details. We discuss some of the features of large-scale scientific codes that are amenable to object-oriented design. The concept of *efficient portability* is elucidated, explaining why object-oriented user codes need not be altered when moving code to radically different computer architectures. In fact, we envision codes running without major changes on serial machines, such as a Sun or Vax; vector machines, such as a Cray; and on massively parallel processor systems, such as the Connection Machine. Comparisons between FORTRAN and the object-oriented programming language of C++ are illustrated with simple examples of matrix operations.

Introduction

Computers remain an incomplete tool for engineers and scientists. They are a useful tool in that they can perform computational tasks of a magnitude far beyond the capability of any human. They are incomplete in that, for full utilization of the underlying machine, it has been necessary to view them as "white (transparent) boxes"—a user must specify not only *what* to do, but *how* to do it. For example, to fully utilize vector machines such as the Cray, an engineer must be sure that FORTRAN routines allow the underlying system to vectorize, which in turn requires some knowledge of what vectorization is and how it is implemented. This is somewhat akin to needing to know a little metallurgy before using a hammer.

In an ideal world, all that would be necessary to solve a system on a computer would be to describe the physical situation as well as the appropriate boundary conditions. Details such as choosing the algorithms and data structures to use the underlying hardware would be handled by the programming environment (i.e., computer science details would be handled by computer scientists) in much the same way that the phone system figures out an efficient routing of a call without any elaborate user specification.

The capabilities of object-oriented programming (OOP) in recent years have now made this programming paradigm a very promising tool for the development and implementation of large-scale scientific codes, especially those requiring implementation on the newest computer architectures. OOP techniques are now applied to a wide variety of applications (Booch 1990). It has been recently used in graphics (Vlissides and Linton 1988), in object-

[1]Assoc. Prof., Dept. of Civ. Engrg., Univ. of New Mexico, Albuquerque, NM 87131.

[2]Computer Sci., Kachina Technologies, Inc., Albuquerque, NM 87112.

[3]Prof., Dept. of Computer Sci., Univ. of New Mexico, Albuquerque, NM.

Note. Discussion open until March 1, 1993. Separate discussions should be submitted for the individual papers in this symposium. To extend the closing date one month, a written request must be filed with the ASCE Manager of Journals. The manuscript for this paper was submitted for review and possible publication on August 11, 1991. This paper is part of the ***Journal of Computing in Civil Engineering***, Vol. 6, No. 4, October, 1992. ©ASCE, ISSN 0887-3801/92/0004-0480/$1.00 + $.15 per paper. Paper No. 2379.

oriented data bases (Howard 1991), in software user interfaces (Gorlen 1987), and most recently in scientific code development (Wagner et al. 1991; Angus and Thompkins 1989; Forslund et al. 1990).

OOP has tremendous potential in the scientific software business. This approach to programming computational codes promises to produce more and more easily maintained software for less effort and expense. Conventional software labors under the weight of seemingly endless lines of code, while OOP allows programmers to build an application program by using existing or easy-to-build modules called "objects." The modules become active when they are attached to the rest of the computational code. Better yet, they become active without disrupting any other parts of the application. Moreover, the modules can be removed entirely without causing the application to breakdown into electronic incoherency. For instance, library "objects" can represent traditional numeric packages. This method takes the place of many individual custom lines of code, saves time, and makes it easier to spot and reduce errors. (Most popular large-scale scientific codes have undergone so many revisions, with so many groups, that they resemble Rube Goldberg schematics.) Therefore, it seems natural to apply OOP to the design of modules for scientific computations.

This is the first of two papers [see also Ross et al. (1992)] describing OOP issues for scientific code development. This paper discusses how a scientific problem is decomposed using object-oriented design for computer modeling. It also provides comparisons between the most popular scientific programming language of FORTRAN and the object-oriented programming language of C++. Finally, some simple examples of matrix operations using C++ are provided for the purpose of illustrating some of the OOP issues discussed herein. The second paper (Ross et al. 1992) offers some specific numerical code applications using C++ and discusses optimization issues for OOP.

Object-Oriented Design

In object-oriented programming, any physical or logical entity in the model is an "object". The definition of a type of object is called a "class," and each particular object of that type is known as an "instance" of the class. The definition of operations on or between objects are called "methods," and the invocation of the methods is referred to as "passing a 'message.'" If a method is defined as a symbol (e.g., +, *, =), then the method is often referred to as an "operator." For example, in Fig. 1 we have a class called matrix, which includes the methods on matrices such as the definitions of the operators = (assignment) and * (multiplication). Entities **A**, **B**, and **C** are declared to be instances of the class matrix; matrix is the type of object, and **A**, **B**, and **C** are specific instances of this type. In order to multiply **A** and **B** and store the result in **C** (**C** = **A***B), messages involving the operators * and = are passed from **A** and **B** to **C**.

The distinction between a method and a message can be illustrated by referring to the analogous terms in the procedural language FORTRAN. In FORTRAN, a method is the definition of a function or subroutine (procedure), and a message is like the CALL statement (i.e., the invocation of the function or subroutine). In this paper the terms "function," "procedure," and "method" are used interchangeably. Because the word "method" is so pervasive in this paper, it is italicized when used in the specific context of a function attached to an OOP class, e.g., + *method* on class matrix is


```
// partial C++ class definition
class matrix    {
                ...
                matrix  operator = (matrix&);              // method =
                matrix  operator * (matrix&,matrix&)       // method *
                ...
                };
...
// user code
matrix  A,B,C;          // A, B, and C are declared instances of the class matrix
...
C=A*B;                  // messages = and *
```

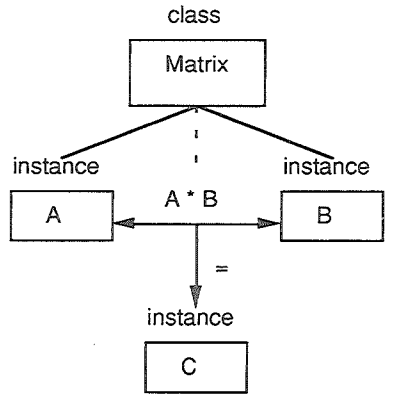


FIG. 1. Example of Matrix Objects

italicized. When used as an approach, such as the finite element method, it is not italicized.

Operator (or *method*) "overloading" allows the programmer to think in the same terms for application of a function to two different objects. An example is printing an integer or printing a string, where we have two different objects but the same desired printing operation. C++ allows *any* function or operator to be overloaded and the routine called will depend on the type of the operands, that is, the object to which the message is sent. This concept is used in a limited manner in conventional programming languages. For example, the operator * is overloaded in FORTRAN. The compiler knows that integer*integer returns an integer, whereas real*real returns a real. Is there any reason why multiplication of integer matrices should be referred to differently from the multiplication of real matrices?

OOP languages such as C++ allow for objects to be placed into a conceptual hierarchy. Thus, a class has ancestor classes preceding it in the hierarchy, and the class is said to "inherit" the properties of its parents, which inherit the properties of its parents, etc. Instances of a class also inherit methods from the class. Class hierarchies (subclasses) allow the programmer to specify that certain *methods* are inherited (while others may not be) from the superclass to the subclass (Luger and Stubblefield 1989). For example, consider the following C++ definitions of 1-D and 2-D nodes in a numerical code implementing a finite difference or finite element analysis, as shown in Fig. 2.

Node1 contains a location in one dimension, internally referred to as **x**.

```
Note:  double means double precision floating point
       void means there is no return type (i.e., it's a procedure - not a function)
       cout is the standard output stream (usually a file or the display)
       << is a message between an object on the right and an output stream on the
          left meaning, essentially, "print me"

class Node1
        {
        protected:    double      x;
        public:       double      getx()  { return x ; }
                      void        print_location()      { cout << x; }
        }

class Node2 : Node1
        {
        protected:    double      y;
        public:       double      gety()  { return y; }
                      void        print_location()      { cout  << x << y;}
        }
```

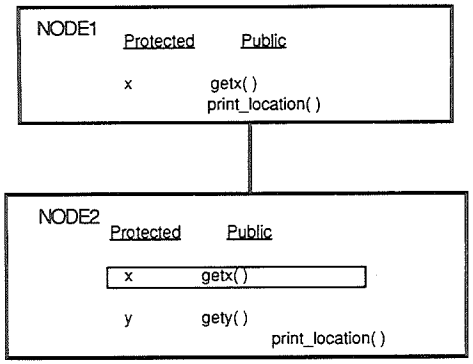


FIG. 2. Inheritance in Object Node

It has a routine **getx** to retrieve this value and a routine **print_location** to display its position. **Node2** *inherits* all that **Node1** contains (small rectangular box in Fig. 2), and it adds a location in the second dimension, **y**, as well as a routine for retrieving it, **gety**. Also, **Node2** redefines the meaning of **print_location**. This allows user code to print the location of a node without worrying about what kind of node it is. In Ross et al. (1992), we demonstrate that user code based on an object structure of this type for handling a one-dimensional finite difference problem can be easily modified to handle a similar problem in two dimensions.

OOP also allows for data hiding, also referred to as "encapsulation." In Fig. 2, "protected" things are known only to class and its descendent classes in the hierarchy, whereas "public" things are known to all users of the class. It is also possible to declare part of a class to be "private"—known only to the class. The idea is to make the *methods* that manipulate an object public, while making the variables and methods that are used to implement an object either protected or private. This is exactly how an abstract data type is defined. The implementation of the object is said to be encapsulated so that only the object itself knows how *methods* are performed; and yet the functionality of the *methods* is offered to the outside. The focus is on the data being manipulated, not on the procedures used for the manipulation. Further advantages of data encapsulation will be illustrated in an example.

Object-oriented programming has its roots in software engineering, emphasizing modular development for increased software extensibility and reusability as well as better control of complexity and cost of software maintenance. Extensible programs are developed incrementally. This type of problem solving is achieved through an overall top-down design in conjunction with a bottom-up approach to the implementation details. The object-oriented approach allows the programmer to add functionality to a program by using the existing structure to add new instances and class definitions so that whole code modules do not have to be modified or completely rewritten. Reusability means that there is a distinct separation between the functionality and actual implementation of a program or subprogram. In this way the programmer can use an existing set of class definitions without knowing how the *methods* are implemented. Maintainability and reliability are enhanced because changes that are needed in an implementation will be localized to a specific object module. Thus object-oriented programming allows for rapid prototyping of an application.

Historical Development of OOP Environments

Smalltalk was the first design of an OOP language, developed in the early 1970's at Xerox Palo Alto Research Center under the direction of Alan Kay. The original Smalltalk interpreters were built as extensions of the LISP environments and were used primarily as simulation-based applications. The roots of the Xerox work may be found in the SIMULA and MODULA families of languages as well as the LOGO language produced by S. Papert at MIT (Papert 1980) at about the same time. The philosophy behind Smalltalk was that of "letting the programmer solve a complex problem by simulating the interactions of the components of the problem." This approach is sometimes referred to as a "homomorphic representation" where the objects of the problem situation are mapped down into computational objects. Of course the procedures or *methods* of these computational objects reflect the functions the entities have in the actual problem (Goldberg 1984).

Various other researchers added to the OOP approach. Smalltalk created objects as a result of both specifications and procedural abstractions across the entities of the problem domain. Thus the function of all individual object instances in a problem domain would be represented by the definitions and *methods* contained in a class. Each instance would then "inherit" the *methods* belonging to that class. Object-oriented languages extensions to LISP, such as CLOS (Common LISP Object System), allow multiple inheritance, the ability to inherit from any number of parent classes (Keene 1988). We think this addition has proven to be an important supplement to the original Smalltalk design.

While the original OOP languages were all LISP-based it was only natural that the same methodology should be created in the C language. This occurred in the mid-1980s with the appearance of C++ (Ellis and Stroustrup 1990) and Objective C (Cox 1986). The C++ approach is now much more generally used.

Recent Developments Using OOP for Numerical Codes

Recent research indicates significant promise for OOP languages and for C++ in particular. C++, unlike most OOP languages, was created with runtime efficiency and low storage overhead as explicit design goals. Thus, C++ is often employed in areas where these attributes are critical.

Angus and Thompkins (1989), have compared C++ with C for computational fluid dynamics codes. They found that the C++ code ran 1.5–3 times slower depending on the underlying machine. Their C codes were specially written for each architecture while their C++ user code was tested unmodified. Table 1 illustrates the speed of C++ on various configurations indicated by the number of nodes in use on a Hypercube. The goal of this seminal work was to demonstrate that C++ user code could be ported from one architecture to another without alteration, yet remain efficient on both. We expand upon this idea briefly later in this paper, as well as in Ross et al. (1992).

More recent works in the application of OOP techniques to scientific applications reveals the true potential of this environment. A recent dissertation (Segal 1990) presented work in the area of object-oriented approaches to computational geometry. This work specified a code written in C with an OO preprocessor. Since the code was not written in C++, it did not provide direct access to classes, it could not specify a distinction between public and private data access, and it was not efficient from an OO point of view. The work did make some cursory suggestions for finite element codes, however. For example, an "intangible" function was used to specify matrix spaces at a high level in the abstraction for situations where the basis vectors of the matrix were not yet known. Then a "tangible" function was used at lower levels of the hierarchical structure when the basis vectors became known. This has the effect in finite element theory of mapping element matrices in local coordinates into a global coordinate system where all elements are combined.

MacDonald (1988, 1989) and MacDonald et al. (1990) conduct research in object-oriented structures for numerical operations on algebraic equations. This work in CLOS addresses issues of conceptual clarity of object structures in this domain.

A recent paper (Forslund et al. 1990) describing a plasma-particle simulation code was written in C++ at the Los Almos National Laboratory. The paper reiterates the fact that although C++ "has not yet become of widespread use in scientific applications," it should provide "a better paradigm for distributed memory parallel codes." The particle simulation code was originally written in FORTRAN for serial computation, but is now being rewritten for employment on top of a distributed programming tool kit on a Sun workstation. In this connection the paper describes some of the difficulties encountered when using C++ for parallel scientific computation.

One thought currently being given to the implementation of *very* large scientific codes is to eventually harness several supercomputers together

TABLE 1. C++ on Hypercube iPSC/2 in Milliseconds per Grid Cell (Angus and Thompkins 1989)

	Number of Nodes in Grid					
Grid size (1)	1 (2)	2 (3)	4 (4)	8 (5)	16 (6)	32 (7)
10,206	8.92	5.36	2.96	1.69	1.02	0.79
76,055	—	—	2.55	1.31	0.69	0.42
250,588	—	—	—	—	0.66	0.37
586,845	—	—	—	—	—	0.33

over high speed channels. Unfortunately, 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 an Ethernet. As with conventional programming techniques, library modifications may be necessary to fully use this new architecture; however, user code written under the *efficient portability* paradigm need not be altered. It can immediately be run on the new system with larger problem sets being the only change.

Large-Scale Scientific and Computational Codes

Most large-scale scientific calculations are associated with the solution of partial differential equations (PDEs). These PDEs model realistic physical problems that are characterized by space and time, where the solution to a given physical system is desired in both space and time. The solution of PDEs for very simple systems can be accommodated in a closed-form fashion, but the number of such cases is very limited as is the value of their results. Most useful physical applications are so complex as to require extensive numerical computations.

The PDEs are described by classical mathematical operations on scalar fields and vector fields (e.g., curl, divergence, and gradient) describing the physical domain of the problem. Because of the complexity of the PDEs, discretization methods are commonly used to provide a solution.

The two most popular discretization methods used to solve complex PDE problems are known as the finite difference method and the finite element method. These two methods are very similar in that they allow the analyst the ability to model problems involving space and time coordinates, various materials, and various boundary conditions. For both the finite difference method and the finite element method two different coordinate systems can be used to describe the physical and geometric domain of the problem, a Lagrangian system and an Eulerian system. In addition, the PDEs can be solved with one of two schemes, the explicit method or the implicit method. All these methods and systems are illustrated with simple OOP implementations in Ross et al. (1992).

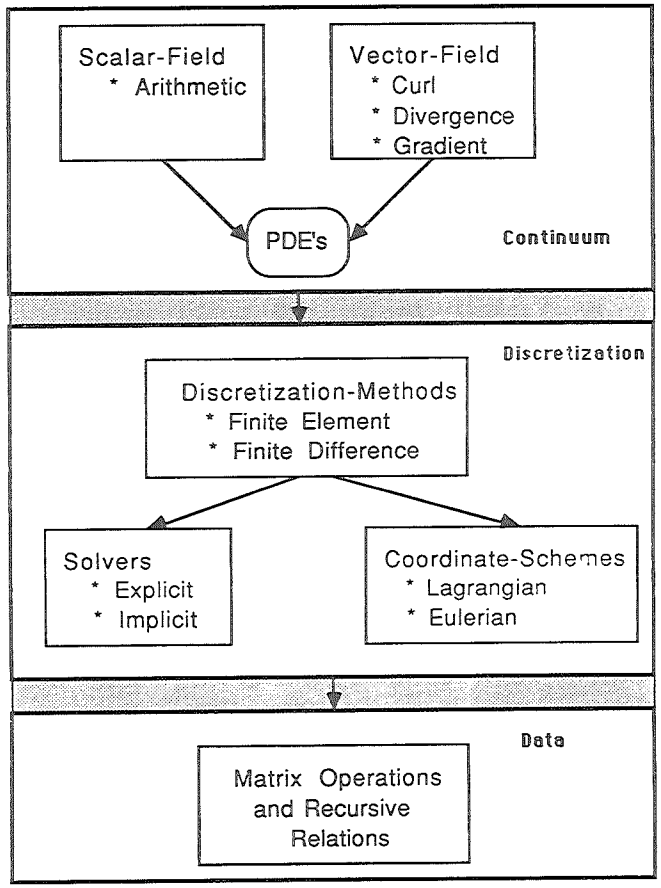
An OOP environment for scientific calculations must therefore address the various modeling methods (finite difference and finite element), coordinate systems (Eulerian and Lagrangian), and solution methods (explicit and implicit). Scientific codes lend themselves to an OOP environment because of their modularity and similarity. Most of the features of the various approaches are the same, and differ only in small ways as previously discussed. For example, a general code would contain coordinate system modules, modeling modules, and solution approach modules. More importantly, the various aspects of scientific codes lend themselves very nicely to an object-based abstraction. For example, *nodes* share many of the same characteristics in finite difference and finite element formulations; *elements* in the finite element method share some properties with *cells* in the finite difference method; and *material models* for the two methods are similar.

In an OOP environment, nodes and elements and material models can all be modeled as objects. Numerous objects will "inherit" properties from other objects and *methods* will operate on objects. For example, in a finite element code, the object element_velocity will inherit information from the object node_velocity, and the "implicit_solution OOP *method*" will operate on the object "node_velocity." Such convenient and similar descriptions

using object-based abstractions will allow the analyst tremendous power in combining various features of different solution approaches and in quickly building dedicated and reusable scientific software.

Fig. 3 shows a hierarchy that an engineer or scientist might use to decompose a physical problem so that it can be analyzed by a computer in an OOP environment. At the continuum level, the basic mathematics governing the properties of the system to be studied is specified. At the discretization level, the problem is decomposed into subpieces and interactions between these subpieces; this is the level where such things as boundary conditions and stability issues are implemented. At the data level, primitive structures (e.g., matrices) and operations are mapped to the computer architecture so as to best utilize the underlying machine. As the engineer or scientist has to delve further down this hierarchy in order to solve the problem, productivity in code development and modification decreases.

It would be best for code productivity if the transitions between continuum, discretization, and data layout were well defined. In other words, a user should be able to operate in the continuum and/or discretization worlds



without worrying about the actual data layout, which is, after all, simply a machine-dependency issue. Higher-level languages like FORTRAN were developed in part to hide the machine dependencies of fundamental data types like integers and reals, which was more difficult at the assembly language level. Now, these same problems reappear in OOP in dealing with aggregate data types such as matrices.

The essential problem with FORTRAN is that it provides no method for encapsulating the structure of the data. One of the greatest hindrances to the full exploitation of the concurrency offered by distributed memory machines is that, with these traditional languages, the programmer must explicitly specify the layout of the data storage. By doing this in an OOP language such as C++, the program design enhances the portability of application codes and, at the same time, improves the development and reusability of the software. Ross et al. (1992) describe a system for developing finite difference and finite element codes that never forces the user below the discretization level. At the lower levels of Fig. 3, libraries implement the software interface between the high-level objects that the user employs and a specific architecture. It is at this library level that the machine dependent details of data storage, parallelization, and synchronization are implemented.

Ideally, objects and the operations on them should be defined in the most general mathematical form, at which point the libraries can then determine what specific operation is performed, as well as the specific discretization methodology that is optimal. It would be best to develop continuum objects (e.g., Vector-Field) with continuum operators (e.g., curl) on them that handle more and more of the discretization steps. Eventually, one would like to do everything at the continuum level, where it is most accessible to and understandable by the scientist or engineer. The examples given here and in Ross et al. (1992) are, however, largely trapped at the discretization level. Extending them to the continuum level in a way that fully utilizes an underlying architecture is a quite nontrivial computer-science problem; however, work is being done along these lines (Wagner et al. 1991). Since it is difficult to automate the selection of a discretization scheme, a user's choices could occur at this level, where retention of as much of the elegance of the mathematics as possible could be accommodated.

Note that it is possible to extend an object system "upward" to handle a higher level abstraction, such as moving from discretization to continuum levels in an object library for scientific code. This will require substantial modification of the code unless this extension was kept in mind throughout the development of the original system. Thus, some of the same software engineering design problems remain as with conventional coding techniques. While OOP is not a panacea, the focus on concepts inherent in the paradigm eases the computer-science problem somewhat, and OOP retains its big advantage of divorcing the computer science (how the tool works) from the engineering problem (what the tool is being employed to do).

C++ Runtime Efficiency

The goal of the work reported by Angus and Thompkins (1989) was to demonstrate that C++ user code can be ported from one architecture to another without alteration, yet remain efficient on both. Table 2 illustrates the efficiency of C++ when compared to C on the same architecture. The interesting thing about this work is that, while the C code was customized

TABLE 2. Comparison of Sequential C to C++ in Seconds per Iteration on Sun-3 and Hypercube iPSC/2 (Angus and Thompkins 1989)

Machine (1)	C (2)	C++ (3)
Sun-3	72.4	191.8
iPSC/2 (single node)	23.1	78.9

for each architecture, they made no attempt in developing their libraries to take advantage of the underlying machine. For example, a matrix was simply defined as an ordered sequence of vectors, and the work was passed on the vector class. This is the obvious way of defining the library, but it is the first thing you would change if speed was critical. Note that a change in the library definition of matrix and associated operators requires no change in user code, which is usually not the case in standard FORTRAN.

C++ is not a pure OOP language, sacrificing some of the elegance of the paradigm in order to gain run-time and memory-usage efficiency. Some of the performance advantages of C++ over pure OOP languages are described in a paper by Ross et al. (1992).

Portability and Efficient Portability

User code is considered to be portable if it can be moved from one machine to another without change. *Efficient portability* is defined as the ability to move user code between different architectures without modification, yet retain run-time efficiency on both machines. A review of what allows code to be portable will illuminate what might be necessary for efficient portability.

FORTRAN is portable. It contains certain *fundamental data types* such as integers whose internal data structure is hidden from the user by the compiler. For example, the user need not know how negative numbers are represented in the machine. Operations on integers such as +, −, *, / are guaranteed to work, at least within certain machine-dependent ranges. Thus, for a FORTRAN program that performs integer arithmetic running on a 16-bit word, one's complement machine should work without modification on a 32-bit word, two's complement machine, even if the underlying machine languages are very different. The reason for this is that the data layout for integers has been encapsulated by the FORTRAN compiler.

C++ allows the user to define an object that is essentially a fundamental data type. At this point we emphasize that a major goal of a C++ environment is to allow user code to be portable and efficient across different architectures. For example, on a Cray a matrix might best be stored as a series of row vectors (or column vectors in FORTRAN) with operations set up so as to increment within the vector on the innermost loop; a matrix addition in FORTRAN for a Cray that incremented across columns instead of across rows within a column on its innermost loop would have no vectorization gain. On a Connection Machine (CM), however, it would be better to store a matrix in block form (see Fig. 4). Matrix addition is most efficiently implemented in block form as well, with each processor performing operations on fixed-size submatrices.

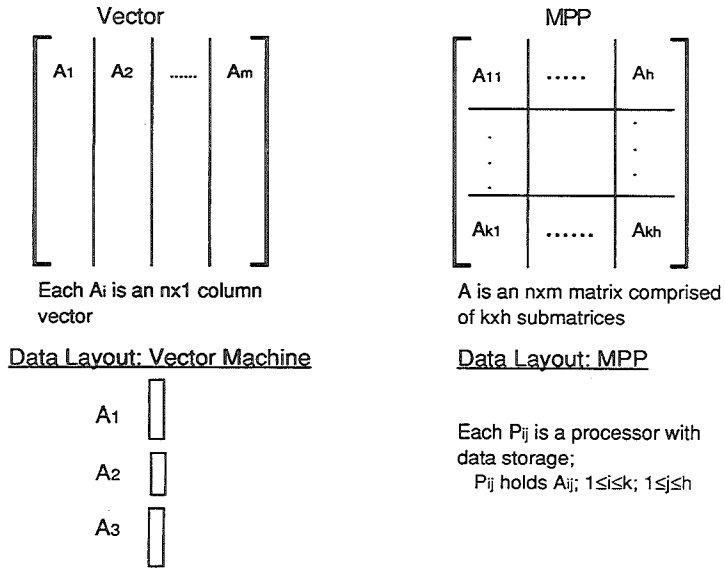


FIG. 4. Possible Data Layout of n × m Matrix on Two Architectures

Simple Example

The best data layout for a matrix is often critically dependent on the underlying architecture. As in Fig. 4, it may well be best to view a matrix as a sequence of column vectors on a vector machine such as the Cray, and to view it as a block matrix on a massively parallel architecture. The most efficient algorithm for various matrix operations will depend on the underlying data layout. Code designed for serial architectures is often horribly inefficient on massively parallel processors (MPP). Even porting between MPP machines may require fine-tuning such things as the optimal size of subblocks.

Furthermore, there are radically different processor topologies possible for different MPP architectures, and it may be wise to modify some underlying data structures when porting code between two different kinds of MPP machines. For example, in a Hypercube architecture, there are 2^n processors where each processor has n neighbors directly connected to it. This architecture favors algorithms that minimize passing data between processors that are not "neighbors." On a (fictional) MPP transputer where all processors lie the same distance from each other, very different data layouts and algorithms would be desired for efficiency purposes.

In FORTRAN, all routines that use a matrix "know" its data layout. Porting a FORTRAN user program from a Cray to an MPP machine will, at the very least, require altering all subroutine calls (including input/output [I/O]) and probably substantial rewrites beyond this. In C++, since the view of the data is encapsulated within the libraries, these rewrites are unnecessary. In other words, the libraries must be rewritten for each machine (as with FORTRAN), but C++ user code *need not be modified* at all to run efficiently, using the strengths and avoiding the weaknesses of different architectures. The goal is a system whereby lower-level algorithms

FORTRAN USER CODE (Cray Vector machine)

```
C Code fragment to read in A & B, set C=A+B, output C, optimized for Cray
C   Inputs a matrix, stores it in column major form
    CALL CRAYIN(A,N,M)
    CALL CRAYIN(B,N,M)
C   Adds A & B, stores result in C
    CALL CRAYAD(A,B,C,N,M)
C   Exports C stored in column major form
    CALL CRAOUT(C,N,M)
```

FORTRAN USER CODE (MPP)

```
C Code fragment to read in A & B, set C=A+B, output C, optimized for a
C particular MPP C machine
C
C   Inputs a matrix, stores it in submatrix form
C   each submatrix is NSUB by MSUB (this might be hardwired in MPPIN)
    CALL MPPIN(A,N,M,NSUB,MSUB)
    CALL MPPIN(B,N,M,NSUB,MSUB)
C   Adds A & B, stores result in C
    CALL MPPADD(A,B,C,N,M,NSUB,MSUB)
C   Exports C stored in submatrix form
    CALL MPPOUT(C,N,M,NSUB,MSUB)
```

C++ USER CODE

```
(* C++ code fragment optimized for Cray or MPP *)
    cin >> A >> B; // input A & B
    C = A + B;
    cout << C; //output C
```

FIG. 5. User Code Fragments for Matrix Addition

that are tied to the architecture can be written once, and higher-level code can be written by users who need not know or care about such underlying efficiency tradeoffs. Fig. 5 shows user code fragments for adding two matrices.

A properly constructed C++ library is not subject to some of the coding problems that could cause grief in FORTRAN. The C++ code fragment in Fig. 5 assumes A, B, and C all have known dimensions. The FORTRAN fragments in Fig. 5 also assume known dimensionality, but FORTRAN doesn't have a well-accepted method of giving this information to all potential subroutines at one time. If the user accidentally types CRAYIN(A, M, N), and this is buried in a much larger code, it may take forever to find; whereas, if the C++ user accidentally gives the wrong dimensions for *A*, the library routine "+" can note the discrepancy in row and column size of its operands and generate a run-time error of the form:

"attempt to add 2 matrices of incompatible row sizes in line xx of file yy."

Example User Code Written in C++ for Matrix Operations

A useful feature of OOP is its potential for developing a library where the user code specifies certain properties of an object and trusts the library to handle *all* of the details associated with this information. Some properties need not be described even by the user code. For example, matrix sparsity can be tested by the matrix input operator although intermediate knowledge

that the library can not infer from properties and operations may need to be supplied by the user.

Fig. 6 offers an example of what a user would do to perform a few simple vector and matrix operations (with several comments to highlight the flavor of basic operations in C++). The library file "vector.h" would include the actual vector and matrix class structures as well as associated operations. Fig. 7 shows sample input and output for the code in Fig. 6.

One benefit of C++ is operator overloading. If the user wants to multiply two matrices A and B, the user simply enters A*B; the creator of the matrix library defines the (*) operation. This is good for the user, because it is easy to understand what is going on in a program, and it is useful to the library implementer because everything is set up in a convenient modular form. Further, the library can determine which (*) operation is needed, without any attempts at efficiency in the user code. For example, in C++ if A and B are both sparse the library can automatically run a more efficient routine than if A and B are dense. The technique of allowing the library to select a specific algorithm for implementing a particular function could

```
/* vector.C        Lewis R. Wagner
 *      simple vector and matrix manipulations in C++
 */

#include "vector.h"

main()
{
  int      i;
  const int VEC_SIZE = 5;
  vector    a(VEC_SIZE),b(VEC_SIZE),c(VEC_SIZE),d;
  matrix    A(VEC_SIZE,VEC_SIZE),B(VEC_SIZE,VEC_SIZE);

  d.set_size(VEC_SIZE);     // demonstrates setting size of a vector dynamically
  cin >> a; // read in a vector
  cin >> A; // read in a matrix
  b=a;                      // vector = vector, copies the CONTENTS of a into b
  c=a+b;                    // vector = vector + vector
  i=a*b;                    // vector = vector * vector
  d=i*b;                    // vector = integer * vector
  cout << "a*(b=a): " << i << "\n"; // print out an integer
  cout << "     a: " << a;         // print out a vector
  cout << "    b=a: " << b;
  cout << "    a+b: " << c;
  cout << "(a*b)*b: " << d;
  cout << "A: \n" << A;       // print out a matrix
  B=i*A;                    // matrix = integer * matrix
  cout << "i*A:\n" << B;
  cout << "a*A: \n" << a*A;          // vector * matrix

  matrix C(VEC_SIZE,1),D(1,VEC_SIZE);
  C.make_column(0,a);       // assign vector a to 0th column of matrix C
  D[0] = a;                 // assign vector a to the 0th row of matrix D

  A = A*C*D;
  cout << "C: \n" << C;
  cout << "D: \n" << D;
  cout << "A = A*C*D:\n" << A;
}
```

FIG. 6. Example User Code—vector C

vector.dat - the input file

```
1 2 3 4 5

0 1 2 3 4
1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8
```

Results of executing vector.C on vector.dat

```
a*(b=a): 55
    a:   1   2   3   4   5
  b=a:   1   2   3   4   5
  a+b:   2   4   6   8  10
(a*b)*b:   55  110  165  220  275
A:
   0   1   2   3   4
   1   2   3   4   5
   2   3   4   5   6
   3   4   5   6   7
   4   5   6   7   8
i*A:
   0  55 110 165 220
  55 110 165 220 275
 110 165 220 275 330
 165 220 275 330 385
 220 275 330 385 440
a*A:
  40  55  70  85 100
C:
   1
   2
   3
   4
   5
D:
   1   2   3   4   5
A = A*C*D:
  40  80 120 160 200
  55 110 165 220 275
  70 140 210 280 350
  85 170 255 340 425
 100 200 300 400 500
```

FIG. 7. Input and Output Files for Example User Code

produce significant efficiency gains if the matrix object "knows" more elaborate properties about itself. By contrast, in FORTRAN the user code must specify which matrix algorithm to employ.

Compatibility between C++ and FORTRAN

The ability to incorporate preexisting FORTRAN routines into C++ libraries is highly desirable. FORTRAN libraries can be incorporated in the low levels of a C++ library (Floyd 1989), but should not just be thrown in randomly since much of the internal representation of the data is hidden (one of the major advantages of object-oriented programming). For example, the writer of the library could simply choose to call a FORTRAN routine for sparse matrix multiply, provided the internal data structures are set up accordingly, while the user would code "A*B" and rely on the library, not knowing that the library was not all written in C++. The communication problems here are roughly the same as linking a FORTRAN subroutine in C.

Ease of Use versus Efficiency

As previously noted, OOP enables one to design scientific codes in a more modular fashion than is practical to do with FORTRAN. It also allows for much more flexibility in the type of data structures used. A significant and compelling reason to use OOP, however, is the natural decomposition of the problem for parallelization. One goal is the development of a class structure that makes maximum use of inheritance for shared memory. For example, in a finite element code an element might reside entirely on a single processor of some parallel framework.

The most immediate gain evident in the use of OOP languages is in code

production and modification. No OOP language can currently outperform FORTRAN on the Cray, for example, but this is largely due to the amount of work already invested in vectorizing compilers for FORTRAN. To our knowledge, efforts to vectorize C++ are in their infancy.

More advanced tools such as object/class browsers and graphical debuggers are largely unavailable on supercomputers. Even if these tools were available, it makes more sense in terms of resource utilization to develop code on a workstation. Indeed, one of the primary benefits of OOP for large-scale scientific applications is that user code can be developed without worrying about the underlying data representation; the internal data representation is strongly tied to the underlying computer architecture.

Graphical Interfaces

The greatest success for OOP techniques has been in the field of graphics (Segal 1990). What might be more useful than having a better programming language, like C++, would be a comprehensive environment in which to program scientific codes. A graphical user interface (GUI) could handle such diverse tasks as managing inventories of objects and object hierarchies available, as well as the pre- and postprocessing of data.

Traditionally, the sequence of using an existing scientific code is as follows: data is preprocessed, a scientific code is run on this preprocessed data, and the results are postprocessed to be understandable by a human. The drawbacks of this sequence are that the writer of the user code must understand the form of the data sent from the preprocessor and to be sent to the postprocessor, as well as lengthened turnaround time between complete runs.

An object could have graphical input and viewing *methods* attached to it as easily as mathematical *methods*. This has advantages in terms of library development, modification, and use. A library developer can attach graphical *methods* to an object without disturbing any other *methods* and in a manner such that operation is conceptually independent. This allows for incremental development of the system as well as straightforward modification. The user need only know the name of the *methods* to call. At a higher level, even this amount of code could be automatically generated from a menu. This system could also allow for rapid turn around time between experiments.

Summary

This paper provides an introduction to OOP techniques and design, as well as a survey of the emerging literature in applying these techniques to the development of scientific codes. OOP allows for the design of more modular scientific codes than is practical to do with FORTRAN. Moreover, OOP also provides for the natural decomposition of a problem for implementation on massively parallel processor architectures.

Object-oriented programming techniques are mature enough to tackle serious scientific codes with a fair degree of efficiency compared with FORTRAN on serial machines. OOP techniques are particularly well suited to vector architectures as well as parallel architectures because of the parallelism inherent in an OOP viewpoint. We suggest that the appropriate segregation for the design of scientific codes is between library code and user code. OOP languages allow a great deal of data structure information

to be encapsulated, allowing for a much cleaner library/user distinction than is available in FORTRAN. Further, OOP techniques may be employed to realize the ideal of *efficient portability* of user code across widely different architectures.

Of the available OOP languages, C++ has some unique traits for use in scientific code development. An overriding design philosophy in the evolution of C++ has been "you don't pay an execution time price for a feature you don't use." This gives C++ a distinct edge in run-time efficiency over other leading OOP languages that don't share this design philosophy. In Ross et al. (1992), some simple scientific codes developed with OOP techniques and implemented in C++ are critically examined and the desirable and undesirable aspects of C++ as an OOP language are discussed.

Acknowledgments

The authors are grateful to the Air Force Phillips Laboratory for sponsoring this research under contract F29601-90-C-0046. The authors wish to thank Dr. Ian Angus, Boeing Computer Services, for his early discussions and data on this subject. We are also grateful to Drs. David Forslund and Stephen Pope of the Los Almos National Laboratory; Prof. John MacDonald, University of Washington; and Mr. Paul Morrow of the Phillips Laboratory for their comments and suggestions during the course of this research.

Appendix I. References

Angus, I. G., and Thompkins, W. T. (1989). "Data storage, concurrency, and portability: an object oriented approach to fluid mechanics." *Proc. 4th Conf. on Hypercubes, Concurrent Computing, and Applications*, Mar.

Booch, G. (1990). *Object-oriented design with applications*. Benjamin Cummings, Palo Alto, Calif.

Cox, B. J. (1986). *Object-oriented programming: an evolutionary approach*. Addison-Wesley, Reading, Mass.

Ellis, M. A., and Stroustrup, N. (1990). *The annotated C++ reference manual*. Addison-Wesley, Reading, Mass.

Floyd, M. (1989). "Making the C-to-Fortran connection." *Dr. Dobb's J.*, 14(154), 22–27.

Forslund, D., Wingate, C., Ford, P., Junkins, S., Jackson, J., and Pope, S. (1990). "Experiences in writing a distributed particle simulation code in C++. *Proc. 1990 Usenix C++ Conf.*, Usenix Association, 117–190.

Goldberg, A. (1984). *Smalltalk-80: the interactive programming environment*. Addison-Wesley, Reading, Mass.

Gorlen, K. E. (1987). "An object-oriented class library for C++ programs." *Software-Practice and Experience*, 17(12), 899–922.

Howard, H. C. (1991). "Project specific knowledge bases in AEC industry." *J. Comp. in Civ. Engrg.*, ASCE, 5(1), 25–41.

Keene, S. E. (1988). *Object-oriented programming in common LISP: a programmer's guide to CLOS*. Symbolics Press and Addison-Wesley, Reading, Mass.

Luger, G., and Stubblefield, W. (1989). *Artificial intelligence and the design of expert systems*. Benjamin Cummings, Redwood City, Calif.

MacDonald, J. A. (1988). "An outline of Arizona." *Proc. 20th Conf. Computer Sci. and Statistics*, American Statistics Society, 282–291.

MacDonald, J. A. (1989). "Object-oriented programming for linear algebra." *Proc. Object-Oriented Program Systems Language Applications SigPlan Notices*, Association for Computing Mechanics, 24, 175–184.

MacDonald, J., Stuetzle, W., and Buja, A. (1990). "Painting multiple views: and interactive technique for 'feeling' the shape of complex, multi-dimensional data." *Proc. Object-Oriented Program Systems Language Applications SigPlan Notices*, Association for Computing Mechanics, 25, 245–257.

Papert, S. (1980). *Mindstorms*, Basic Books, New York, N.Y.

Ross, T., Wagner, L., and Luger, G. (1992). "Object oriented programming for scientific codes: examples in C++." *J. Comp. in Civ. Engrg.*, ASCE, 6(4), 497–514.

Segal, M. (1990). "Programming language support for geometric computations," PhD thesis, University of California, Berkeley, Calif.

Vlissides, J. M., and Linton, M. A. (1988). "Applying object-oriented design to structured graphics." *Proc. USENIX C++ Conf.*, USENIX Association, Oct.

Wagner, L., Luger, G., and Ross, T. (1991). "Object-oriented programming in C++ on the cray for scientific codes." *Technical Report PL-TR-91-1037*, Phillips Lab., Kirtland Air Force Base, Albuquerque, New Mexico.