Parallel Object-Oriented (amputation Applied to a Finite Element Problem

Jon B. Weissman and Andrew S. Grimshaw
Department of Computer Science
University of Virginia

Robert I. Ferraro
Jet Propulsion Laboratory/California Institute of Technology

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 non-portable codes, extensive development time, and restricts parallel programming to the domain of the expert programmer. We believe these problems are not inherent to parallel computing but are the result of the 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 which 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.

1.0 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 vast majority of today’s systems are programmed in an architecture-specific way using low-level message-passing primitives that are both hard to use and lead to non-portable codes. These systems are typically programmed in Fortran or C in a data-parallel SPMD style. We believe that the problem is not the architectures, but the tools that have been used to program them. We will show in this paper 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, 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, while the pro-

1. This work has been partially funded by grants JPJ-959303 and NASA NGL-50970.
grammer performs tasks (data and program decomposition) that compilers perform poorly. ‘1’ bus, Mental exploits the capabilities of both compilers and humans. Mental is currently available on a range of platforms from networks of heterogeneous workstations to tightly coupled machines such as the InteliPSC/860. An important benefit of the Mental 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 (FEM). The application was chosen for three reasons: 1) it is a real, non-trivial, 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 III IIFP Hypercube, Intel PSC/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 Mental version. The code computes the electric or magnetic field on an unstructured finite element mesh which defines the scattering objects as well as the space surrounding it.

Our work on the Mental implementation of this code focuses on two issues: what is the overhead penalty which must be paid in order to use Mental for this application, and how easy is it to apply Mental 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 Mental version is comparable to the hand-coded version for small numbers of processors, but does not scale as well in this initial implementation.

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

2.0 The FEM Problem

The finite element method (FEM) has been in use for many years in structural mechanics [10] and has become popular in recent years as a technique for use on electromagnetic problems [13]. FEM has the advantage of being able to deal with the specific geometry of objects by using unstructured gridding which follows an object's shape. This can be of particular importance in FEM scattering problems, where the correct representation of a scatterer's surface is necessary for accurate computation. Finite elements are used in 2 and 3 dimensional electromagnetic 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 in [3]. For this work, we have concentrated on a 2D FEM problem.
The general scattering problem solved by the 2D FEM code is illustrated in Fig. 1. The code solves a

\begin{equation}
\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = \frac{1}{\rho^2} \frac{\partial u}{\partial t} + \frac{\partial u}{\partial r} \frac{\partial u}{\partial r} + \frac{\partial u}{\partial \theta} \frac{\partial u}{\partial \theta}
\end{equation}

subject to the boundary conditions:

\begin{equation}
\frac{\partial u}{\partial t} + \frac{\partial u}{\partial n} = 0 \quad \text{on} \quad \Gamma \setminus \Gamma_D
\end{equation}

where \( u \) is the electric field, \( \rho \) is the material density, and \( \Gamma \) and \( \Gamma_D \) are the boundaries and the Dirichlet boundary, respectively.
The field quantities (electric or magnetic) may be expressed as a linear combination of these basis functions:

\[ F(x) = \sum_i d_i \xi_i(x) \]  

(3)

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

\[ \mathbf{K} \cdot \mathbf{d} = \mathbf{F} \]  

(4)

where \( \mathbf{d} \) is the vector of field coefficients \( d_i \) from Eq. (3), and \( \mathbf{K} \) and \( \mathbf{F} \), known as the stiffness matrix and force vector, are given by expressions involving integrals of individual basis functions or products of basis functions. Since all basis functions are localized to a handful of finite elements, these integrals are non-zero only for those elements which contain the basis functions involved. This results in a \( \mathbf{K} \) 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 \( \mathbf{K} \) and \( \mathbf{F} \) 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 finite element method. The FEM 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 \( \mathbf{K} \). The stiffness matrix is banded and symmetric, in addition to being very sparse. During matrix assembly, the force vector \( \mathbf{F} \)
\( \mathbf{r} \) is also computed by the elements. This vector becomes the right-hand-side vector during the matrix solve compilation.

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 ILU-conjugate gradient\(^{[11]}\). 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.

### 3.0 Mentat overview

Mentrat is a parallel object-oriented programming environment developed at the University of Virginia. Mentrat 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 re-invested to port the application to a new architecture.

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

The Mentrat 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 (Mentrat classes) that are of sufficient computational complexity to warrant parallel execution. The remaining complex tasks are handled by Mentrat.

There are two primary components of Mentrat: the Mentrat Programming Language (MPL)\(^{[6]}\) and the Mentrat run-time system\(^{[7]}\). MPL is an object-oriented programming language based on C\(^{++}\)\(^{[14]}\) that masks the complexity of the parallel environment from the programmer. Mentrat classes consist of contained objects (local and member variables), their procedures, and a thread of control. Instances of Mentrat classes, known as Mentrat objects, are the computation grains. Because Mentrat is based on a layered virtual machine model, and each layer introduces some amount of overhead, Mentrat classes must be medium-to-large grained to mask these overheads.
Mental classes are denoted by the inclusion of the keyword "mental" in the class definition, as in the mental class sparse worker shown below. Mental classes may be defined as either persistent or regular,

```
persistent mental class sparse worker {
  // private data and function members
  public:
    complexvec* vec mult (complexvec* vec);
};
```

Instances of regular Mental classes are logically stateless, thus the implementation may create a new instance to handle every member function invocation. Persistent Mental 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 Mental classes are used exactly like C++ classes, as in the fragment below. One difference is that persistent Mental objects are instantiated by the `create` command.

Mental supports a notion of parallelism encapsulation. Parallelism encapsulation takes two forms that we call intra-object encapsulation and inter-object encapsulation. Intra-object encapsulation of parallelism means that callers of a Mental 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 Mental object member function) need not concern themselves with the parallel execution opportunities between the different Mental object member functions they invoke. Thus, the data and control dependencies between Mental 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 Mental is the macro data flow model[7], a large-grain, graph-based, data-driven computation model. The Mental 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 re-compiled and can execute immediately.

Mental runs on Sun 3s, Sun 4s, the Intel iPSC/2 and iPSC/860, and the Silicon Graphics Iris. We are currently porting Mental to the TMC CM-5 and the Intel Delta (Paragon). Performance results on a range of applications are available, and are quite encouraging [8].

4.0 Object-Oriented EMDeSign

Converting the sequential Fortran IEM code into a parallel object-oriented code via Mental requires two major steps: 1) porting the existing code to a sequential object-oriented language(C++) and 2) porting the
implementing a C++ version and also provides further evidence for the suitability of the object-oriented parts.

The initial implementation provided a first step towards the implementation of a C++ version and also provided further evidence for the suitability of the object-oriented parts. However, this initial implementation was limited in its functionality, and its integration into the existing system was challenging. The decision to implement a C++ version was driven by the need to improve performance, and the lack of data abstraction in the existing system necessitated a more sophisticated approach. The C++ implementation was designed to address these limitations, providing a more robust and scalable solution.

One of the challenges faced during this work was the need to consider the object-oriented approach in C++. Our initial implementation was less efficient, and the process required rethinking the approach to ensure compatibility with the existing system. The decision to implement a C++ version was driven by the need to improve performance and address the limitations of the initial implementation. The C++ implementation was designed to address these challenges, providing a more efficient and scalable solution.

Moreover, the C++ implementation was designed to be more efficient, and the lack of data abstraction in the existing system necessitated a more sophisticated approach. The C++ implementation was designed to address these limitations, providing a more robust and scalable solution. The decision to implement a C++ version was driven by the need to improve performance, and the lack of data abstraction in the existing system necessitated a more sophisticated approach. The C++ implementation was designed to address these challenges, providing a more efficient and scalable solution.

In conclusion, the C++ implementation was designed to address the limitations of the initial implementation, providing a more efficient and scalable solution. The decision to implement a C++ version was driven by the need to improve performance, and the lack of data abstraction in the existing system necessitated a more sophisticated approach. The C++ implementation was designed to address these challenges, providing a more robust and scalable solution.
digm as applied to scientific applications like the FEM problem. However, while 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 FEM implementation is its decomposition into C++ classes. Some of the C++ classes will become parallel or Mentat classes in the parallel FEM 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 Fig. 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 FEM 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. 3pt/6pt quadrature). A part of the C++ specification for the finite element hierarchy is shown in Fig. 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 `getKf` and are assembled into the sparse stiffness matrix during the first phase. This sparse matrix is stored as a list of sparse vectors, each row is represented by the
sparse vec class. The sparse matrix is a special class known as a Mentor 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 Fig. 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 implementation. 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 Mentor classes in the parallel version. These are discussed next. In the next section we will also show how everything fits together in the parallel F;M implementation.

5.0 Mentor 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
The parallel E:M code is based on the parallel I:O object-oriented model of computation provided by Mental. While the design of the parallel E:M 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 Mental, computation granularity is specified via a mechanism known as Mental classes, and load balance is achieved by an even partitioning of work across the instantiated Mental objects.

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

The selection of Mental classes is based upon exploiting opportunities for parallelism and achieving an acceptable computation granularity given Mental 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 Mental classes will need to be “computationally heavy”, i.e. large-grained (i.e., achieve reasonable performance given these factors. The E:M application performs two main computations, element assembly and matrix solve, and time will be implemented via Mental classes.

For element assembly, there are many opportunities for parallelism since 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 Mental 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 which computes the contributions for a collection of elements, `elm_coll`, see Fig. 8. Notice that the `elm_coll` class contains C++ objects (of type element) as part of its representation.

A number of `elm_coll` objects are instantiated at runtime and each computes in parallel. Each `elm_coll` is assigned enough elements to achieve an acceptable computation granularity. The number
of clmtColl objects instantiated and how the individual components assigned to a particular clmtColl are discussed later. Once the clmtColls 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 clmtColl initiates the element computations and invokes an assemble operation on the matrix. The Mental class sparse matrix represents the stiffness matrix, see Fig. 9. Matrix assembly is performed via the member function assemble called by each clmtColl. The definition of the Mental 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 Bi-conjugate gradient algorithm (BCG) [11] implemented by the solve member function of the Mental class sparse matrix. Our implementation exploits the most profitable opportunity for parallelism in the algorithm, namely the sparse matrix-vector multiplications done in each iteration of the BCG algorithm.

Parallelizing the matrix-vector multiplications requires that another class Mental 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
persistence mental 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. Mental class sparse_worker specification

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

Atrun time, the programmer specifies the number of clint_col 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. Since these phases are non-overlapping (i.e. solve does not begin until assembly has completed), the number of clint_col is and sparse workers will be the same.

Achieving acceptable performance in the parallel EIM code depends upon good load balance. It is sufficient to load balance the assembly and solve phases separately since they are independent - only the synchronization between these phases is needed. The assembly load balance requires that the clint_col objects
each perform about the same amount of computation. An even partition of the elements across the cmtcols would seem to be an easy solution. However, the general FEM 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 cmtcol 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 cmtcol objects.

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 since the matrix has non-uniform sparsity (i.e. the number of non-zeros per row differs) and only non-zero 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 non-zeros 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 FEM design has a number of flaws that limit the scalability and performance of the system. The most obvious as seen clearly in Fig.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 since 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. Nonetheless, the performance results are good when compared to a hand-coded version that does not suffer from these limitations. The results are presented in the next section.

Once the C++ version of the FEM code had been fully implemented and tested, the Mental version (about 3000 lines of code) took about two 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 perfor-
mance on Ibis 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 a 8MB iPSC/860 node at ORNL, a 128 node machine. We eventually ran on a 16MB/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.

6.0 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 2304 9pt quadrilateral elements (9313 nodes). This is considered a small problem. We computed speedups with respect to the sequential C++ EM code run on a single 860 node, see Fig. 12.

The results are divided into the several phases: 1) problem setup is the time taken for the loadmat files from C1-S and create the element partitions, 2) assembly is the time taken for the issue the problem setup is the time taken for the problem assembly and create the matrix assembly operations, 3) assembly and distribute includes the time to distribute the matrix out 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 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 doesn't 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.

7.0 Summary and Future Work

The early results of the research are encouraging. The initial design and implementation of the parallel EiM code using Mentat took under two 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. We have validated only part of this claim: that the EiM problem has a natural representation in a object-oriented framework and we have provided further evidence that Mentat is an easy-to-use programming environment for developing parallel object-oriented scientific applications. However, the other part of the claim is that performance does not suffer greatly with Mentat when compared to the hand-coded version. Until the limitations with the current Mentat design are removed, this claim cannot yet be made.

Other researchers have begun to report on the experience of using object-oriented implementation techniques for scientific problems [1,4]. Our experience is similar to [1] in that programmer efficiency seems to be a more clear benefit than execution efficiency at present. The authors [11] report that the C++ perfor-
mance is within an order of magnitude of the Fortran code (ours is within a factor of 2-3), and that this provides some hope.

While the performance was not as good as the hand-coded version, we have identified several performance enhancements that are being incorporated into the next parallel E&M design. The most important of these is the elimination of the sparse matrix bottleneck: the matrix will be assembled in a distributed fashion by the sparse workers directly. This will speed up assembly and eliminate the need to explicitly distribute the matrix out to the workers. We are also planning to distribute the dot products, parallelizing the inner products that occur in the BCG loop, thus eliminating the fan-in communication bottleneck to the sparse matrix. Instead, the sparse workers will interact in a more tightly-coupled fashion. Some encapsulation will be traded for performance. These changes will allow us to run much larger problems, and should yield a noticeable improvement in performance and scalability.

Acknowledgments

Chiang Chang was responsible for implementing the sequential C++ version of the E&M application. Part of the research described in this paper 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.

8.0 References


