PARALLEL ADAPTIVE MESH REFINEMENT TECHNIQUES FOR PLASTICITY PROBLEMS*

WILLIAM J. BARRY, MARK T. JONES, AND PAUL E. PLASSMANN†

Abstract. Accurately modeling the nonlinear properties of materials can be computationally expensive. Parallel computing offers an attractive way for solving such problems; however, the efficient use of these systems requires the vertical integration of a number of very different software components. To investigate the practicality of solving large-scale, nonlinear problems on parallel computers, we explore the solution of two- and three-dimensional, small-strain plasticity problems. We consider a finite-element formulation of the problem with adaptive refinement of an unstructured mesh to accurately model plastic transition zones.

We present a framework for the parallel implementation of such complex algorithms. This framework, using libraries from the SUMAA3d project, allows a user to build a parallel finite-element application without writing any parallel code. To demonstrate the effectiveness of this approach on widely varying parallel architectures, we present experimental results from an IBM SP parallel computer and an ATM-connected network of Sun UltraSparc workstations. The results detail the parallel performance of the computational phases of the application during the process while the material is incrementally loaded.

1. Introduction. The simulation of large-scale, nonlinear problems can represent a considerable computational challenge. Parallel computing offers a significant resource for use in the solution of such problems; however, any parallel implementation must address a number of software and algorithmic issues to take advantage of these machines. As an example, we consider the modeling of small-strain plasticity problems. These problems exhibit multiple scales, if we wish to accurately model the extent of plastic zones in the material, and nonlinear properties. These nonlinear material properties can be solved by incrementally applying external forces; this incremental process requires the solution of a sequence of dynamically varying subproblems.

Our approach is based on a finite-element discretization where the small-scale structure is resolved by adaptive h-refinement of the computational mesh. An efficient parallel implementation for these problems requires the "vertical integration" of a number of different computational tasks: (1) parallel adaptive refinement of an unstructured mesh, (2) dynamic partitioning and redistribution of the mesh, (3) computation of internal stress with adjustments for plasticity, and (4) assembly and solution of large, sparse linear systems. The software used in this exercise is from SUMAA3d, a project that aims to develop scalable algorithms and software for problems based on unstructured meshes. This software has been used successfully in a similar approach for the solution of linear elasticity problems [6]. An important feature of this approach is that the user need not write any parallel software.

This paper is organized as follows. In §2 we review the radial-return approach used in modeling small-strain plasticity. In §3 we present a high-level description of

* The first author received support from the Computational Science Graduate Fellowship Program of the Mathematical, Information and Computational Sciences Division of the Office of Computational and Technology Research within the DOE Office of Energy Research. The second author received support from NSF grants ASC-9501583, CDA-9529459, and ASC-9411394. The third author was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

†The address of the first author is Department of Civil and Environmental Engineering, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213. The address of the second author is The Bradley Department of Electrical and Computer Engineering, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061. The address of the third author is Department of Computer Science and Engineering, The Pennsylvania State University, University Park, PA 16802.

the framework used to solve these problems, and we discuss issues particular to this parallel implementation. In §4 we describe the experimental results from an IBM SP system and an ATM-connected network of workstations. Finally, we summarize the results in §5.

2. Small-Strain Plasticity. For many engineering applications, the assumption of a linear stress-strain relationship fails to accurately model the constitutive behavior of real-world materials. Small-strain plasticity theory presents a method to solve problems in solid mechanics that are geometrically linear, but materially nonlinear. For materially nonlinear problems, it is not necessary to rewrite the basic variational statements; rather, the nonlinear problem is solved by applying the external forces incrementally and ensuring that force equilibrium is achieved at each load step before proceeding to the next load step.

In developing an adaptive scheme for the treatment of problems in incremental elastoplasticity, the main issues of concern are the calculation of a suitable error indicator function, the incorporation of a mesh refinement strategy, and the selection of a method for mapping the internal variables between successive meshes. Efforts have been directed toward the development of a posteriori error estimators for applications in small-strain plasticity [1, 2, 3, 4, 5, 17], large-strain plasticity [11, 15], and localized plasticity [14, 16]. Aspects of nonlinear adaptive finite-element methods related to efficiency and implementation, however, have received comparatively little attention; we address these aspects here. We use (1) the parallel, h-adaptive software from the SUMAA3d project to compute updated meshes; (2) linear 3-node triangular elements to produce a constant-stress approximation over each element; and (3) a simple mapping scheme in which refined elements inherit the interpolated internal variables of their parent elements.

Owen and Hinton [13] present a well-defined formulation for small-strain elastoplastic analysis based on a Von Mises yield condition and a radial-return method for enforcing the yield condition. Plastic deformation begins once the stress level exceeds the material's yield stress. For the case of metal plasticity, the yield condition may be written as

$$F = f(\sigma) - k(\kappa) = 0, \tag{2.1}$$

where $f(\sigma)$ is some scalar measure of the stress level and k is the yield limit, which may depend on κ, a material parameter that can represent strain-hardening phenomena. For the case of ideal plasticity, $k(\kappa)$ takes the constant value of the uniaxial yield stress of the material, σ_Y. The numerical results in this study were performed under the assumption of ideal plasticity. We use the yield criterion suggested by Von Mises,

$$\overline{\sigma} = \sqrt{3}\sigma_Y, \tag{2.2}$$

where $\overline{\sigma}$ is the effective stress.

During any increment of stress, the changes in strain are divided into an elastic part and a plastic part so that

$$d\epsilon = (d\epsilon)_e + (d\epsilon)_p. \tag{2.3}$$

The elastic strain increments are related to stress increments by a symmetric matrix of material constants, D. The plastic strain increments are assumed to be normal to the yield curve, F, based on the normality principle [18]. The total strain increment at a point can be written as

$$d\epsilon = D^{-1} d\sigma + d\lambda \frac{\delta F}{\delta \sigma}, \tag{2.4}$$

where $d\lambda$ is the plastic multiplier, which is nonzero only if the yield stress is exceeded. This relation is the origin of the nonlinearity in the formulation; the plastic increment of strain will occur only if the elastic stress increment puts the stress in violation of the yield condition.

The radial-return method is an approach for handling this nonlinearity to obtain material stresses consistent with its yield condition. In this method, the first iteration of each load step is assumed to take place under elastic conditions. However, the resulting stress level at any integration point may exceed the yield stress; the radial-return method reduces the effective stress by bringing the current stress back to the yield curve in the direction normal to the yield curve. During the process of returning stresses to the yield surface, plastic (irrecoverable) strains are accumulated. The direction for the stress reduction is given by the product of the elastic material coefficients, D, and the flow vector, a, which has six components corresponding to the partial derivatives of the yield surface with respect to the six independent Cauchy stresses. This reduction ensures that the direction of plastic flow is normal to the yield surface. The plastic multiplier, $d\lambda$, determines the magnitude of the stress reduction step and can be computed as

$$d\lambda = \frac{a^T d_D}{d_D^T a} d\epsilon. \tag{2.5}$$

Therefore, for a material point which currently has a stress level, σ_p, that exceeds the yield stress, σ_Y, the radial-return method is used to bring the stress level back to the yield curve as

$$\sigma_{red} = \sigma_p - d\lambda d_D. \tag{2.6}$$

The process described above still has a potential problem. Specifically, using large load steps in an analysis can lead to an inaccurate prediction of the final point, σ_{red}, on the yield curve, especially when the stress point, σ_p, is in the vicinity of a region of large curvature of the yield surface. Greater accuracy can be obtained by reducing the stress in successive stages through a so-called subincremental method [13]. This method enhances the accuracy of the stress reduction procedure outlined above and facilitates the use of larger load steps in nonlinear analyses. We use this approach within the framework described in the next section.

3. Parallel Algorithms. In this section we outline the algorithm used to solve these plasticity problems. We then describe the global mesh data structure used in the parallel solution of these problems; this data structure is the framework used by SUMAA3d in finite-element applications. Within this framework, we discuss the implementation of the major operations in the solution process for the plasticity formulation in §2. We describe how, through the use of the framework, a user can implement a complex finite-element application without writing any parallel code.

3.1. Solution Process. The parallel implementation is based on an implicit integration scheme where the strains are increased subincrementally as first described in [12]. Based on this approach, we give the algorithm in Figure 3.1. The first integration step is accomplished in the linear region; the external force is scaled so that the maximum internal stresses reach their elastic limit. Following this initial step, the tangent stiffness matrix, K_T, is computed based on the current element stresses, σ, at each incremental step. The radial return method is used to ensure that the stresses remain on the yield surface and satisfy the Von Mises relation, as described in the preceding section.

We use two tolerances. The first, tol_{res}, is a residual tolerance that specifies how small the magnitude of the residual force vector (a measure of nonequilibrium) should be in the solution of each nonlinear problem. A second tolerance, tol_{err}, determines how small the local error estimates for each element should be before another integration step is attempted.

```
Construct a mesh, M, to conform to the domain
Partition M across the processors
a ← 0 /* initialize the displacements */
Set L_ext based on elastic limit of internal stresses
While (L_ext < L_final) do
    Do
        Compute f_ext, the external stresses, based on L_ext
        Compute f_int, the internal stress
        Δf = f_ext − f_int
        While (||Δf|| > tol_res) do
            Assemble K_T
            Solve K_T Δa = −Δf
            a ← a + Δa
            Compute f_int, the internal stress
            Δf = f_ext − f_int
        endwhile
        Compute local error estimates for every element
        (Un)Refine mesh based on error indicator function
        Repartition mesh
    Until (local error ≤ tol_err)
    Increment L_ext
endwhile
```

FIG. 3.1. *Outline of the parallel plasticity algorithm*

The incremental elastoplastic analyses comprise a series of load steps in which the total external load, L_{ext}, is incrementally increased and applied to the structure. The iterative reduction of the residual forces is required to obtain a force equilibrium of the structure that is not in violation of the yield condition. We use a Newton-Raphson scheme for the solution of the nonlinear systems; the method is simple and displays good convergence. Refinement within the iterative solution process of each load step would disrupt the solution of these systems. Thus, we perform the mesh refinement after a solution is obtained for the load step.

The adaptive refinement of the finite-element mesh is facilitated by an indicator function. The output of the function is an indication as to whether each element should be refined or unrefined or is adequately resolved. Such functions are typically local in the sense that they depend only on data local to an element and, in some cases, neighboring elements. We have chosen to use a fairly complex indicator function for this problem to (1) demonstrate the generality of the parallel infrastructure discussed in the paper and (2) accurately resolve the region in which elements transition from elastic to plastic deformation. A characteristic of this transition region, in the problems of interest, is that the second derivatives of the effective stress are higher than elsewhere in the mesh. Our error indicator, therefore, is a function of the second derivatives of the effective stress, σ, on element e and is given by

$$\int_e ||\sigma_{xx}^2 + \sigma_{xy}^2 + \sigma_{yx}^2 + \sigma_{yy}^2|| dxdy. \tag{3.1}$$

In Figure 3.2 we illustrate the steps for computing this quantity for linear elements in two dimensions. We note that the complexity is somewhat reduced for quadratic elements. The effectiveness of this indicator is evaluated in §4.

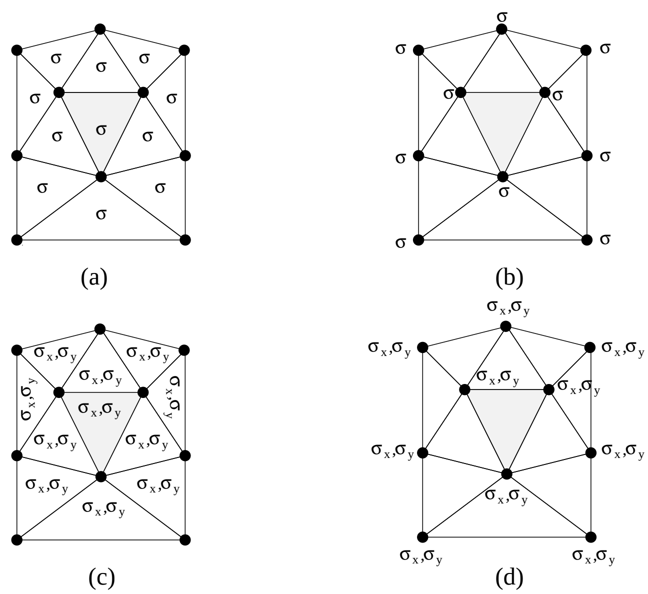

FIG. 3.2. *Computation and data required to compute an error indicator for the shaded triangle, based on the second derivatives of the effective stresses. In submesh (a) the effective stress at each triangle is computed; this computation is local to the triangle because only the displacements at each of its three vertices are required. In submesh (b) the stress at each vertex is computed by combining the contribution of all triangles adjacent to each vertex; note that data from outside the submesh is required for vertices on the outer edge of the submesh. In submesh (c) the gradient of the stress in each triangle is computed using the vertex stresses computed in (b). Finally, in submesh (d), the gradient of the stress at each of the vertices is computed as for submesh (b). The second derivatives of the stresses can now be computed for the shaded triangle using only the information at each of its three vertices.*

3.2. Data Structures. In this subsection we describe the data structures that are required for effective sequential and parallel computation. We then show how the computations in the algorithm from Figure 3.1 are implemented on this data structure and how the problem-specific aspects of these computations can be isolated from the parallel operations.

First we define the global data structure used to store the mesh and the required solution information. For convenient and effective computation, the mesh is stored such that (1) each element knows and can access the elements with which it shares an edge, (2) each element knows and can access the vertices contained by the element, and (3) each vertex knows and can access the elements containing it. Further, for this plasticity computation, solution information is associated with each of the vertices as well as with each of the elements. This global data structure is illustrated in Figure 3.3.

For parallel computation the mesh must be partitioned into submeshes that are assigned to each processor; however, the properties of the global data structure must

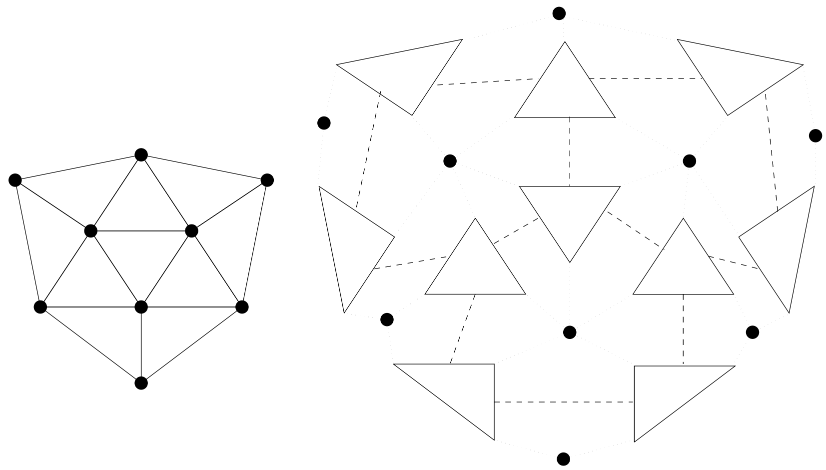

FIG. 3.3. *An exploded view of the data structure for the mesh on the left, is depicted on the right.*

be maintained. Each processor *owns* a unique set of vertices and a set of elements belonging to its submesh. Recall that the global data structure allows for each vertex to access all the elements containing that vertex. For the parallel case, each owned vertex should be able to access all the elements to which it is adjacent; in some cases these elements will be owned by other processors. This access can be accomplished either by sending a message[1] requesting data from another processor or by storing up-to-date "ghost" copies of the nonlocal elements. We have chosen to store ghost copies because they can be used to reduce the number of messages sent and, as described later, allow for problem-specific code to be isolated from the parallel code.

Similar to the vertex case, each owned element must be able to access elements with which edges are shared and each of the vertices contained within the element. This access is facilitated by storing ghost copies of the necessary nonlocal elements. The required ghost elements and vertices for an example mesh are illustrated in Figure 3.4. This approach can be used for higher-order elements; for example, it has been used to solve an elasticity problem with fourth-order elements [6]. We note that every ghost element and vertex must have up-to-date copies of the solution information; for example, each ghost vertex must have copies of the most recently computed displacements. Clearly, some method must be used to coordinate the updates to these ghost copies and the associated data. This updating is part of any parallel operation on the global data structure; we will discuss several such operations in the following subsection.

Finally, we note that other data structures, including vectors and matrices, are partitioned according to vertex ownership. For example, consider a vector representing the displacements at all the vertices; if a processor owns a vertex, then it owns and stores the entries associated with the displacements at that vertex.

3.3. Parallel Operations. Virtually every operation of the algorithm in Figure 3.1 requires parallel computation on the global data structure. In this subsection we describe how this global data structure, in combination with certain parallel library operations, allows for a user to construct a complex parallel application without

[1] We assume that the parallel architecture is based on message passing or that there is memory "local" to processors such that it is advantageous to store frequently accessed data on this local memory. For simplicity, we will refer to all nonlocal memory accesses as messages.

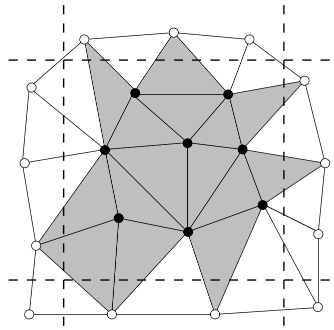

FIG. 3.4. *The dashed lines depict a partitioning of the data. The shaded vertices and elements are owned by the processor assigned the center partition. The unshaded elements and vertices are the ghost copies of nonlocal elements and vertices required for the global mesh data structure.*

writing any parallel code. The fundamental idea is that users write subroutines that operate on individual elements, individual vertices, and local vectors and then call library operations to coordinate the parallel operations.

3.3.1. Matrix and Vector Assembly. The plasticity computation requires the assembly of the tangent stiffness matrix, K_T, and the external and internal stress vectors, f_{ext} and f_{int}. Similar assemblies are required for elasticity computations as well as most other finite-element applications. The size of the element matrices and vectors and the relative cost of the element computations differ widely across these applications. For example, unlike standard linear elasticity, the computation of the element matrix for this plasticity formulation depends on the computed stresses for the element. What is common across the vast majority of applications is that element matrix and vector computations require only information that is local to that element and, further, that these element matrices be assembled into a global matrix whose structure depends on the underlying finite-element mesh.

We have constructed parallel library routines that, based on the underlying mesh, compute the structure of the global sparse matrix and allocate the necessary storage. Another library routine calls the user's element matrix computation subroutine, passed as an argument, for every element in the mesh, and assembles these element matrices into the global matrix. The user need only write the subroutine for evaluating a single element and then call the appropriate parallel library subroutines. A similar set of routines exists for the assembly of vectors from element-based vectors.

3.3.2. Vector Operations. Most applications require a small number of parallel vector operations. In the algorithm in Figure 3.1, an inner product is required to compute the norm of the residual force vector, Δf, a vector subtraction/addition is required to compute Δf and a, and gather/scatter operations are required to apply these vectors to the global mesh data structure.

First, we recall that a processor is responsible only for the entries of a vector corresponding to vertices that it owns. A processor can therefore allocate a vector whose size is proportional to the number of local vertices and operate only on this "local" vector. For example, a vector addition/subtraction is purely local and can be implemented without parallel computation. An inner product requires only a single global reduction, a library routine that is available in most message-passing libraries, including MPI [10].

The vector operations that depend on the global mesh data structure are those that project newly computed values onto the mesh or try to recover values from the mesh and place them in a vector. Because of the complexity of the global mesh data structure and the need for updating ghost copies, library routines are used to isolate the user from the global data structure. The means of isolation must allow for a variety of operations because of the wide range of applications under consideration. We have implemented two basic operations: (1) a **gather** operation, in which the information associated with vertices, or a subset of the information, is gathered into a vector, and (2) a **scatter** operation, in which the entries of a vector are scattered to the corresponding vertices, including all ghost copies. Note that a similar functionality exists for element-based information. The user can implement a wide range of operations on these local vectors without writing any parallel code and then call the gather/scatter operations to update the global mesh data structure.

For example, consider the updating of the nodal displacements $a \leftarrow a + \Delta a$. At the beginning of the operation, the incremental and current displacements reside with each of the vertices and, given that Δa was computed as part of a matrix solution on the previous step, Δa exists only as a vector with entries only on the owning processor. At the end of the operation, every vertex, including ghost copies, must have the correct value of a. To perform this update: (1) a library routine is called to gather the current displacements into a local vector on each processor (only owned vertices are involved, not ghost vertices); (2) the two local vectors, representing a and Δa, are added in a local vector operation implemented by the user; and (3) a library routine is called to scatter the resulting sum back onto the mesh, including the ghost copies.

3.3.3. Mesh Refinement. A parallel library has been written for adaptive mesh refinement [9]; a user makes a call to the library, which handles the refinement and updating of the global mesh data structure. The user must provide a function that indicates whether an element is to be refined, unrefined, or left unchanged. This function accesses only information local to an element and the vertices contained in that element.

Because of its complexity, the error indicator evaluation described in Figure 3.2 involves several of the operations described above: (1) compute the effective stress for each element and assemble these stresses into a vector that represents the effective stress at each vertex, (2) compute the gradient of the effective stress for each element and assemble these stresses into a vector that represents the gradient of the effective stresses at each vertex, and (3) compute the norm of the second derivatives of the effective stress at each element and determine whether (un)refinement of the element is required. Note that each step requires the vertex data that we computed at the prior step (we are computing a sequence of derivatives).

The effective stress on an element in Step 1 can be computed from the displacements at the vertices contained in the element. The stresses at the vertices can therefore be computed by an assembly operation just as for f_{ext}.[2] The vector containing these vertex stresses is then scattered to the appropriate vertices, including the ghost copies. Note that the user need only write the code to compute the effective stress on a single element.

Given that the effective stress at every vertex is now available, the gradients of the effective stress can be computed from information local to an element. Step 2 can, therefore, be computed exactly as Step 1, with the user writing only the code to compute the gradient of the stress on a single element, given the stresses at each of the vertices. After Step 2, the gradients of the effective stress are available at every

[2]Recall that, for linear elements, the stresses are constant on each element. We are computing the stress at a vertex by averaging the stress at each of the neighboring elements.

vertex of the mesh, thereby allowing the computation of the error indicator function to proceed using information local to the element.

3.3.4. Additional Parallel Libraries. Some additional parallel operations are required for most applications. These libraries operate on the global mesh data structure or the associated matrices/vectors. They are sufficiently complex that a user cannot and should not be expected to write them. The BlockSolve95 library [7, 8] is used to solve the linear systems; the user simply makes a call to BlockSolve95, indicating the matrix and vector to be solved as well as the desired options.

Because the mesh is refined more in some areas than others, a load imbalance occurs after each refinement step. This imbalance is remedied by the repartitioning step indicated in the algorithm in Figure 3.1. The repartitioning is accomplished by using the unbalanced recursive bisection algorithm presented in [6]. The computation of the new partition requires no interaction with the user; however, moving vertices and elements to their new location does require some interaction. This interaction is necessary because the information stored with every vertex and element is defined by the user. This flexibility is required to implement a wide range of applications. However, because the structure of the information is not defined in the global data structure, the user must supply code to pack and unpack the information at vertices and elements. The partitioning library software calls these pack/unpack routines for each vertex and/or element being moved to a new location.

3.3.5. Summary of User-Supplied Code. The user is responsible for writing a main routine, similar to that in Figure 3.1, that calls the appropriate parallel library functions. In addition, the user must supply code for operations on individual elements and vertices. These operations include the computation of an element stiffness matrix and the packing/unpacking of user data associated with a single vertex. The user need not write any parallel code nor operate directly on the mesh data structure.

4. Experimental Results. In this section we present results that illustrate the performance of this application on two parallel architectures. In addition, we demonstrate the utility of the adaptive refinement technique proposed in §3. The results are given for two- and three-dimensional versions of a pressure vessel with a single crack on the outer edge. Meshes for the two problems at an intermediate stage of the refinement process at moderate loading are shown in Figure 4.1.

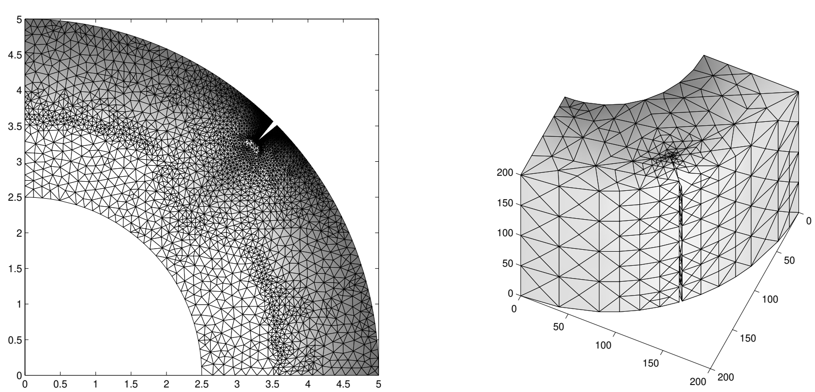

FIG. 4.1. *A partially refined mesh for the two-dimensional pressure vessel is given on the left and the surface triangles for a partially refined mesh for the three-dimensional pressure vessel is given on the right. All dimensions are in millimeters.*

The application is implemented in the C programming language using MPI for communication [10] and the parallel mesh libraries from SUMAA3d described in §3. The results are collected from two architectures: (1) an IBM SP parallel computer with SP3 thin nodes, each node with two 128 megabyte memory cards, and a TB3 switch; and (2) an ATM network of workstations composed of a set of 12 Sun Ultra 2 Model 2170 workstations, each with 256 megabytes of memory, connected via 155 Mbs ATM links to a Synoptics ATM switch.

TABLE 4.1
Characteristics of the final mesh for each of the testbed problems

Name	Load Steps	Max. Load	Number of Vertices	Number of Elements	Number of Unknowns	Number of Nonzeros
PV3D0	10	15.0	16,804	85,874	50,412	1,047,948
PV3D1	10	14.0	35,972	187,223	107,916	2,264,997
PV3D2	10	15.0	140,615	747,319	421,845	8,935,752
PV3D3	10	14.0	269,044	1,445,001	807,132	17,216,412
PV3D4	10	14.0	419,873	2,262,898	1,259,619	26,898,195
PV2D1	10	14.0	47,555	98,211	95,110	709,957
PV2D2	10	14.0	93,908	194,976	187,816	1,404,396
PV2D3	10	14.0	185,548	369,449	371,096	2,776,628

We give basic characteristics of the final mesh for each of the problems in the testbed in Table 4.1. The two-dimensional pressure vessel problems begin with PV2D and the three-dimensional problems begin with PV3D; different size problems are obtained by adjusting the error tolerance, tol_{err}. The number of steps used in the incremental loading is given in the column labeled Load Steps. The maximum loads are given in dynes per unit surface area. Of significant concern is the partitioning of the global data structure. This partitioning affects the amount of interprocessor communication as well as the number of ghost copies that must be kept. In Table 4.2 we give statistics that indicate the quality of the partitioning as well as the number of ghost copies. The maximum volume of data sent by any processor is proportional to the maximum number of cross edges, and the maximum number of messages sent is proportional to the maximum number of neighbors for any single processor. We note that it is advantageous to have large subproblems assigned to each processor: the quality of the partitioning and the percentage of ghost copies improve as the problem size per processor increases. We can expect higher parallel efficiencies for larger subproblem sizes.

4.1. Execution Characteristics. We now examine the characteristics of the runtime on the two architectures. In Table 4.3 we examine the percentage of time spent in the operations discussed in §3. The vast majority of the execution time is spent in solving the linear systems; the proportion of the execution time spent in this phase typically increases as the problem size increases.

In Tables 4.4 and 4.5, respectively, we give the maximum and average execution rates for several operations. The maximum rate is the highest rate for any solution iteration and the average rate is the weighted average of the rates for all solution iterations. The maximum rates typically are achieved when the mesh has reached its maximum size, because the parallel efficiency is higher for the larger mesh sizes. Because most of the execution time is spent with these larger mesh sizes, however, the average rates will be fairly close to the maximum rate. The average refinement rates for the three-dimensional problems are much lower than the maximum because there are many steps during which little or no refinement takes place. In addition, there is

TABLE 4.2
Indicators of the quality of the partitioning. The number of processors is denoted by p.

Name	p	Max. Number of Cross Edges	Max. Number of Neighbors	Total Ghost Vertices	Total Ghost Elements
PV3D0	8	4,205	7	12,401	33,478
PV3D0	12	3,044	9	12,810	33,836
PV3D1	8	6,840	7	22,846	64,016
PV3D1	12	3,184	8	14,764	37,683
PV3D2	24	8,011	13	60,321	162,613
PV3D3	48	7,836	25	122,588	331,875
PV3D4	64	8,626	21	202,627	557,739
PV2D1	16	503	6	6,126	3,942
PV2D2	32	499	6	12,704	8,215
PV2D3	64	477	10	26,400	17,053

TABLE 4.3
Percentage of the execution time spent in the major phases of the solution process. The number of processors is denoted by p.

Name	Arch.	p	Partition & Move	Matrix Solution	Assembly (matrix/rhs)	Gather & Scatter	Mesh Refine
PV3D0	ATM	8	0.7	91.1	6.9	0.6	0.4
PV3D0	ATM	12	1.2	92.4	5.4	0.6	0.4
PV3D1	ATM	8	0.9	89.4	8.0	0.8	0.6
PV3D1	ATM	12	1.3	91.1	6.3	0.5	0.6
PV3D3	SP	48	2.7	87.6	7.2	0.6	0.9
PV3D4	SP	64	2.7	88.6	6.4	0.5	0.9
PV2D1	SP	16	1.2	96.8	1.3	0.14	0.19
PV2D2	SP	32	1.1	97.7	0.9	0.13	0.14
PV2D3	SP	64	1.1	97.1	0.5	0.13	0.25

still a significant cost involved in evaluating the error indicator for each element in the mesh which is included in the refinement times. The indicator functions must be evaluated whether or not elements are refined.

Most of the operation rates scale well as the problem size scales with the number of processors. Again, the exception is the refinement algorithm. In [9] we show that this refinement algorithm is scalable under certain conditions. For this plasticity problem, however, the three-dimensional case has two inherent difficulties: (1) the number of elements refined relative to the total number of elements is small, and hence there is a large overhead associated with examining all the elements when refining only a few, and (2) the elements being refined are concentrated on a small number of processors, leading to an imbalance in the work.

4.2. Adaptive Refinement Results. The refined meshes in Figure 4.1 illustrate the refinement of elements transitioning from elastic deformation to plastic deformation. The purpose of using adaptive refinement is to significantly reduce the number of vertices required to achieve an acceptable error. In Figure 4.2, we compare the approximation error as a function of the number of vertices used for adaptive refinement and for uniform refinement. These results are obtained from the two-dimensional pressure vessel problem. For uniform refinement, a mesh is constructed

TABLE 4.4

Maximum rates of execution for the major phases of the solution process. Denoted are the number of vertices added per second (refinement), the megaflops rate for the matrix solution process, and the number of elements assembled per second for the matrix assembly process. The number of processors is denoted by p.

Name	Arch.	p	Vertices Refined per Second	Mflops	Elements Assembled per Second
PV3D0	ATM	8	319	46.5	33,920
PV3D0	ATM	12	339	45.8	46,194
PV3D1	ATM	8	411	64.7	38,681
PV3D1	ATM	12	636	73.3	57,983
PV3D3	SP	48	3,361	1,033	360,844
PV3D4	SP	64	3,445	1,369	485,188
PV2D1	SP	16	20,659	251	413,219
PV2D2	SP	32	34,791	452	812,538
PV2D3	SP	64	34,296	835	1,510,330

TABLE 4.5

Average rates of execution for the major phases of the solution process. Denoted are the number of vertices added per second (refinement), the megaflops rate for the matrix solution process, and the number of elements assembled per second for the matrix assembly process. The number of processors is denoted by p.

Name	Arch	p	Vertices Refined per Second	Mflops	Elements Assembled per Second
PV3D0	ATM	8	169	41.1	28,191
PV3D0	ATM	12	180	38.5	40,612
PV3D1	ATM	8	186	56.5	29,463
PV3D1	ATM	12	213	59.1	46,452
PV3D3	SP	48	1,502	926	289,711
PV3D4	SP	64	1,688	1,174	369,677
PV2D1	SP	16	7,374	243	372,331
PV2D2	SP	32	13,169	428	689,995
PV2D3	SP	64	6,227	779	1,435,557

for which the area of all elements is approximately equal. Note the significant advantage obtained by using adaptive refinement.

5. Summary. We have given a framework for implementing a variety of applications on unstructured finite-element meshes. This framework allows a user to construct a complex, large-scale application without writing any parallel code. This is accomplished by providing a carefully selected set of parallel library subroutines that operate on a shared global data structure representing an unstructured finite-element mesh.

To demonstrate the utility of this framework we have constructed a code for solving large-scale small-strain plasticity problems in two and three dimensions. This code is implemented with adaptive refinement to allow for the efficient modeling of elastic/plastic transition regions. We have demonstrated that the resulting parallel application runs effectively on an IBM SP parallel computer and on a network of workstations. A similar approach could be used in the parallel implementation of a wide variety of other finite-element applications.

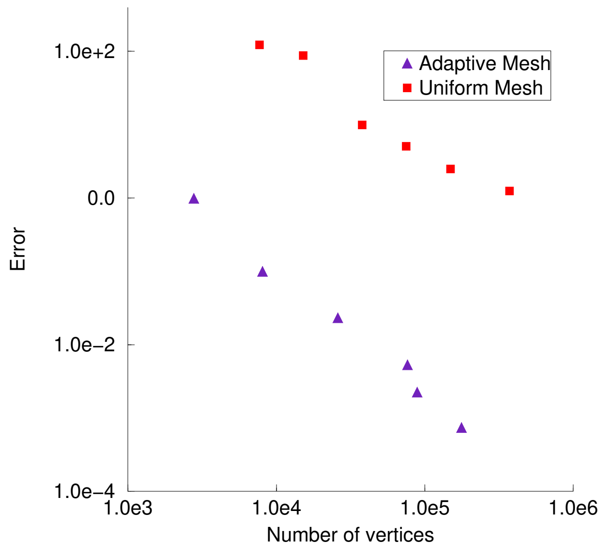

FIG. 4.2. *Comparison of the maximum element error as a function of the number of vertices used for a mesh using elements of uniform area and for an adaptive mesh*

REFERENCES

[1] C. Almeida, *An effective adaptive procedure in nonlinear finite element analysis*, in Proceedings of the International Computers in Engineering Conference and Database Management Symposium, A. Busnaina and R. Rangan, eds., Sept. 17-20 1995, pp. 197–203.

[2] I. Babuska, O. Zienkiewicz, J. Gago, and E. de Oliveira, eds., *Accuracy Estimates and Adaptive Refinements in Finite Element Computations*, Wiley, 1986.

[3] C. Carstensen, D. Zarrabi, and E. Stephan, *On the h-adaptive coupling of fe and be for viscoplastic and elasto-plastic interface problems*, Journal of Computational and Applied Mathematics, 75 (1996), pp. 345–363.

[4] H. Cramer, M. Rudolph, G. Steinl, and W. Wunderlich, *A hierarchical adaptive finite element strategy for elastic-plastic problems*, in Advances in Finite Element Technology, B. Topping, ed., Civil-Comp Press, Edinburgh, UK, 1996, pp. 151–159.

[5] C. Johnson and P. Hansbro, *Adaptive finite element methods for small-strain plasticity*, in Finite Inelastic Deformations-Theory and Applications, D. Besdo and S. Stein, eds., Springer, 1992, pp. 273–288.

[6] M. T. Jones and P. E. Plassmann, *Computational results for parallel unstructured mesh computations*, Computing Systems in Engineering, 5 (1994), pp. 297–309.

[7] ———, *Scalable iterative solution of sparse linear systems*, Parallel Computing, 20 (1994), pp. 753–773.

[8] ———, *BlockSolve95 users manual: Scalable library software for the parallel solution of sparse linear systems*, ANL Report ANL-95/48, Argonne National Laboratory, Argonne, Ill., Dec. 1995.

[9] ———, *Parallel algorithms for adaptive mesh refinement*, SIAM Journal on Scientific Computing, 18 (1997), pp. 686–708.

[10] Message Passing Interface Forum, *MPI: A message-passing interface standard*, International Journal of Supercomputing Applications, 8 (1994).

[11] L. Nam-Sua and K. Bathe, *Error indicators and adaptive remeshing in large deformation finite element analysis*, Finite Elements in Analysis and Design, 16 (1994), pp. 99–139.

[12] G. Nayak and O. Zienkiewic, *Elasto-plastic stress analysis. Generalization for various constitutive relations including strain softening*, Int. J. Num. Mech. Sci., 5 (1972), pp. 113–35.

[13] D. Owen and E. Hinton, *Finite Elements in Plasticity: Theory and Practice*, Pineridge Press Limited, 1980.

[14] D. Peric, J. Yu, and D. Owen, *On error estimates and adaptivity in elastoplastic solids: Applications to the numerical simulation of strain localization in classical and cosserat continua*, International Journal for Numerical Methods in Engineering, 37 (1994), pp. 1351–1379.

[15] R. Tatambe, S. Yunus, C. Rajakumar, and S. Saigal, *Examination of flux projection-type error estimators in nonlinear finite element analysis*, Computers and Structures, 54 (1995), pp. 641–653.

[16] N. Wiberg, X. Li, , and F. Abdulwahab, *Adaptive finite element procedures in elasticity and plasticity*, Engineering with Computers, 12 (1996), pp. 120–141.

[17] K. Yuge and N. Iwai, *Nonlinear finite element analysis by progressive mesh refinement. 4. Studies of the criteria for mesh refinement*, Technical Reports of Seikei University, Japan, 32 (1995), pp. 25–26.

[18] O. Zienkiewicz and R. Taylor, *The Finite Element Method Volume 2: Solid and Fluid Mechanics, Dynamics and Non-linearity*, McGraw-Hill Book Company (UK) Limited, fourth ed., 1991.