Contemporary Mathematics
Volume 00, 0000

A Compiler for Parallel Finite Element Methods with Domain-Decomposed Unstructured Meshes

JONATHAN RICHARD SHEWCHUK AND OMAR GHATTAS

ABSTRACT. Archimedes is an automated system for finite element methods on unstructured meshes using distributed memory supercomputers. Its components include a mesh generator, a mesh partitioner, and a data-parallel compiler whose input is C augmented with machine-independent operations for finite element computations, and whose output is parallel code for a particular multicomputer. We describe an elegant implementation of domain decomposition and give preliminary performance results.

1. Introduction

Data-parallel languages such as High Performance Fortran make it possible to quickly and portably program multiprocessors. However, most current compilers are not satisfactory for programming finite element simulations, because they cannot support complicated parallel data structures.

There are several reasons why effective parallel compilers for finite elements are difficult to construct. If unstructured meshes are desired, the finite element code must use indirect addressing to process elements and to form stiffness matrices; but parallel indirect addressing is difficult. Communication costs will be high unless data structures are intelligently divided among processors. Furthermore, few data-parallel compilers provide explicit support for performing operations on a processor-by-processor basis; this makes it impossible to use domain decomposition methods to explicitly manage parallelism.

To address these problems, we are developing Archimedes, a system that generates finite element code for distributed memory supercomputers. The structure of Archimedes is diagrammed in Fig. 1. Its components include a mesh generator, a mesh partitioner, placement and routing heuristics, and a compiler.

The mesh generator uses an algorithm due to Ruppert [**4**] to create quality two-dimensional meshes on complex straight-line domains, and can also refine meshes based on *a posteriori* error estimates. An example of a mesh generated and refined this way is illustrated in Fig. 2.

Meshes are partitioned by a geometric algorithm due to Miller, Teng, Thurston, and Vavasis [**3**]. The partitioner serves three purposes. It divides a mesh into subdomains, to be mapped to separate processors (Fig. 3). It generates a nested dissection ordering on each subdomain, and thereby improves the performance of domain decomposition methods. Finally, a lesser known fact is that one

1991 *Mathematics Subject Classification.* Primary 65Y05, 65M55; Secondary 68N20, 65F10.

The first author was supported in part by the Natural Sciences and Engineering Research Council of Canada.

This paper is in final form and no version of it will be submitted for publication elsewhere.

©0000 American Mathematical Society
0000-0000/00 $1.00 + $.25 per page

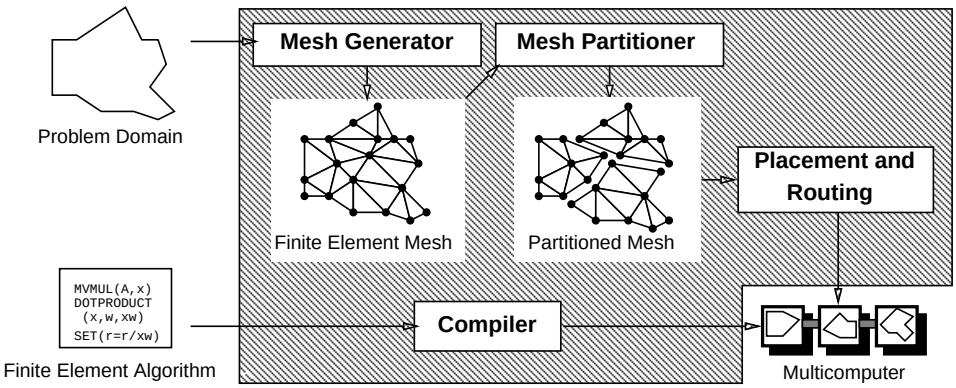

FIGURE 1. Structure of Archimedes.

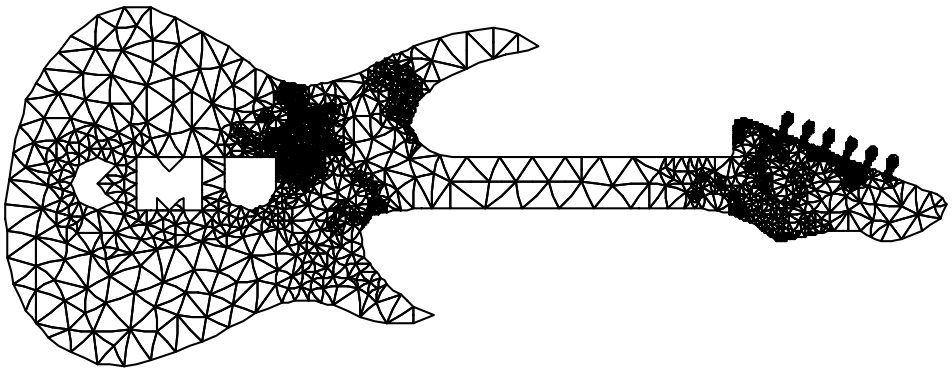

FIGURE 2. Refined mesh of an electric guitar.

can form a nested dissection ordering that improves memory cache performance because physically adjacent nodes tend to be grouped together in memory.

The communication graph for the partition of Fig. 3 is illustrated in Fig. 4. The nodes of this graph represent processors, and edges are drawn between any two processors having adjacent subdomains. On most multicomputers, communication is faster if adjacent subdomains are mapped to nearby processors. Hence, we use placement heuristics to find such a mapping. Some multiprocessors can be sped up by explicitly choosing communication routes between processors; routing heuristics are provided for these systems. The placement and routing heuristics are described in detail by Feldmann, Stricker, and Warfel [**2**].

Archimedes' compiler takes as input C code with special machine-independent operations for finite element computations, and outputs parallel code for a particular multicomputer. Users write parallel code without knowing the underlying communication mechanisms of the parallel architecture. This simplifies the task of writing parallel finite element code, or experimenting with iterative linear solvers. The remainder of this paper describes the parallel operations provided

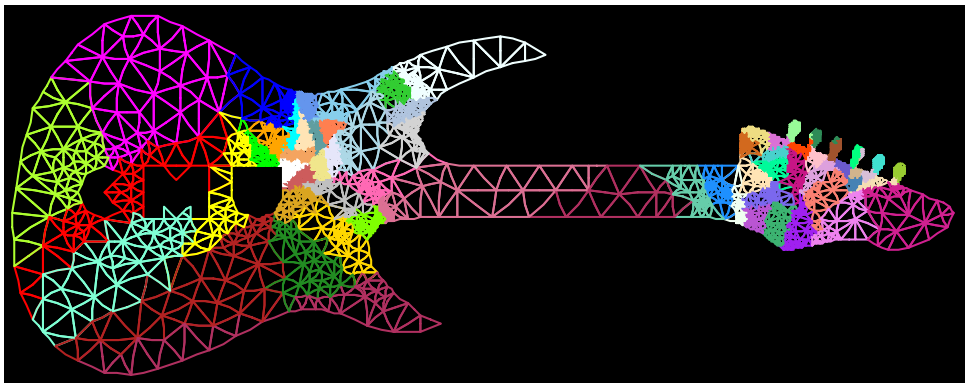

FIGURE 3. Partitioned electric guitar mesh.

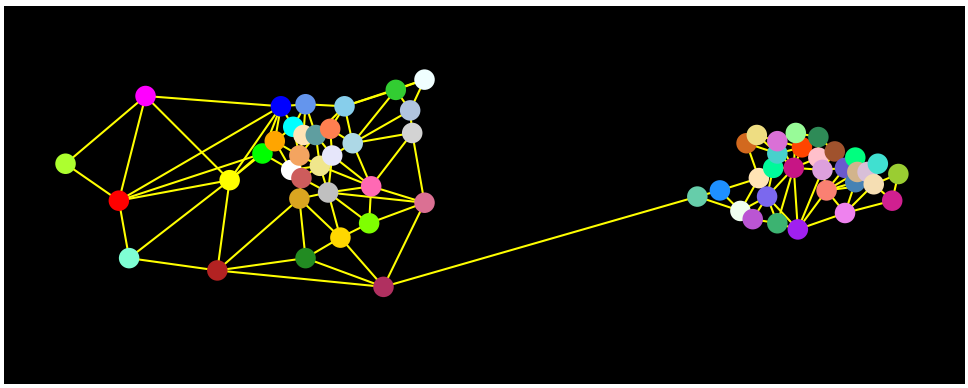

FIGURE 4. Communication graph of partitioned mesh.

by the compiler.

2. Parallel operations for domain decomposition

2.1. Data distribution and communicating operations. The data distribution of the stiffness matrix K is the key to our implementation. Let P be the number of processors. Each processor p holds a *processor stiffness matrix* K^p, which is a portion of the global stiffness matrix. Effectively, K^p contains zero rows and columns for each node not mapped to processor p; of course, these zeroes are not actually stored in memory. The value of the global stiffness matrix is $K = \sum_{i=1}^{P} K^i$.

Archimedes' partitioner maps each mesh element to only one processor, and K^p is defined by the set of elements mapped to processor p. Element stiffness matrices are assembled into processor stiffness matrices in parallel without communication, but the global stiffness matrix K is never actually formed. We say that K is *partially assembled*, because it is not assembled across processor

boundaries.

Here, our methodology is at variance with traditional data-parallel compilers. Nodes and edges on subdomain boundaries are shared by multiple processors. Accordingly, a distributed stiffness matrix may have storage allocated for an edge on several processors. ("Edge" here should be read to include self-edges, i.e., diagonal entries of the stiffness matrix.) Each processor stores the nonzero portion of its processor stiffness matrix in Compressed Sparse Row format.

Distributed vectors may have storage for a node allocated on several processors. Ordinarily, they are stored so that the duplicated nodes have duplicated values. In other words, a vector x is distributed so that each processor knows the elements of x corresponding to the nodes mapped to that processor. For reasons that will become clear in the next paragraph, we say that x is *fully assembled*.

Performing a distributed matrix-vector product of the form $y = Kx$ is a two-step process. In the first step, each processor p takes the product $y^p = K^p x$. This step uses a standard sequential sparse matrix-vector product, and requires no communication. At this point, we say that y, like K, is partially assembled, because the true value of y is $y = \sum_{i=1}^{P} y^i$. The second step is to *fully assemble* y. To accomplish this, each processor communicates with its neighbors (along the routes of the communication graph in Fig. 4) and sums each processor's value for each shared node. For example, if processors p and q share node j, then both processors will take the sum $y_j^p + y_j^q$ as the value of y_j. We call this step a *communicating sum*.

Many iterative methods for solving $Kx = y$ can be implemented with only two communication operations: communicating sums, and parallel reductions (such as dot product). Several local operations are also required, such as sparse matrix-vector multiply and elementwise vector operations. If the stiffness matrix is unsymmetric, our data distribution makes it trivial to obtain the transpose of the global stiffness matrix without communicating; hence, iterative methods such as biconjugate gradients (which requires the product $K^T x$) are easy to implement.

We can also use the communicating sum to form a diagonal preconditioner. Each processor extracts the diagonal of its processor stiffness matrix, and a communicating sum is used to find the diagonal of the global stiffness matrix. Thereafter, the diagonal can be used as a preconditioner without further communication.

2.2. Domain decomposition. We present a domain decomposition method appropriate for a sequence of linear problems having the same global stiffness matrix, as arise in time-dependent problems.

Order the variables so that those interior to subdomain 1 come first, followed by subdomain 2, etc. Last comes the set $\mathbb{I}$ of variables corresponding to interface nodes (each shared by two or more subdomains). The system $Kx = y$ has the form

$$\begin{bmatrix} K_{11} & 0 & \dots & 0 & K_{1\mathbb{I}} \\ 0 & K_{22} & & 0 & K_{2\mathbb{I}} \\ \vdots & & \ddots & & \vdots \\ 0 & 0 & & K_{PP} & K_{P\mathbb{I}} \\ K_{\mathbb{I}1} & K_{\mathbb{I}2} & \dots & K_{\mathbb{I}P} & K_{\mathbb{I}\mathbb{I}} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_P \\ x_{\mathbb{I}} \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ x_P \\ y_{\mathbb{I}} \end{bmatrix}.$$

A standard nonoverlapping domain decomposition technique is to use block elimination of $x_1, x_2, \dots, x_P$ to yield the Schur complement system $\widetilde{K}_{\mathbb{I}\mathbb{I}} x_{\mathbb{I}} = \widetilde{y}_{\mathbb{I}}$, where $\widetilde{K}_{\mathbb{I}\mathbb{I}} = K_{\mathbb{I}\mathbb{I}} - \sum_{i=1}^{P} K_{\mathbb{I}i} K_{ii}^{-1} K_{i\mathbb{I}}$ and $\widetilde{y}_{\mathbb{I}} = y_{\mathbb{I}} - \sum_{i=1}^{P} K_{\mathbb{I}i} K_{ii}^{-1} y_i$. This system is then solved by an iterative Krylov subspace method. Contrary to standard practice, we explicitly form the Schur complement matrix $\widetilde{K}_{\mathbb{I}\mathbb{I}}$.

If we ignore zero rows and columns, each processor stiffness matrix is of the form $K^p = \left[\begin{smallmatrix} K_{pp} & K_{p\mathbb{I}} \\ K_{\mathbb{I}p} & K_{\mathbb{I}\mathbb{I}}^p \end{smallmatrix}\right]$, where $K_{\mathbb{I}\mathbb{I}} = \sum_{i=1}^{P} K_{\mathbb{I}\mathbb{I}}^i$. By factoring K_{pp} (using a nested dissection ordering), each processor p forms (without communicating) a *processor Schur complement* $\widetilde{K}_{\mathbb{I}\mathbb{I}}^p = K_{\mathbb{I}\mathbb{I}}^p - K_{\mathbb{I}p} K_{pp}^{-1} K_{p\mathbb{I}}$. Afterward, the Schur complement is a partially assembled matrix, just like the stiffness matrix K — in other words, it has the property that $\widetilde{K}_{\mathbb{I}\mathbb{I}} = \sum_{i=1}^{P} \widetilde{K}_{\mathbb{I}\mathbb{I}}^i$. Hence, we can use $\widetilde{K}_{\mathbb{I}\mathbb{I}}$ in Krylov methods, with the same data distribution and communicating sum used for K.

Each processor Schur complement is a dense matrix coupling the boundary nodal unknowns of that processor's subdomain. This density is advantageous, because most modern microprocessors perform dense matrix-vector multiplication at two to ten times the speed of sparse matrix-vector multiplication, and because we can easily apply a Neumann-Neumann preconditioner, as described by Bourgat et al. [**1**]. By applying a communicating sum to $\widetilde{K}_{\mathbb{I}\mathbb{I}}^p$, each processor forms a fully assembled *processor preconditioner* $M_{\mathbb{I}\mathbb{I}}^p$. $M_{\mathbb{I}\mathbb{I}}^p$ is the submatrix of $\widetilde{K}_{\mathbb{I}\mathbb{I}}$ that represents only the boundary of subdomain p. Although $\widetilde{K}_{\mathbb{I}\mathbb{I}}^p$ may be singular, $M_{\mathbb{I}\mathbb{I}}^p$ generally is not, and each processor can easily factor or invert $M_{\mathbb{I}\mathbb{I}}^p$ (which is dense). Our inverse preconditioner (which approximates $\widetilde{K}_{\mathbb{I}\mathbb{I}}^{-1}$) is $M^{-1} = \sum_{i=1}^{P} (M_{\mathbb{I}\mathbb{I}}^i)^{-1}$. (For simplicity, we are abusing notation: each inverse is taken by ignoring zero rows and columns, which represent nodes not in the subdomain; but the summation takes these zero rows and columns into account. To be pedantically correct, the above sum should read $\sum_{i=1}^{P} (R^i)^T (R^i M_{\mathbb{I}\mathbb{I}}^i (R^i)^T)^{-1} R^i$, where R^p is a global-to-processor restriction matrix.) M^{-1} is a partially assembled matrix, and may be manipulated in the same fashion as K and $\widetilde{K}_{\mathbb{I}\mathbb{I}}$ (although the inverted matrices that compose M^{-1} need not be explicitly formed).

Below, we give our domain decomposition algorithm and the performance observed solving a heat conduction problem on a 64-processor iWarp. We use the mesh of Fig. 2, which has 8837 unknowns and employs quadratic triangular elements. For domain decomposition, an additional overhead of 0.6810 seconds is required to form the Schur complement matrix; this is quickly amortized if multiple right-hand sides must be solved.

(i) Each processor assembles its processor stiffness matrix K^p.
(ii) Each processor forms its (dense) processor Schur complement $\widetilde{K}^p_{\mathbb{I}\mathbb{I}}$.
(iii) With a communicating sum, each processor forms its (dense) processor preconditioner $M^p_{\mathbb{I}\mathbb{I}}$, which is then factored or inverted.
(iv) For each time step (or right-hand side):
    (a) Each processor assembles, element-by-element, its partially assembled force vector y^p.
    (b) From y^p and the factors of K^p, each processor forms its partially assembled reduced force vector $\widetilde{y}^p_{\mathbb{I}}$. A communicating sum is used to fully assemble the reduced force vector $\widetilde{y}_{\mathbb{I}}$.
    (c) The Schur complement system $\widetilde{K}_{\mathbb{I}\mathbb{I}} x_{\mathbb{I}} = \widetilde{y}_{\mathbb{I}}$ is solved iteratively. The values on the interface nodes ($x_{\mathbb{I}}$) are thus found.
    (d) Using triangular backsubstitution with the factors of K^p, each processor finds from $x_{\mathbb{I}}$ the values on its interior nodes (x_p).

Iterative Solution Time (after K^p, $\widetilde{K}^p_{\mathbb{I}\mathbb{I}}$, $M^p_{\mathbb{I}\mathbb{I}}$, and y^p are formed)		
Method	Iterations	Seconds
Conjugate Gradients	111	0.5104
+ Diagonal Preconditioner	91	0.4508
Domain Decomposition + CG (steps b–d)	42	0.1981
+ Diagonal Preconditioner	36	0.1841
+ Processor Preconditioner	12	0.1667

We recommend this approach because of its simplicity. By writing a communicating sum and parallel dot product, and using standard sparse matrix libraries, one can quickly implement an efficient domain decomposition solver.

References

1. J.-F. Bourgat, R. Glowinski, P. Le Tallec, and M. Vidrascu, *Variational formulation and algorithm for trace operator in domain decomposition calculations*, Second Int. Conf. Domain Decomposition Methods (T. Chan, R. Glowinski, J. Périaux, and O. Widlund, eds.), SIAM, 1989.
2. A. Feldmann, T.M. Stricker, and T.E. Warfel, *Supporting sets of arbitrary connections on iWarp through communication context switches*, Proc. 5th Annual ACM Symp. Parallel Algorithms and Architectures, 1993, pp. 203–212.
3. G.L. Miller, S.-H. Teng, W. Thurston, and S.A. Vavasis, *Automatic mesh partitioning*, Graph Theory and Sparse Matrix Computation (A. George, J.R. Gilbert, and J.W.H. Liu, eds.), Springer-Verlag, 1993.
4. J.M. Ruppert, *A Delaunay refinement algorithm for quality 2-dimensional mesh generation*, To appear in J. Algorithms, 1994.

School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, Pennsylvania 15213-3891
E-mail address: jrs@cs.cmu.edu

Department of Civil Engineering, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, Pennsylvania 15213-3891
E-mail address: oghattas@cs.cmu.edu