Parallel Object-Oriented Computation Applied to a Finite Element Problem

JON B. WEISSMAN[1], ANDREW S. GRIMSHAW[1], AND R. D. FERRARO[2]

[1]*Department of Computer Science, University of Virginia, Charlottesville, VA 22901, and* [2]*Jet Propulsion Laboratory/California Institute of Technology, 4800 Oak Grove Drive, Pasadena, CA 91109*

ABSTRACT

The conventional wisdom in the scientific computing community is that the best way to solve large-scale numerically intensive scientific problems on today's parallel MIMD computers is to use Fortran or C programmed in a data-parallel style using low-level message-passing primitives. This approach inevitably leads to nonportable codes and extensive development time, and restricts parallel programming to the domain of the expert programmer. We believe that these problems are not inherent to parallel computing but are the result of the programming tools used. We will show that comparable performance can be achieved with little effort if better tools that present higher level abstractions are used. The vehicle for our demonstration is a 2D electromagnetic finite element scattering code we have implemented in Mentat, an object-oriented parallel processing system. We briefly describe the application, Mentat, the implementation, and present performance results for both a Mentat and a hand-coded parallel Fortran version. © 1994 John Wiley & Sons, Inc.

1 INTRODUCTION

Developing scientific applications on current parallel computers is difficult due to the absence of suitable programming tools and models to manage the complex details of parallel programming. The majority of today's systems are programmed in an architecture-specific way using low-level message-passing primitives that are hard to use and lead to nonportable codes. These systems are typically programmed in Fortran or C in a data-parallel SPMD style. We believe that the problem lies not with the architectures, but with the tools that have been used to program them. We will show in this article that one can parallelize a real scientific application and obtain good performance with little effort if the right tools are used.

The tool that we have used is Mentat [1], an object-oriented parallel processing system developed at the University of Virginia. Using Mentat, the user is responsible for identifying object boundaries and specifying those object classes that have sufficient computational complexity to warrant parallel execution. The Mentat compiler and run-time system are responsible for managing all aspects of communication, synchronization, and scheduling for the user. Mentat performs tasks that humans perform poorly, whereas the programmer performs tasks (data and program decomposition) that compilers perform poorly.

Received April 1993

This work has been partially funded by grants NSF ASC-9201822, JPL-959303, and NASA NGT-50970.

Scientific Programming, Vol. 2, pp. 133–144 (1993)
CCC 1058-9244/94/040133-12

Thus, Mentat exploits the capabilities of both compilers and humans. Mentat is currently available on a range of platforms from networks of heterogeneous workstations to tightly coupled parallel machines such as the Intel iPSC/860. An important benefit of the Mentat approach is that applications developed on one platform are source code portable from one platform to another. This eliminates another problem common to writing software for parallel architectures, that applications are not portable across platforms.

The vehicle for our demonstration is a 2D electromagnetic finite element scattering code (EM). The application was chosen for three reasons: (1) it is a real, nontrivial, scientific code; (2) the sequential Fortran code was readily available; and (3) the application had previously been hand parallelized for a number of MIMD computers (Caltech/JPL Mark IIIfp Hypercube, Intel iPSC/860, and Intel Delta) using explicit message-passing primitives, providing us with the opportunity to compare the performance of hand-generated parallelism against our compiled Mentat version. The code computes the electric or magnetic field on an unstructured finite element mesh that defines the scattering objects as well as the space surrounding it.

Our work on the Mentat implementation of this code focuses on two issues: what is the overhead penalty that must be paid in order to use Mentat for this application, and how easy is it to apply Mentat to a scientific application like the finite element scattering code. What we have found is that the application domain mapped well to the object-oriented paradigm, and that the performance of the Mentat version is comparable to the hand-coded version. For the initial version of the Mentat implementation described here, the latter claim is only true for small numbers of processors. An optimized version of the Mentat implementation is also discussed. This version addresses the shortcomings in the initial version and has much better scaling properties. The results of the initial and optimized Mentat versions are presented.

This article is organized as follows. Section 2 discusses the EM application and finite element method (FEM). Section 3 provides an overview of Mentat. Section 4 discusses the object-oriented redesign of the EM application. Section 5 describes the parallel EM implementation via Mentat. Section 6 presents some preliminary results obtained with the Mentat version, and Section 7 provides a summary and future work.

2 THE EM PROBLEM

The FEM has been in use for many years in structural mechanics [2] and has become popular in recent years as a technique for use on EM problems [3]. FEM has the advantage of being able to deal with the specific geometry of objects by using unstructured gridding that follows an object's shape. This can be of particular importance in EM scattering problems, where the correct representation of a scatterer's surface is necessary for accurate computation. Finite elements are used in 2D and 3D EM scattering problems to model objects of complex composition. The "hand-coded" FEM code has been implemented on several MIMD computers, using explicit message passing. A complete description of this code, along with parallel implementation description and performance, is found [4]. For this work, we have concentrated on a 2D FEM problem.

The general scattering problem solved by the 2D EM code is illustrated in Figure 1. The code solves a Helmholtz equation for the electric or magnetic fields in the vicinity of a set of scatterers:

$$\nabla \cdot \left(\frac{\nabla E}{\mu}\right) + \frac{\omega^2}{c^2} \varepsilon E = 0 \qquad (1)$$

Here E is the electric field, and μ and ε are materials constants. An absorbing boundary condition on the boundary Γ uniquely specifies the problem. The FEM solves the equivalent "weak form" integral equation:

$$\int_\Omega d^2v \left[\frac{\nabla T \cdot \nabla E}{\mu} - \frac{\omega^2}{c^2} \varepsilon E T\right] = \int_\Gamma ds \frac{T}{\mu} \frac{\partial E}{\partial n} \qquad (2)$$

The boundary condition is incorporated into the surface integral on Γ.

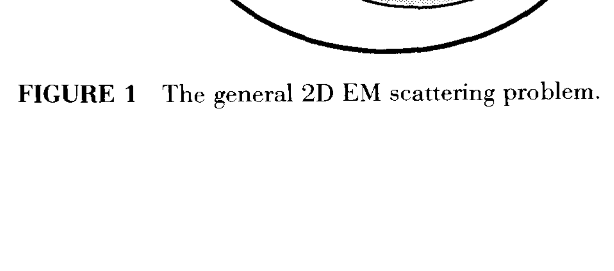

FIGURE 1 The general 2D EM scattering problem.

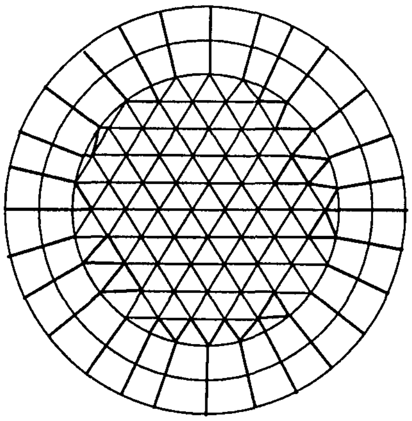

FIGURE 2 A simple finite element mesh.

The 2D integral equation is transformed into a set of linear equations by decomposing the problem domain into a set of finite elements. The problem domain Ω is meshed with nodal points at which the solution is to be found, matching the geometry of the objects. These nodes are then tiled with a set of finite elements as in Figure 2. In 2D, the elements might be triangles or quadrilaterals. A set of basis functions are defined at each node in the mesh, which have nonzero value only within the elements of which it is a part. These basis functions are generally some polynomial function, which is 1 at the node defining it, 0 at all other nodes in the element, and 0 along the edges of the element opposite the defining node. An example of a linear basis function at a node in a section of finite element mesh is given in Figure 3. The function is continuous inside and across elements, dropping to zero at the element edges, which do not intersect the node. On all other elements in the grid, the basis function is identically zero.

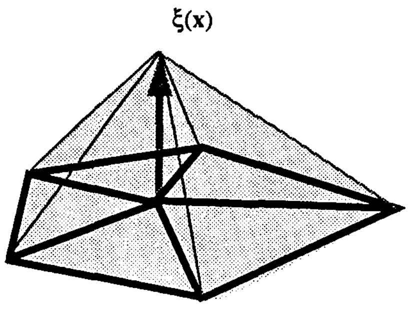

FIGURE 3 Node-based linear basis function.

The field quantities (electric or magnetic) may be expressed as a linear combination of these basis functions:

$$E(x) = \sum_i d_i \xi_i(\mathrm{x}) \tag{3}$$

where $\xi_i(\mathrm{x})$ is the basis function at the ith node, and $\mathrm{d_i}$ is its coefficient in the representation for E. Notice that because, by definition, all other basis functions are 0 at node i, the value of $\mathrm{d_i}$ is in fact the value of E at the ith node. We also write the test function T in terms of these basis functions. Substituting these into Equation (2), and recognizing that the test function T must be arbitrary, results in a matrix equation for the field coefficients $\mathrm{d_i}$:

$$\overline{\mathrm{K}} \cdot \mathrm{d} = \mathrm{F} \tag{4}$$

where d is the vector of field coefficients $\mathrm{d_i}$ from Equation (3), and $\overline{\mathrm{K}}$ and F, known as the stiffness matrix and force vector, respectively, are given by expressions involving integrals of individual basis functions or products of basis functions. Because all basis functions are localized to a handful of finite elements, these integrals are nonzero only for those elements that contain the basis functions involved. This results in a $\overline{\mathrm{K}}$ matrix, which is quite sparse. As a matter of practice, these integrals are computed on an element by element basis, with each element's contribution to $\overline{\mathrm{K}}$ and F added in its turn. In this manner, the complexity of integrating over a domain of irregular geometries is reduced to integrating over a set of regular finite sized elements.

This is the basic FEM. The EM finite element application consists of two primary computation phases: (1) matrix assembly and (2) matrix solve. In matrix assembly, the finite elements compute contributions (i.e., matrix values) that are assembled (i.e., added) into the stiffness matrix $\overline{\mathrm{K}}$. The stiffness matrix is banded and symmetric, in addition to being very sparse. During matrix assembly, the force vector F is also computed by the elements. This vector becomes the right-hand vector during the matrix solve computation.

In matrix solve, the system of equations represented by the stiffness matrix with the force vector as the right-hand side, is solved by a conjugate-gradient algorithm known as the bi-conjugate gradient (BCG) [5]. The algorithm requires three basic operations: matrix-vector multiplication, vector dot product, and vector saxpy. The solve phase poses challenges to achieving good performance on parallel machines due to the sparse nature of the matrix-vector operations.

3 MENTAT OVERVIEW

Mentat is a parallel object-oriented programming environment developed at the University of Virginia. Mentat was designed to address two problems that plague programming parallel MIMD architectures. First, writing parallel programs by hand is very difficult. The programmer must manage communication, synchronization, and scheduling. The burden of correctly managing the environment often overwhelms programmers, and requires a considerable investment of time and energy. Second, once implemented on a particular MIMD architecture, the resulting codes are usually not portable. Thus, considerable effort must be reinvested to port the application to a new architecture.

Mentat offers a solution to these problems by providing: (1) easy-to-use parallelism; (2) high performance via parallel execution; and (3) application portability across a range of platforms. The premise underlying Mentat is that writing programs for parallel machines does not have to be hard. Instead, it is the lack of appropriate abstractions that has made parallel architectures difficult to program, and hence, inaccessible to mainstream, production system programmers.

The Mentat approach exploits the object-oriented paradigm to provide high-level abstractions that mask the complex aspects of parallel programming, communication, synchronization, and scheduling from the programmer. Instead of worrying about and managing these details, the programmer is free to concentrate on the details of the application. The programmer uses application domain knowledge to specify those object classes (Mentat classes) that are of sufficient computational complexity to warrant parallel execution. The remaining complex tasks are handled by Mentat.

There are two primary components of Mentat: the Mentat Programming Language (MPL) [6] and the Mentat run-time system [7]. MPL is an object-oriented programming language based on C++ [8] that masks the complexity of the parallel environment from the programmer. Mentat classes consist of contained objects (local and member variables), their procedures, and a thread of control. Instances of Mentat classes, known as Mentat objects, are the computation grains. Because Mentat is based on a layered virtual machine model, and each layer introduces some amount of overhead, Mentat classes must be medium-to-large-grained to mask these overheads.

Mentat classes are denoted by the inclusion of the keyword "mentat" in the class definition, as in the mentat class `sparse_worker` shown below. Mentat classes may be defined as either persistent or regular.

```
persistent mentat class sparse_worker {
// private data and function members
public:
    complexvec* m_vec_mult
    (complexvec* vec);
    ...
};
```

Instances of regular Mentat classes are logically stateless, thus the implementation may create a new instance to handle every member function invocation. Persistent Mentat classes maintain state information between member function invocations. This is an advantage for operations that require large amounts of data, or that require persistent semantics. Instances of Mentat classes are used exactly like C++ classes, as in the fragment below. One difference is that persistent Mentat objects are instantiated by the *create* command.

```
sparse_worker worker;
...
worker.create ();
result = worker.m_vec_mul (rhs_vec);
```

Mentat supports a notion of parallelism encapsulation. Parallelism encapsulation takes two forms that we call intra-object and inter-object encapsulation. Intra-object encapsulation of parallelism means that callers of a Mentat object member function are unaware of whether the implementation of a member function is sequential or parallel. Inter-object encapsulation of parallelism means that programmers of code fragments (e.g., a Mentat object member function) need not concern themselves with the parallel execution opportunities between the different Mentat object member functions they invoke. Thus, the data and control dependencies between Mentat class instances involved in invocation, communication, and synchronization are automatically detected and managed by the compiler and run-time system without further programmer intervention.

The computation model underlying Mentat is the macro data flow model [7], a large-grain, graph-based, data-driven computation model. The Mentat run-time system supports the macro data flow model via the provision of a virtual macro data flow machine. Because the compiler uses a virtual machine model, porting applications to a new architecture does not require any user source level changes. Once the virtual machine has been ported, user applications are recompiled and can execute immediately.

Mentat runs on Sun 3s, Sun 4s, the Intel iPSC/2 and iPSC/860, and the Silicon Graphics Iris. We are currently porting Mentat to the IBM RS-6000, TMC CM-5, and the Intel Paragon. Performance results on a range of applications are available, and are quite encouraging [9].

4 OBJECT-ORIENTED EM DESIGN

Converting the sequential Fortran EM code into a parallel object-oriented code via Mentat requires two major steps: (1) porting the existing code to a sequential object-oriented language (C++) and (2) porting the C++ implementation to Mentat. Both pose different challenges. The Mentat conversion will be discussed in the next section. The C++ conversion requires a redesign or "paradigm shift" from the Fortran domain to the object-oriented domain. This is a nontrivial conversion even before opportunities for parallelism are considered. The use of global structures, aliasing, and the lack of data abstraction in Fortran made the transition to C++ challenging.

4.1 The Approach

There are several approaches that can be taken to convert an existing Fortran implementation to C++, and of these we considered two candidates: (1) reuse the Fortran artifact and wrap C++ classes around the existing code and (2) redo everything in C++. Although we operated on a very short time budget, we believed that option 2 was clearly the best choice because this affords us the greatest flexibility in experimenting with different problem decompositions. We also believe that a "pure" object-oriented system is easier to extend than a mixed-language implementation. Because this work is part of a continuing research effort, this is an important factor. The downside is a performance loss experienced in the sequential object-oriented code due to disparities in current compiler technology and the quality of existing Fortran numeric libraries. We expect this gap to close in the future, however.

Our approach then was to treat the Fortran code as a functional specification of the behavior of the EM application at an algorithmic level. For example, the element computation implementation in C++ mirrored the Fortran. The Fortran code was used as a guide or reference for low-level details and the mathematics. At the high level, we selected a class-object hierarchy that reflected our knowledge of the problem domain. Once a natural class structure was established, we inspected the Fortran code for algorithmic details that were necessary to faithfully reimplement the numeric computations encapsulated in the C++ member functions.

During this conversion the main difficulty was the lack of data abstraction in the Fortran code—this is not surprising because Fortran does not support this notion. The data structures were typically represented as collections of separate arrays. Data that were logically connected had to be inferred by its use or by the code comments. An example of this was the absence of an explicit representation for the elements themselves. The implicit element representation was scattered across numerous global arrays. The object-oriented approach requires just the opposite: that logically connected data be represented together and encapsulated. Another difficulty with the conversion was the use of complex numbers extensively in the Fortran code. We had to implement a fairly extensive complex class in C++. Our implementation was less efficient than the built-in, optimized, complex data type provided by Fortran.

One of the research objectives of this work is to consider the effort involved in converting existing scientific applications to an object-oriented platform (C++) and then to Mentat. The port from the sequential Fortran implementation to a fully tested C++ code took two graduate students 6 weeks (about one man-month of effort). Part of this time was needed to gain familiarity with the problem domain, which was unfamiliar, and to review the details of Fortran. We feel that the short time frame validates our decision to implement a C++ version and also provides further evidence for the suitability of the object-oriented paradigm as applied to scientific applications like the EM problem. However, although this evidence suggests a good fit between the object-oriented paradigm and this particular problem domain, there are performance trade-offs. These are discussed in the final section.

4.2 C++ Classes

The heart of the sequential EM implementation is its decomposition into C++ classes. Some of the C++ classes will become parallel or Mentat classes in the parallel EM implementation. Discussion of Mentat classes is deferred until the next section. The problem domain can be broken down into two phases: element assembly and matrix solve. The class-object hierarchies reflect this decomposition.

The first phase involves the finite element computations needed to construct the sparse stiffness matrix and right-hand-side vector. During this phase, each element computes a contribution to the matrix. We represented the elements as C++ objects contained within a finite element class hierarchy, shown in Figure 4.

The hierarchy is rooted by the virtual base class element and the derived classes reflect the different types of finite elements that are used in EM problems. The element type depends both on the physical characteristics of the material (e.g., 2D/3D or triangle/quadrilateral), and on the way the element computes its matrix contribution (e.g., 3 pt/6 pt quadrature). A part of the C++ specification for the finite element hierarchy is shown in Figure 5.

The element representation is simply the nodal points that define its boundaries. The derived classes contain element-specific information, such as the basis functions, that are needed in the element computations. The element contributions are computed by *get_kf* and are assembled into the sparse stiffness matrix during the first phase. This sparse matrix is stored as a list of sparse vectors, and each row is represented by the sparse_vec class. The sparse matrix is a special class known as a Mentat class and this is discussed in the next section.

During the second phase of the computation, matrix solve, "dense" vectors of complex numbers are computed by the application. The representation of complexvec is a memory-contiguous variable-sized array of complex type. The complexvec class specification is given in Figure 6. Memory contiguity is important in the parallel domain for objects that are transported between address spaces (such as objects of type complexvec).

Both the sparse_vec and complexvec classes had been implemented previously and we were able to reuse them with slight modification (to use complex numbers). The C++ classes form the basis for the parallel EM design. The remaining C++ classes in the application have a dual role: these classes can be treated as C++ classes as in the sequential version or as Mentat classes in the parallel version. These are discussed next. In the next section we will also show how everything fits together in the parallel EM implementation.

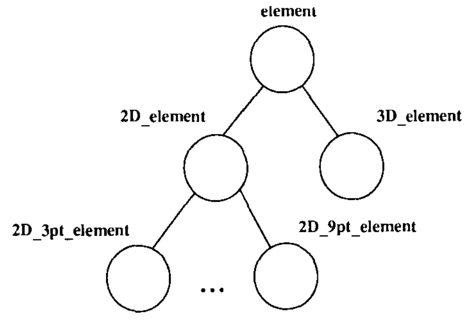

FIGURE 4 Finite element class hierarchy.

```
// Base class of the element hierarchy
class element {
    int *nodes; // nodal points
    int num_nodes;
    ...
public:
    // returns matrix and force-vector contributions
    virtual KF_contrib* get_kf ();
    element ();
};

// 3pt triangle 2D element
class 2D_3pt_element : 2D_element {
// basis fns
...
public:
    KF_contrib* get_kf ();
    2D_3pt_element (int *nodes, ...);
    ...
};
```

FIGURE 5 Finite element class specification.

```
class complexvec : public DD_array {
    int start_index, range;
    // memory continguous representation
    complex a[1];
public:
    complexvec (int cols);
    complexvec* saxpy(complexvec *f, complex& m);
    complexvec* ssxpy(complexvec *f, complex& m);
    complexvec* dot_product (complexvec* f);
    ...
};
```

FIGURE 6 Complexvec class specification.

5 MENTAT EM DESIGN

The design decisions that guided the transformation from the sequential Fortran implementation to the sequential C++ were motivated by three factors: (1) flexibility/extensibility, (2) fidelity, and (3) support for parallelization. Points 1 and 2 have been discussed and this section addresses point 3, transitioning to the parallel EM code.

The parallel EM code is based on the parallel object-oriented model of computation provided by Mentat. Although the design of the parallel EM code is concerned with points 1 and 2 above, it is driven by performance and scalability. In the parallel domain, the most critical factors affecting performance are computation granularity and load balance. In Mentat, computation granularity is specified via a mechanism known as Mentat classes, and load balance is achieved by an even partitioning of work across the instantiated Mentat objects.

The parallel EM system design based on Mentat is illustrated in Figure 7. The assembly (phase 1) and solve computations (phase 2) are shown. The remainder of this section will describe the Mentat classes used, the rationale for choosing them, and other important details of the parallel EM design and Mentat implementation.

The selection of Mentat classes is based on exploiting opportunities for parallelism and achieving an acceptable computation granularity given Mentat overheads and the characteristics of the target architecture. Our target is the Intel iPSC/860, a very unbalanced machine in which communication costs dominate computation costs by several orders of magnitude. The Mentat classes will need to be "computationally heavy", i.e., large-grained, to achieve reasonable performance given these factors. The EM application performs two main computations, element assembly and matrix solve, and these will be implemented via Mentat classes.

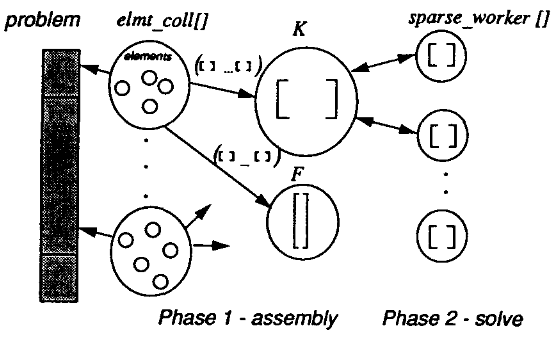

FIGURE 7 Parallel EM architecture.

```
persistent mentat class elmt_coll {
    element** elements;
    int element_num;
public:
    // element setup and partitioning
    void initialize (string *f, int i, int num_coll);

    // compute and assemble all elements
    void assemble (sparse_matrix *K, svector *F);
    ...
};
```

FIGURE 8 Mentat class elmt_coll specification.

For element assembly, there are many opportunities for parallelism because the element computations are independent and may proceed in parallel. To exploit maximal parallelism, we would turn the finite element classes (of Fig. 4) into Mentat classes. However, a single element assembly computation is too fine grained for Mentat and this will lead to unacceptably poor performance. Instead, we define a Mentat class that computes the contributions for a collection of elements, `elmt_coll` (see Fig. 8). Notice that the `elmt_coll` class contains C++ objects (of type element) as part of its representation.

A number of `elmt_coll` objects are instantiated at runtime and each computes in parallel. Each `elmt_coll` is assigned enough elements to achieve an acceptable computation granularity. The number of `elmt_coll` objects instantiated and how the individual elements get assigned to a particular `elmt_coll` are discussed later. Once the `elmt_colls` compute the matrix contributions and right-hand-side force-vector values associated with their contained elements (via *get_kf*), these values must be assembled into the stiffness matrix and force-vector, respectively. The *assemble* member function defined on `elmt_coll` initiates the element computations and invokes an assemble operation on the matrix. The Mentat class `sparse_matrix` represents the stiffness matrix (see Fig. 9). Matrix assembly is performed via the member function *assemble* called by each `elmt_coll`. The definition of the Mentat class `svector`, the force-vector, is omitted.

Most of the computation time is spent in the matrix solve phase. The solve computation is performed by an iterative preconditioned BCG algorithm [5] implemented by the *solve* member function of the Mentat class `sparse_matrix`. Our implementation exploits the most profitable opportunity for parallelism in the algorithm, namely

```
persistent mentat class sparse_matrix {
    // sparse_matrix representation
    sparse_vec** matrix;
    int size;
    ...
    // sparse_worker information
    int num_workers;
    sparse_worker* workers;
    ...
public:
    // Each elmt_coll assembles to matrix
    void assemble (K_list* K_contrib);

    // Solve matrix equation using rhs vector F
    void solve (svector* F);

    // Set up matrix with number of workers
    void initialize (int num_workers);
    ...
};
```

FIGURE 9 Mentat class sparse_matrix specification.

```
persistent mentat class sparse_worker {
    // sparse_worker representation
    sparse_vec** my_rows;

    // region of global matrix stored by worker
    region my_reg;

    // partial result for matrix-vector multiply
    complexvec* result;
    ...
public:
    // Distributes rows to worker
    void initialize (sparse_vec_list* sparse_rows,...);

    // Sparse mvec multiplication
    complexvec* m_vec_mult (complexvec* vec);
    ...
};
```

FIGURE 10 Mentat class sparse_worker specification.

the sparse matrix-vector multiplications done in each iteration of the BCG algorithm.

Parallelizing the matrix-vector multiplications requires that another class Mentat class, sparse_worker, be defined (see Fig. 10). The sparse_worker class is responsible for performing matrix-vector multiplication on disjoint regions of the sparse matrix. A number of sparse_worker objects are instantiated at runtime, and the sparse_matrix is partitioned into row-contiguous regions and distributed to the sparse_worker objects. This is done via the sparse_worker member function *initialize*. Once the sparse_matrix has been distributed fully to the sparse_worker objects, the sparse_matrix object engages the sparse_workers in parallel matrix-vector multiply operations (via *m_vec_mult*) repeatedly during the solve phase. The sparse_workers are encapsulated within the sparse_matrix (Fig. 9) and this has performance implications as we will see.

The Mentat classes, sparse_matrix, elmt_coll, and sparse_worker, reflect the computationally intensive phases of the application and result in a granularity suitable both for Mentat and the target architecture. These classes also allow sufficient parallelism in the application to be exploited. One important advantage of Mentat is that the serial EM code requires only a few "ifdefs" to turn these Mentat classes into C++ classes (see Fig. 11)—under 20 lines of code are unique to either the serial or parallel version.

At runtime, the programmer specifies the number of elmt_coll objects for the assembly phase and the number of sparse_worker objects for the solve phase. The number of objects should match the total number of available processors assuming the granularity is sufficient. For large applications, this is usually the case. On the iPSC/860 under NX, only one object (i.e., process) may be placed on a processor. Because these phases are nonoverlapping (i.e., solve does not begin until assembly has been completed), the number of elmt_colls and sparse_workers will be the same.

Achieving acceptable performance in the parallel EM code depends upon good load balance. It is sufficient to load balance the assembly and solve phases separately because they are independent—only a synchronization between these phases is needed. The assembly load balance requires that the elmt_coll objects each perform about the same amount of computation. An even partition of the elements across the elmt_colls would seem to be an easy solution. However, the general EM problem will contain elements of different types—more complex elements require more computation to determine matrix contributions. A good load balance solution ensures that each elmt_coll has roughly the same number of elements of each type. As an approximation to this, our implementation randomizes the element input files and randomly assigns elements to the elmt_coll objects.

```
#ifdef Mentat
persistent mentat class sparse_worker {
#else
class sparse_worker {
#endif
// as above
...
};
```

FIGURE 11 Dual Mentat/C++ class specification using "ifdefs."

Similarly, load balancing the solve phase requires that the sparse_worker objects are evenly balanced for the matrix-vector multiplications. An even partition of the sparse_matrix across the sparse_workers may not lead to load balance because the matrix has nonuniform sparsity (i.e., the number of nonzeros per row differs) and only nonzero positions of the matrix will be multiplied by the sparse_workers. Instead, load balancing is achieved by ensuring that each sparse_worker has about the same number of nonzeros in the matrix region that it has been assigned. Note that the number of rows assigned to each sparse_worker will, in general, be different.

The initial parallel EM design has a number of flaws that limit the scalability and performance of the system. The most obvious, as seen clearly in Figure 7, is that a single sparse_matrix object is a bottleneck for both matrix assembly and matrix solve. During the solve phase, partial results from the matrix-vector multiplies are fanned into the sparse_matrix, thus creating a communication bottleneck. The problem is due to the encapsulation of the sparse_workers within the sparse_matrix. This is a classic problem with the object-oriented paradigm. The single sparse_matrix object also limits the size problems that can be run because the entire matrix is assembled in one address space before it is distributed to the sparse_workers. Furthermore, no attempt was made to parallelize the dot products that occur within the BCG loop. These dot products are a good source of parallelism, especially for large vectors. The optimized version of the parallel EM design has addressed all of these problems.

Once the C++ version of the EM code had been fully implemented and tested, the Mentat version (about 3,000 lines of code) took 2 weeks to complete. One of the major problems we had with Mentat on the iPSC/860 was the need to force arithmetic operands to be double-word aligned to get good performance on this machine. This required some low-level pointer code and was time-consuming to implement and test. The memory bottleneck imposed by the single sparse_matrix object did not allow our EM problem to fit on an 8-MB iPSC/860 node at ORNL, a 128-node machine. We eventually ran on a 16-MB/node iPSC/860 at Caltech. Fortunately, the Mentat system binaries ported smoothly to the Caltech machine—no recompilation of the Mentat system code was necessary.

6 PRELIMINARY RESULTS

The initial Mentat EM code was developed on an 8-node Intel iPSC/860 at JPL and run on a 64-node Intel iPSC/860 at Caltech. The data collected are from an EM application that consisted of 2,304 9 pt quadrilateral elements (9,313 nodes). This is considered a small problem. We computed speedups with respect to the sequential C++ EM code run on a single i860 node (see Fig. 12).

The results are divided into the several phases: (1) problem setup is the time taken for the elmt_colls to read the element files from CFS and create the element partitions, (2) assembly is the time taken for the elmt_colls to complete the matrix assembly operations, (3) assembly and distribute include the time to distribute the matrix to the workers, (4) solve is the time taken for the matrix solve operation, and (5) total is the total time taken by the application. We should reiterate that virtually no optimization of the initial Mentat version had been performed.

Our results are compared with a hand-coded optimized parallel Fortran EM implementation that has been in development for some time. We expected the performance to be worse than the hand-coded version, but how much? The results indicate that this is indeed the case, but speedups were achieved even though the problem was small and the given implementation limitations that have been discussed (see Fig. 13). Comparison of the Mentat and the hand-coded versions indicates that the Mentat implementation is competitive with the hand-coded version for small numbers of processors, but that performance does not scale well as the number of processors is increased. This is due to the limitations that have been discussed, namely the sparse_matrix bottleneck for assembly and matrix-vector communication, and the sequential dot products in the solver. It is not surprising that the initial Mentat version does not scale given the design.

The optimized Mentat version does not suffer from this problem and the comparative performance is presented in Figure 14. These results indicate that the optimized Mentat version is scaling in a manner similar to the hand-coded Fortran. The assembly phase scales identically to the hand-coded version while the solve phase scales almost as well. The slight discrepancy is probably due to Mentat overheads often seen for small problems. For larger problems this overhead is often amortized by the computation. We expect the

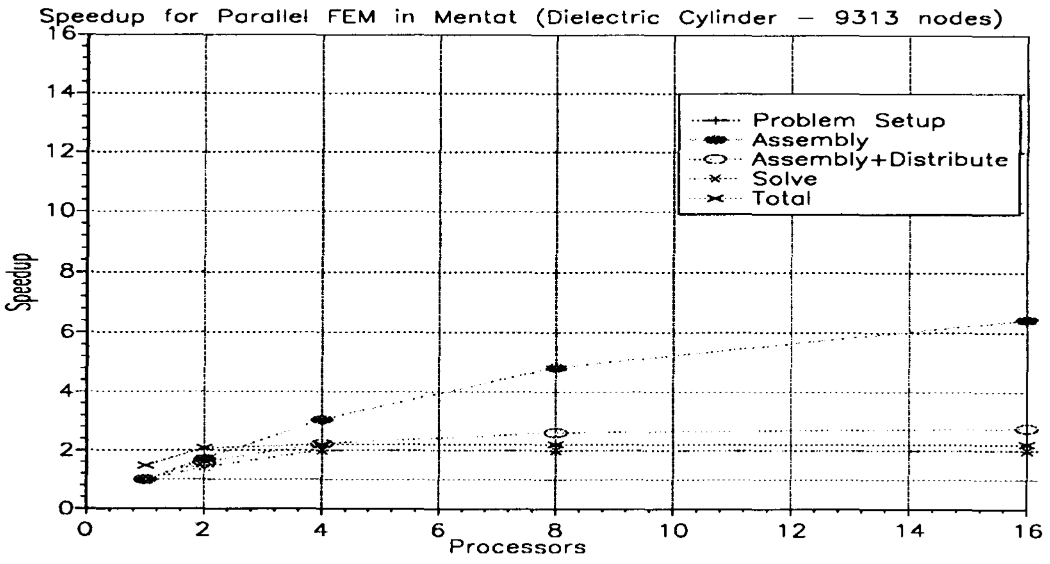

FIGURE 12 Parallel EM performance with initial Mentat version.

performance of the Mentat version to more closely match the hand-coded version for larger problems.

7 SUMMARY AND FUTURE WORK

The early results of the research are encouraging. The initial design and implementation of the parallel EM code using Mentat took less than 2 months, including the time to perform the paradigm shift from Fortran to C++. This indicates to us that the parallel object-oriented model in general, and Mentat in particular, is well suited to this problem domain. What we have found is that the EM problem has a natural representation in an object-oriented framework and performance is encouraging. We have also provided further evi-

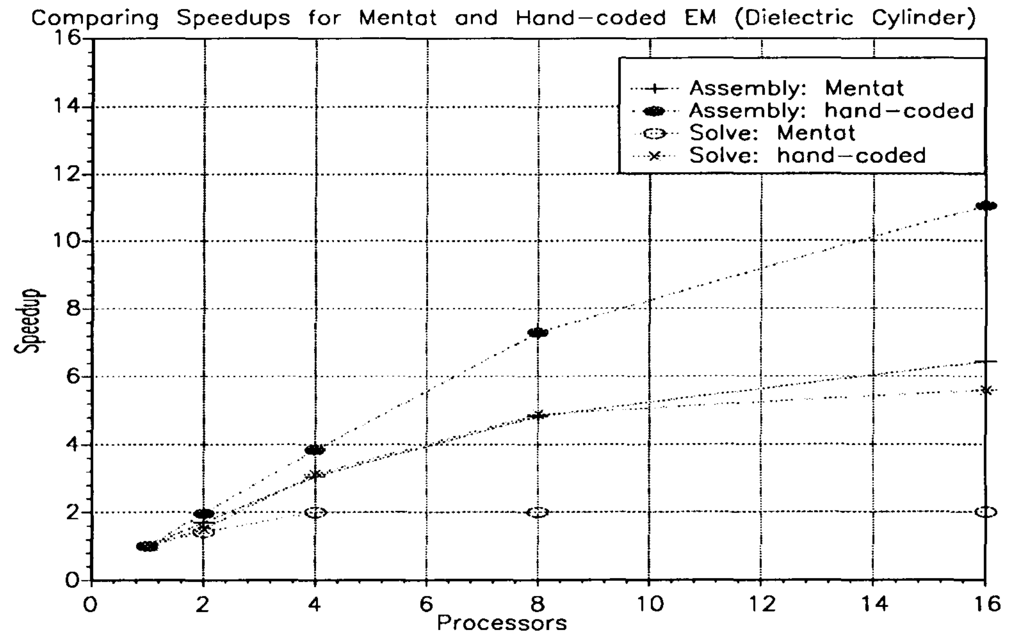

FIGURE 13 Comparison of initial Mentat version to hand-coded EM version.

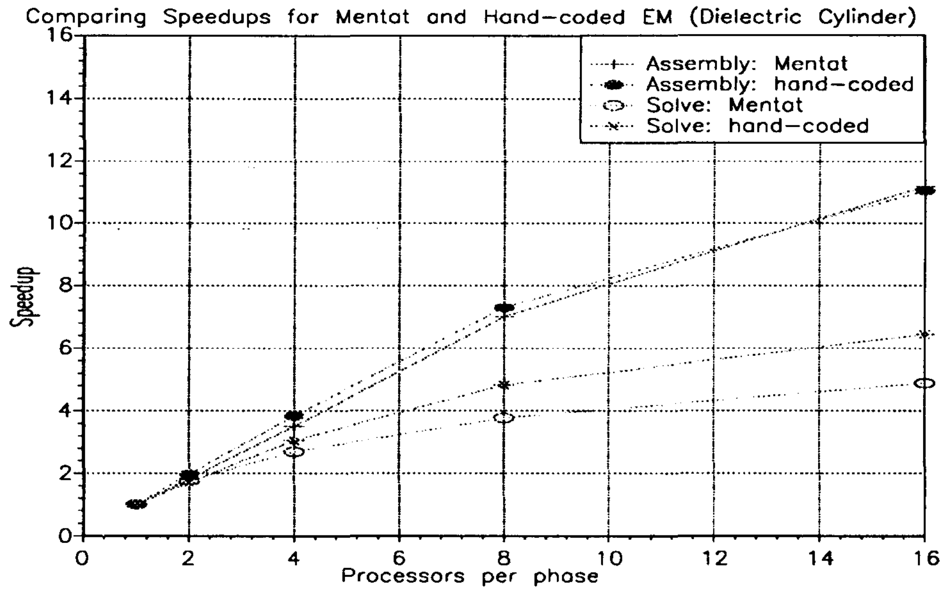

FIGURE 14 Comparison of optimized Mentat version to hand-coded EM version.

dence that Mentat is an easy-to-use programming environment for developing parallel object-oriented scientific applications.

Other researchers have begun to report on the experience of using object-oriented implementation techniques for scientific problems [10, 11]. Our experience is similar to Angus and Stolzy [10] in that programmer efficiency seems to be a more clear benefit than execution efficiency at present. They [10] report that the C++ performance is within an order of magnitude of the Fortran code (ours is within a factor of 2–3), and that this provides some hope.

Although the Mentat version has similar scaling properties to the hand-coded version, total elapsed times are not as good due to inefficiencies in the serial portions of the code. Some of this inefficiency can be attributed to superior numeric libraries and compiler optimizations in Fortran relative to C++. Future work addressing the serial bottlenecks is needed.

ACKNOWLEDGMENTS

Chiang Chang was responsible for implementing the sequential C++ version of the EM application. Part of the research described in this article was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government or the Jet Propulsion Laboratory, California Institute of Technology.

This research was performed in part using the Intel Gamma operated by Caltech on behalf of the Concurrent Supercomputing Consortium. Access to this facility was provided by the Jet Propulsion Laboratory.

REFERENCES

[1] A. S. Grimshaw, "Easy to use object-oriented parallel programming with Mentat," *IEEE Comput.*, pp. 39–51, 1993.

[2] T. J. R. Hughes, *The Finite Element Method.* Englewood Cliffs, NJ: Prentice-Hall, Inc., 1987.

[3] R. P. Silvestri and R. L. Ferrari, *Finite Elements for Electrical Engineers.* New York: Cambridge University Press, 1983.

[4] R. D. Ferraro, "Solving Partial Differential Equations for Electromagnetic Problems on Coarse-Grained Concurrent Computers," *Pier 7*, T. Cuik and J. Patterson, Eds., EMW Publishing, Cambridge, MA, 1993, pp. 111–155.

[5] D. A. H. Jacobs, *Sparse Matrices and Their Uses.* London: Academic Press, 1982.

[6] A. S. Grimshaw, E. Loyot Jr., and J. B. Weissman, *Mentat Programming Language (MPL) Reference Manual.* University of Virginia, Computer Science TR 91-32, 1991.

[7] A. S. Grimshaw, *Proceedings of the Fifth Distributed Memory Computing Conference.* Charleston, SC, 1990, pp. 9–12.

[8] B. Stroustrup, *The C++ Programming Language* (2nd ed). Reading, MA: Addison-Wesley, 1991.

[9] A. S. Grimshaw, W. T. Strayer, and P. Narayan, "Dynamic object-oriented parallel processing," *IEEE Parallel Distrib. Technol. Systems Appl.*, May 1993, pp. 33–47, 1993.

[10] I. G. Angus and J. L. Stolzy, *C++ at Work Conference,* 1991.

[11] D. W. Forslund, et al., *Usenix C++ Conference,* 1990.

[12] A. S. Grimshaw, E. A. West, and W. R. Pearson, "No pain and gain!—Experiences with Mentat on biological application," *Concurrency Practice Experience,* vol. 5, pp. 309–328, 1993.

[13] J. W. Parker, T. Cwik, R. D. Ferraro, P. C. Liewer, P. Lyster, and J. E. Patterson, *Proceedings of the Sixth Distributed Memory Computing Conference.* Portland, OR: IEEE Computer Society Press, 1991.