

SIFFEA: Scalable Integrated Framework for Finite Element Analysis

Xiangmin Jiao, Xiang-Yang Li, and Xiaosong Ma

Department of Computer Science
University of Illinois, Urbana, IL 61801, USA
{jiao,xli2,xma1}@uiuc.edu

Abstract. SIFFEA is an automated system for parallel finite element method (PFEM) with unstructured meshes on distributed memory machines. It synthesizes mesh generator, mesh partitioner, linear system assembler and solver, and adaptive mesh refiner. SIFFEA is an implicit parallel environment: The user need only specify the application model in serial semantics; all internal communications are transparent to the user. SIFFEA is designed based on the object-oriented philosophy, which enables easy extensibility, and a clear and simple user interface for PFEM.

1 Introduction

The parallel finite element method (PFEM) is widely used in the computational solution of system simulations in various engineering problems. Generally, there are six steps in the finite element analysis: mathematical modeling, geometric modeling, mesh generation, linear system formulation, numerical solution, and adaptive refinement. Many software packages are available on parallel machines for each individual component of FEM. However, there is a lack of automated integrated systems for PFEM, and a user must do a lot of application coding in integrating softwares from various sources.

To address this problem, we are developing SIFFEA, an integrated framework for FEM on scalable distributed memory machines. SIFFEA includes a mesh generator, a mesh distributor, a linear system assembler, an adaptive mesh refiner, and interfaces with mesh partitioners and linear system solvers. It also contains a novel user interface for specifying the mathematical model. SIFFEA is designed based on object-oriented philosophy, and written in C++. The communications are carried out using MPI for portability. Each component of SIFFEA is highly encapsulated, and the components interact with each other through well-defined interfaces, so that the implementation of one component can be changed without affecting the others.

Below we give a brief description of several classes in SIFFEA: `Mesh`, `MeshGenerator`, `MeshPartitioner`, `Solver`, and `MeshRefiner`. A `Mesh` object encapsulates all information about the mesh, including topological and geometrical data. A `MeshGenerator` object is activated by a `Mesh` object. It contains all functions related to mesh generation and will generate triangulations and fill them into the

Mesh object. A MeshPartitioner object contains the interfaces with mesh partitioning packages, and is activated by a Mesh object. A Solver object assembles and solves a linear system, and performs error analysis. A MeshRefiner object takes care of adaptive mesh refinement based on the results of error analysis.

Fig.3 is the interaction diagram of the system. It shows the timeline of objects' active periods and operations. The whole timeline can be roughly divided into four parts, shown as S1, S2, S3 and S4 at the left side of the diagram. They correspond to the four stages of the computation: mesh generation, mesh partitioning, linear system assembly and solution, and adaptive mesh refinement. Note that the lengths of objects shown in the diagram are not proportionally scaled to their real life time. The operations in the dashed-line rectangle, i.e., operations in S2, S3 and S4 are repeated until error requirements are satisfied. Notice that this diagram only provides a sequential view of the computations. In fact, except for MeshGenerator and MeshRefiner, each object exists on all the processors and its operations are carried out in parallel.

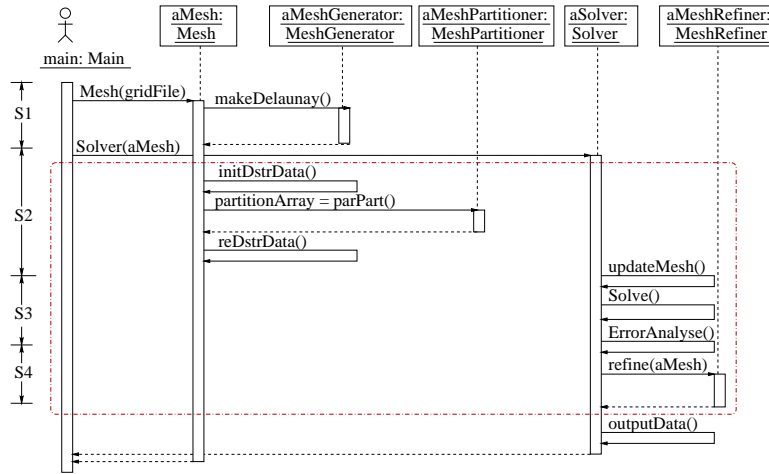


Fig. 1. Interaction diagram of SIFFEA.

SIFFEA provides a clear and simplified user interface. Up to date, the design of SIFFEA has been focusing on linear elliptic problems. For this kind of problems, the user only needs to provide a geometry specification, and a class encapsulating the serial subroutines for computing the element matrices and vectors, and the boundary conditions. All internal communications of SIFFEA are transparent to the user. We are looking into extending SIFFEA for time-dependent and nonlinear problems, while preserving a simple user interface.

The rest of the paper is organized as follows. Section 2 introduces the basic concepts about mesh generation and refinement, and their sequential and parallel algorithms. Section 3 describes the interface with the mesh partitioner, and the

algorithms for distributing the mesh onto processors after partitioning. Section 4 addresses the issues about user mathematical model specification and parallel linear system assembly. Section 5 discusses the basic concepts behind the error estimator and the adaptive mesh refiner. Section 6 presents some preliminary experimental results. Section 7 compares SIFFEA with some other PFEM systems. Section 8 concludes the paper with the discussion of future implementation and research directions.

2 Mesh Generation and Delaunay Refinement

Mesh generation is a fundamental problem in FEM. In particular, the *unstructured mesh* is widely used in FEM solvers, due to its advantages of varying local topology and spacing in reducing the problem size, and adapting to complex geometries and rapid changing solutions.

It is well-known that the numerical errors in FEM depend on the quality of the mesh. In particular, it is desirable that the angles of each element are not too small [2, 17]. The *Delaunay triangulation* and *Delaunay refinement* algorithms generate high quality meshes satisfying this criterion. Specifically, Delaunay triangulation maximizes the minimum angle among all elements [16]; Delaunay refinement allows one to add extra vertices, called *Steiner points*, in order to further improve the quality of the mesh [14].

SIFFEA contains a two-dimensional mesh generator, which can generate exact Delaunay triangulations, constrained Delaunay triangulations, and conforming Delaunay triangulations on complex domains. The conforming Delaunay triangulations guarantee no small angles, and are thus suitable for finite element analysis. The mesh generator also supports adaptive mesh refinement based on *a posteriori* error estimation, which will be discussed in Sec. 5.

The input to the mesh generator is a PSLG [10] description of the geometric domain, which can contain points and line segments. These line segments are called the *constrained edges*. The input domain can also contain holes, as shown in Fig. 2. A clockwise or counterclockwise sequence of edges of a polygon defines a hole. In particular, a clockwise sequence defines a hole inside the polygon, and a counterclockwise sequence defines a hole outside the polygon. Every hole has labels, which specify whether the hole belongs to the mesh, and the element size requirement inside the hole. Boundary conditions may be associated with the constrained edges. To achieve this goal, the user labels each constrained edge with an integer, which will be inherited by the Steiner points on it; then he/she defines a subroutine for computing the boundary condition based on the coordinates and the mark of a point. By convention, the user marks the uninterested edges with 0.

The mesh generation is designed using the object-oriented technology. We separate the mesh representation from the algorithms. Internally, the `Mesh` class uses the efficient quad-edge [8] data structure to represent the mesh. The algorithms, including Delaunay triangulation, Delaunay refinement, and adaptive

refinement, are designed using the high-level interfaces of the `Mesh` class. Each algorithm is encapsulated in a separate class.

The current mesh generator in SIFFEA is sequential, which is observed the performance bottleneck of PFEM. We are also currently designing and implementing a parallel mesh generation algorithm. The basic idea is as follows. Assume that there are p processors. We first apply the quadtree [6] based technique to add some Steiner points to the input point set. Then we apply some point separator [13] or the k - d tree [5], to divide the input point set and the Steiner points into p subdomains. Each processor is then assigned a subdomain, and performs the Delaunay triangulation and refinement on it. Note that the Delaunay refinement may add Steiner points on the separator edges between subdomains. Therefore, after each processor finishes triangulation and refinement locally, it must update its point set, by merging with the Steiner points on its adjoining separator edges from other processors; then it resumes the Delaunay refinement. This process repeats until no new points are added. Notice that the Steiner points added by the quadtree technique roughly make the points set well-spaced, thus it reduces the number of Steiner points added on the separator edges later on and hence reduce communication.

3 Mesh Partition and Distribution

After mesh generation, the mesh must be distributed across processors, so that in the subsequent steps, the linear system can be assembled and solved in parallel. In this section, we describe our approaches for mesh partition and distribution.

3.1 Mesh Partition

Mesh partition is to decompose the mesh into roughly equal size of subdomains while minimizing the cut-edges between the subdomains. Its purpose is to achieve better load balance and reduce communication overhead during linear system assembly and solve. Mesh partition is typically done using a graph partitioning algorithm. In our current implementation, we use ParMetis [9], which is a fast parallel graph partitioning algorithm, to carry out the work.

The ParMetis subroutine for graph partitioning requires the input of a distributed graph. In particular, the mesh nodal graph must be distributed across processors before calling ParMetis. If a sequential mesh generator is used, we accomplish this by assigning vertices in blocks to processors. That is, if n is the number of vertices, and p is the number of processors, then vertex i is assigned to processor $i / \lceil n/p \rceil$. Note that ParMetis only requires vertex adjacency list. To collocate the mesh and utilize memory efficiently, we distribute the vertex coordinates and the element connectivities along with the vertex adjacency list. The initial element partition is determined as follows: We assign an element to the owner of its second largest vertex. This simple heuristic has the advantage of not requiring extra communication to determine the element partition, and it yields fairly even distribution of elements in practice.

After the initial distribution, the ParMetis subroutine `ParMetisPartKWay()` is then called, which returns the new processor assignment of the locally owned vertices on each processor. This assignment will determine the partition of the global stiffness matrix, as discussed in Sec. 4.2. Note that an element partition is also needed by the linear system assembler. Again, the element partition can be derived from the vertex partition as above.

3.2 Data Redistribution

After determining vertex and element partitions, we must ship vertices and elements to their assigned processors. We refer to this stage as *data redistribution*. In particular, four types of data must be communicated at this stage, and they all have all-to-all communication patterns.

First of all, each processor must gather the vertices assigned to it, which includes gathering the global indices, the coordinates, and the adjacency list of each vertex. Note that the adjacency list will be used by memory management of the global matrix. Since a processor is generally assigned nonconsecutive vertices, to facilitate the subsequent steps, we compute a vertex renumbering such that the vertices on each processor are consecutive in the new number system.

Secondly, we ship the element connectivities to the assigned processors. Since the element assignments were based on the old number system, the connectivities are mapped to the new number system after shipped to the owner processors.

One or more vertices of a straddling element on the cut-edges are not local to the element's owner processor. These nonlocal vertices are called the *ghost points*, which are needed by the element matrix and vector computation. In the third step, each processor determines and gathers the ghost points.

Finally the boundary vertices and their marks are shipped to the processors. A boundary vertex must be sent to a processor, if it is owned or is a ghost point of that processor. Since the number of boundary vertices is generally small, a broadcast is used instead of vector scatters, to simplify communication pattern. After the broadcasting, each processor deletes the unneeded boundary vertices.

4 Linear System Assembly

The next step of solving FEM is to assemble and solve a system of linear equations. Our current implementation of SIFFEA has adopted a matrix-based approach. Namely, we assemble a global stiffness matrix and a global load vector from the element matrices and vectors, when solving the linear system. SIFFEA features a novel design for computing the element matrices and vectors and for assembling the global matrix. This design maximizes code reuse and reduces application code development time. This section presents the rationales behind our design. For the solution of the linear system, SIFFEA currently employs the linear solves in PETSc, which includes a variety of iterative solvers and rich functionality for managing matrices and vectors. For more information about PETSc, readers are referred to [4].

4.1 Element Matrix and Vector

Solving a boundary value problem using FEM includes two distinct transformations of its mathematical formulation: the differential equation is first transformed into an integral form (a.k.a. weak form), and then the integral form is transformed into a matrix and vector form. The first transformation demands knowledge about the application, and thus is best handled by the user. The second transformation, on the other hand, is application independent, but tedious and time consuming. Therefore, an ideal interface of a system for FEM should have the flexibility for the user to specify the integral form, but require minimum user's effort to transform the integral form into matrix form.

With the above goals, we choose the integral form as the mathematical modeling input of SIFFEA, and design tools for computing the element matrices and vectors from the integral form. We categorize the tasks of computing element matrices into three levels. The lowest level is the evaluation of values and derivatives of the shape functions at the Gauss points of an element. Since there are only a small numbers of element types that are widely used, we can develop an element library for these common elements. The second level is the numerical integration of a *term* in an integral form to compute a *partial element matrix*, where a term is a product of functions. The number of possible terms is also small, and hence an integrator library can be built on top of the element library. The highest level is the summation of partial matrices into an element matrix. The same categorization applies for computing element vectors. The user typically need write applications only in the highest level using the integrator library.

We now illustrate this idea through a concrete example. Assume the input integral form is

$$\int_{\Omega} (\nabla u \cdot \nabla v + a uv) dx dy = \int_{\Omega} f v dx dy,$$

where Ω denotes the domain, u and v are the base and test functions respectively, a is a constant, and f is a smooth function on Ω . We rewrite the left-hand side as a sum of two integrals, and we get

$$\int_{\Omega} \nabla u \cdot \nabla v dx dy + a \cdot \int_{\Omega} uv dx dy = \int_{\Omega} f v dx dy.$$

We can now use the integrators provided by SIFFEA to compute each integral.

There are three types of integrators. The first type integrates a term over an element and returns a partial element matrix, which are used to compute stiffness matrices. The second type also integrates a term over an element but returns a vector, used to compute load vectors. The last type integrates a term along marked boundaries of an element and returns a vector, used for applying boundary conditions, where the marks are provided by the mesh generator based on the users' input geometric specification. Each integrator type may have several functions, one for each valid term. The interface prototypes of these integrators are given as follows.

```

template <class Element> Matrix aMatIntegrator(const Element&, Fn*);
template <class Element> Vector aVecIntegrator(const Element&, Fn*);
template <class Element> Vector aBndIntegrator(const Element&, Fn*);

```

Note that these integrators are all template functions, because for a given term, the algorithms for computing its integral is the same for different elements. These integrators all have two input parameters: an object of type `Element`, which will be discussed shortly, and a function pointer. The function pointer points to a user function. A null pointer indicates an identity function.

Return to the above example. Let the integrators for the terms of the integral form be `integrate_dudv()`, `integrate_uv()`, and `integrate_v()` respectively. When translated into C++, the integrate form looks as follows.

```

template <class Element>
class ElmIntegralForm {
public:
    Matrix formElementMatrix(const Element& e) {
        return integrate_dudv(e, 0)+a*integrate_uv(e,0);
    }
    Vector formElementVector(const Element& e) {
        return integrate_v(e, f);
    }
    // Other functions, such as Dirichlet boundary conditions, and user function f
};

```

Readers can easily recognize an one-to-one correspondence between the terms in the integral form and in the application code. We encapsulate user's code in a parameterized class, which is consistent with the integrators. The choice of element type is then independent of mathematical specification, as it should be.

An `Element` class encapsulates the nodal coordinates, shape functions, and Gauss points of an element. Each type of finite element has a corresponding `Element` class, but all having the same interface. Note that only the linear system assembler need ever create instances of an `Element` class using the topological and geometrical data from the mesh data structure. In the member functions of the user class `ElmIntegralForm`, an `Element` object is passed in as a parameter. The user, however, need only pass the object to the integrators as a black box. The integrators then compute numerical integrations over the element through its public interfaces. An `Element` class contains public functions for computing the value and derivatives of the shape functions and the determinant of the Jacobian of the element map at the Gauss points, and for retrieving the weights at the Gauss points.

4.2 Global Matrix and Vector Assembly

Recall that in the global stiffness matrix, each row/column (or row/column block) corresponds to a vertex in the mesh, and there is a nonzero at row i

and column j if vertex i and j are adjacent in the mesh. It then follows that the global matrix is symmetric, and its nonzero pattern can be determined statically. SIFFEA employs the vertex adjacency information to preallocate memory space for the global matrix, to minimize the number of memory allocations.

We partition the global matrix onto processors by rows. In particular, a processor owns a row if it owns the corresponding vertices of that row. Linear system assembly works completely under the new vertex number system, so that each processor owns some consecutive rows of the global matrix. The global vector is also partitioned similarly.

To achieve the best scalability, the global matrix and vector are assembled on all processors concurrently. Each processor is in charge of constructing the element matrices and vectors of its locally owned elements, by calling the member function of user class `ElmIntegralForm`. Since all coordinates needed are already cached in the local memory, computing element matrices does not introduce extra communication. Subsequently, the element matrices and vectors are then assembled into the global matrix and vector with moderate amount of communication.

4.3 Boundary Condition Adjustment

After the element matrices are assembled into the global matrix, the global matrix is singular, and the boundary conditions must be applied. In the actual implementation, we adjust the boundary condition during matrix assembly for the best performance. For clarity, we discuss the boundary conditions separately.

The Neumann boundary conditions are handled in the integral form implicitly by the means of boundary integration. For Dirichlet conditions, we adopted a simple approach, in which each row corresponding to a Dirichlet boundary vertex is replaced by the boundary condition at that vertex. To preserve the symmetry of the global matrix, the off-diagonal nonzeros in the column corresponding to that vertex is also eliminated and moved to the right-hand side. This method has the advantages of not changing the number of equations, and hence simplifies the implementation. However, it suffers from a tiny amount of redundant computation. In terms of the user interface, a member function `formDirichletBC()` must be provided by the user in the class `ElmIntegralForm` to compute the boundary condition of a point based on its coordinates and mark.

5 Error Estimation and Adaptive Mesh Refinement

After solving the linear system in parallel, we obtain an initial numerical solution u . The initial mesh M does not guarantee a good solution, and hence it must be refined properly [3]. In this section, we consider the issues regarding to adaptive mesh refinement, and the algorithms for it.

A spacing function such as el_M and lfs_M [14] specifies how fine a mesh should be at a particular region. This function can be derived from the previous numerical results or from the local geometry feature. The spacing function for

a well-shaped mesh should be smooth in the sense that it changes slowly as a function of distance, i.e., it satisfies the α -Lipschitz condition [11].

Our spacing function is based on Li *et al.* [11]. First, a refine-coarsening factor δ_p of each point p is obtained from a *a posteriori* error analysis, by comparing the error vector e of the initial solution u with a scalar global error measurement ϵ . For example, if we choose e to be the magnitude of residual, and ϵ the norm of e , then δ_p can be defined as: (1) $(e_p/\epsilon)^{-1/k}$, if $l < e_p/\epsilon < L$; (2) $1/l^{1/k}$, if $e_p/\epsilon \leq l$; (3) $1/L^{1/k}$, if $e_p/\epsilon \geq L$; where k is the accuracy order of the elements, and $0 < l \leq 1 \leq L$. The new spacing function h is then obtained from δ_p and previous spacing [11].

The mesh is then adaptively refined according to h . We would like to use the structure of the current mesh M as much and as efficiently as possible. The first step of our algorithm is to compute a maximum spacing function f that satisfies the new spacing requirement h . Then, we generate a β -sphere-packing of the domain with respect to f using the following procedure [11].

Let $S_1 = \{B(p, f(p)/2) | p \in M\}$. Then sample points in every triangle of the mesh and let S_2 be the set of these spheres defined by them. The spheres in $S_1 \cup S_2$ are ordered as the following: First the spheres centered on the boundary; then, all other spheres in S_1 in increasing order of radii; then followed by all spheres in S_2 in increasing order of radii. The intersection relation of spheres defines a *Conflict Graph (CG)*. Let S be the set of spheres which form the *Lexical-First Maximal Independent Set (LFMIS)* [11] of *CG*. M' is then the Delaunay triangulation of the centers of the spheres in S . The generated mesh is well shaped and the size is within a constant factor of the optimal mesh [11].

We are currently developing a parallel adaptive mesh refinement algorithm, in which the same idea as the parallel mesh generation is to be used.

6 Experimental Study

The input geometry of our testing case is shown in Fig. 2. The testing mesh is finer than the one shown in Fig. 2, containing 193881 vertices, 535918 edges, and 342033 triangles. The minimum minimum angle of the triangles is 20.1° . The model being solved is a Laplace equation. The boundary conditions are set as follows. The innermost circle has no displacement, and the outermost circle has unit displacement in their normal directions. All the other boundaries have zero natural boundary condition. The computational solution of this sample problem is visualized in Fig. 3, which plots the displacement vectors of the mesh vertices.

We conducted some preliminary experiments on an Origin 2000 with 128 processors at NCSA, which is a distributed shared memory machine. Each processor is a 250 MHz R10000. The native implementation of MPI on Origin was used during the test. The sparse linear solver used was Conjugate Gradient method, where the absolute and relative error tolerances were both set at 10^{-8} . The preprocessing step is done in sequential, including Delaunay triangulation and refinement.

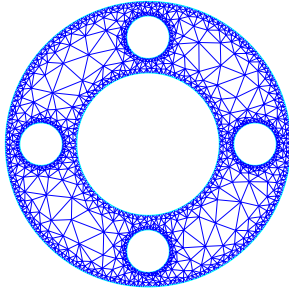


Fig. 2. The input geometry of the testing problem.

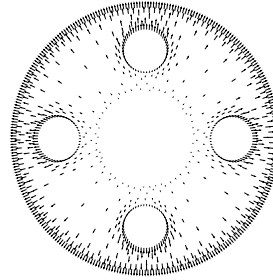


Fig. 3. Displacement vectors.

Table 1. Performance results on Origin 2000 at NCSA.

Time in seconds	1	2	4	8	16	32	64	128
Initial distribution	-	0.41	0.35	0.51	0.35	0.48	0.61	0.46
ParMetis (M)	-	2.0	0.94	0.68	0.84	2.7	23.5	59.0
Redistribution (R)	-	0.50	0.27	0.26	0.25	0.31	1.4	12.3
Linear assembler	12.9	10.7	7.2	5.2	4.5	3.9	2.2	1.4
Linear solver	269.6	186.2	83.8	47.0	17.1	16.6	6.8	5.5
Total	382.5	199.8	92.6	53.7	23.0	24.0	34.5	78.7
Speedup(w/ M,R)	-	1.9	4.1	7.1	16.6	15.9	11.1	4.9
Speedup(w/o M,R)	-	1.9	4.2	7.25	17.5	18.21	39.8	50.6

We present the execution times of data distribution, mesh partitioning, matrix assembly, and linear solver in Table 1. The results show that the execution times of data distributions, which involve intensive communication, do not change much with the number of processors. The mesh partitioner slows down after 32 processors, maybe due to the following two reasons: (1) The initial vertex distribution is essentially random, which causes large amount of communication on large number of processors; (2) the difficulty of the partitioning problem itself increases with the number of the processors. For the similar reasons, the data redistribution also has no speedups for more than 16 processors. The linear system assembler and the linear solver achieve steady speedups up to 128 processors. To isolate the slowdowns of ParMetis and data redistribution, we report the speedups of both with and without ParMetis and data redistribution. The results clearly demonstrate the necessity of parallel mesh generation: After parallel mesh generation, each processor will contain a mesh of a subdomain, which already gives a fairly good partition; then a cheaper partition refinement and data redistribution can be used.

7 Related Work

A lot of work has been done on PFEM in the past few years. Various approaches have been used by researchers. In this section, we summarize some features of related work and highlight their differences from SIFFEA.

Archimedes [15], developed at the CMU, is an integrated tool set for solving PDE problems on parallel computers. It employs algorithms for sequential mesh generation, partitioning, and unstructured parallel computation as a base for numerical solution of FEM. The central tool is Author, a data parallelizing compiler for unstructured finite element problems. The user writes a complete sequential FEM solver, including linear system assembly and solve, and then Author automatically parallelizes the solver. Archimedes is written in C.

The Prometheus library [1], is a parallel multigrid-based linear system solver for unstructured finite element problems developed at Berkeley. It takes a mesh distributed across multiple processors, and automatically generates all the coarse representations and operators for standard multigrid methods. Prometheus does not integrate mesh generation and adaptive mesh refinement.

There is also some work in the area of parallel adaptive mesh refinement. PYRAMID[12], developed at NASA and written in Fortran 90, uses the edge bisection to refine a mesh in parallel. To achieve load balancing, it estimates the size of the refined mesh and redistribute the mesh based on a weighted scheme before the actual refinement is performed. SUMMA3D[7] developed at ANL uses the *longest edge bisection method* to refine a mesh. It applies its own mesh partition algorithm after mesh refinement.

8 Conclusion and Future Work

We have described SIFFEA, an integrated framework for PFEM, which synthesizes several widely used components in FEM. SIFFEA features a novel design for specifying the mathematical model which maximizes code reuse and simplifies the user interface. As an implicit parallel environment, SIFFEA hides all internal communications from the user. The preliminary experimental results have achieved good overall speedups up to 32 processors, and reasonable speedups up to 128 processors for our data distribution and matrix assembly codes. We are underway of designing and implementing algorithms for parallel mesh generation and refinement, to achieve better overall scalability. We also plan to extend the framework to time-dependent and nonlinear problems, and to provide interfaces with multigrid and direct linear solvers.

Acknowledgments

This work started as a course project with Professor Laxmikant Kale, who has encouraged us to work on the problem continuously and to put it into publication. We thank Jiantao Zheng from the TAM Department of UIUC for helpful

discussions on theories of finite element methods. We thank the anonymous referees for their valuable comments. The authors are supported by the Center for Simulation of Advanced Rockets funded by the U.S. Department of Energy under Subcontract B341494.

References

1. ADAMS, M., DEMMEL, J. W., AND TAYLOR, R. L. Prometheus home page. <http://www.cs.berkeley.edu/~madams/Prometheus-1.0>, 1999.
2. BABUŠKA, I., AND AZIZ, A. K. On the angle condition in the finite element method. *SIAM J. Numer. Anal.* 13(2) (1976), 214–226.
3. BABUŠKA, I., ZIENKIEWICZ, O. C., GAGO, J., AND DE A. OLIVEIRA, E. R., Eds. *Accuracy Estimates and Adaptive Refinement in Finite Element Computations*. Wiley Series in Numerical Methods in Engineering. John Wiley, 1986.
4. BALAY, S., GROPP, W. D., MCINNES, L. C., AND SMITH, B. F. PETSc 2.0 users manual. Tech. Rep. ANL-95/11, Argonne National Laboratory, 1998.
5. BENTLEY, J. Multidimensional binary search trees used for associative searching. *Communication of the ACM* 18, 9 (1975).
6. BERN, M. W., EPPSTEIN, D., AND TENG, S.-H. Parallel construction of quadrees and quality triangulations. In *Algorithms and Data Structures, Third Workshop* (Montréal, Canada, 1993), F. K. H. A. Dehne, J.-R. Sack, N. Santoro, and S. Whitesides, Eds., vol. 709 of *Lecture Notes in Computer Science*, Springer, pp. 188–199.
7. FREITAG, L., OLLIVIER-GOOCH, C., JONES, M., AND PLASSMANN, P. SUMMA3D home page. <http://www-unix.mcs.anl.gov/~freitag/SC94demo/>, 1999.
8. GUIBAS, L. J., AND STOLFI, J. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Trans. Graphics* 4, 2 (1985), 74–123.
9. KARYPIS, G., AND KUMAR, V. Parallel multilevel k -way partitioning scheme for irregular graphs. TR 96-036, Computer Science Department, University of Minnesota, Minneapolis, MN 55454, 1996.
10. LI, X. Y., TENG, S. H., AND ÜNGÖR, A. Biting: advancing front meets sphere packing. *the International Journal of Numerical Methods in Engineering* (1999). to appear.
11. LI, X. Y., TENG, S. H., AND ÜNGÖR, A. Simultaneous refinement and coarsening: adaptive meshing with moving boundaries. *Journal of Engineering with Computers* (1999). to appear.
12. LOU, J., NORTON, C., DECYK, V., AND CWIK, T. Pyramid home page. <http://www-hpc.jpl.nasa.gov/APPS/AMR/>, 1999.
13. MILLER, G. L., TENG, S. H., THURSTON, W., AND VAVASIS, S. A. Geometric separators for finite element meshes. *SIAM J. Scientific Computing* 19, 2 (1998), 364–384.
14. RUPPERT, J. A Delaunay refinement algorithm for quality 2-dimensional mesh generation. *Journal of Algorithms* 18, 3 (1995), 548–585.
15. SHEWCHUK, J. R., AND O'HALLARON, D. R. Archimedes home page. <http://www.cs.cmu.edu/~quake/arch.html>, 1999.
16. SIBSON, R. Locally equiangular triangulations. *Computer Journal* 21 (1978), 243–245.
17. STRANG, G., AND FIX, G. J. *An Analysis of the Finite Element Method*. Prentice-Hall, 1973.