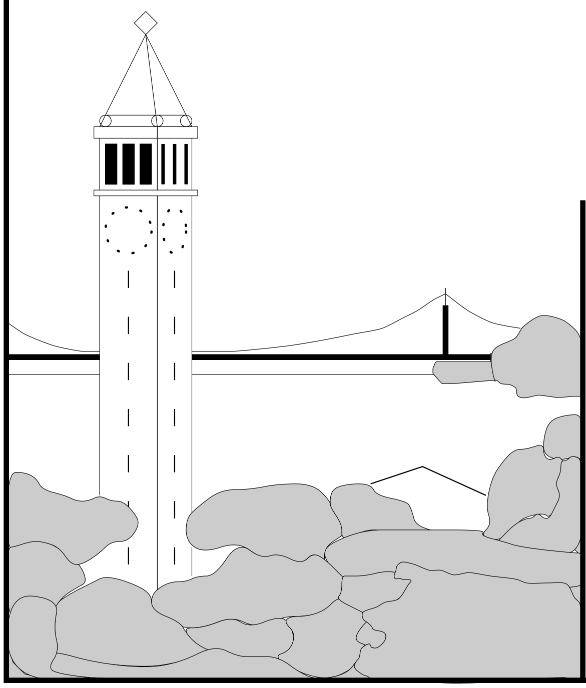
Parallel Multigrid Algorithms for Unstructured 3D Large Deformation Elasticity and Plasticity Finite Element Problems

Mark Adams [1]



Report No. UCB/CSD-99-1036

January 1999

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Abstract

Multigrid is a popular solution method for the set of linear algebraic equations that arise from PDEs discretized with the finite element method. The application of multigrid to unstructured grid problems, however, is not well developed. We discuss a method that uses many of the same techniques as the finite element method itself, to apply standard multigrid algorithms to unstructured finite element problems. We use *maximal independent sets* (MISs), like many "algebraic" multigrid methods, as a heuristic to automatically coarsen unstructured grids. The inherent flexibility in the selection of an MIS allows for the use of heuristics to improve their effectiveness for a multigrid solver. We present heuristics and algorithms to optimize the quality of MISs, and the meshes constructed from them, for use in multigrid solvers for unstructured problems in solid mechanics. We also discuss parallel issues of our algorithms, and multigrid solvers in general, and describe a parallel finite element architecture that we have developed to parallelize a state-of-the-art research finite element code in a natural way for the common computer architectures of today. We show that our solver, and parallel finite element architecture, does indeed scale well, with test problems in 3D large deformation elasticity and plasticity, with over 26 million degrees of freedom on a 640 processor Cray T3E (with 55% parallel efficiency), and on 84 IBM 4-way SMP PowerPC nodes.

Key words: multigrid, unstructured meshes, parallel solvers, finite element method

1 Introduction

This work is motivated by the success of the finite element method in simulating complex physical systems in science and engineering, coupled with the wide spread availability of ever more powerful computers, which has lead to the need for efficient equation solvers for implicit finite element applications. Finite element matrices are often poorly conditioned - this fact has made the use of direct solvers popular as they are relatively unaffected by the condition number of the matrix. As the scale of the problems increase, direct methods, however, possess sub-optimal time and space complexity when compared to iterative methods. Also, as larger and faster computers are becoming more widely available, to a larger number of research institutions and industries, the use of iterative methods will become increasingly more necessary. Thus, given the computational resources available today, and those that are continually being introduced, direct methods are inefficient in solving large problems - resulting in the need to resort to iterative methods.

Many iterative methods are notoriously unreliable on finite element problems of interest. Multigrid is one of a family of highly optimal multilevel domain decomposition methods [27], and is known to be a highly effective method to solve finite element matrices [15, 19, 30, 8, 11, 23]. The application of multigrid techniques to unstructured meshes, that are the hallmark of the finite element method, has not been well developed, and is currently an active area of research. In particular, the development of practical multigrid methods for unstructured finite element problems, of arbitrary geometric complexity and size, is an open problem. This paper discusses methods for the effective application of classical multigrid methods in the solution of the large sparse system of equations that arise from unstructured finite element problems in 3D large deformation elasticity and plasticity.

This paper proceeds as follows: Section §2 briefly introduces multigrid; and section §3 describes our basic algorithm; section §4 describes the methods to optimize the algorithm for 3D solid mechanics problems. Parallel algorithmic issues are discussed in section §5; our parallel finite element architecture is described in section §6. Numerical results on 3D problems in large deformation elasticity and plasticity, with incompressible materials and large jumps in material coefficients, are presented in section §7 with over 26 million degrees of freedom on 640 processors of a Cray T3E (with 55% parallel efficiency) and 84 4-way SMP IBM PowerPC nodes. We conclude in section §8 with potential directions for future work.

2 Multigrid

Multigrid is known to be *the* optimal solution method for the finite difference Poisson equation in serial; in parallel, the FFT is competitive with multigrid [10]. However, unlike the FFT, multigrid has been applied to unstructured second order finite element problems in elasticity [24, 9] and plasticity [15, 19, 23], as well as fourth order problems [14, 30].

Multigrid has been an active area of research for almost 30 years and much literature can found on the subject - a quick introduction is, however, warranted [7]. Multigrid is motivated by the observation that simple (and inexpensive) iterative methods like Gauss-Seidel, Jacobi, and block Jacobi [10], are effective at reducing the *high* frequency error effectively, but are ineffectual in reducing the low frequency content of the error. These simple solvers are called *smoothers* as they render the error smooth by reducing the high frequency content of the error (actually they reduce high *energy* components of the error, leaving the low energy components which are smooth in, for example, Poisson's equation with constant coefficients). The ineffectiveness of simple iterative methods can be ameliorated by projecting the solution onto a smaller space, that can resolve the low frequency content of the solution, in exactly the same manner as the finite element method projects the continuous solution onto a finite dimensional subspace to compute an approximation to the solution. This "coarse grid correction" does not eliminate the low frequency error exactly, though it can "deflate" low frequency error to high frequency error (which can then be eliminated cheaply), by removing an approximation to the low frequency components from the error. Thus, the goal of a multigrid method is to construct, and compose, a series of function spaces in which iterative solves or small direct solves, working together, can economically reduce the entire spectrum of the error. Many multigrid algorithms have been developed - we use the "full" multigrid algorithm (FMG), in our numerical experiments. *V-cycle* multigrid, which is closely related to FMG (see [7]), is shown in Figure 1. Figure 1 uses a provided smoother $x \leftarrow S(A, b)$, and restriction operator R that maps residuals from the fine grid space to the coarse grid space.

function $MGV(A_i, r_i)$
    if there is a coarser grid
        $x_i \leftarrow S(A_i, r_i)$
        $r_i \leftarrow r_i - Ax_i$
        $r_{i+1} \leftarrow R_{i+1}(r_i)$
        $x_{i+1} \leftarrow MGV(R_{i+1}A_iR^T_{i+1}, r_{i+1})$
        $x_i \leftarrow x_i + R^T_{i+1}(x_{i+1})$
        $r_i \leftarrow r_i - A_i x_i$
        $x_i \leftarrow x_i + S(A_i, r_i)$
    else
        $x_i \leftarrow A_i^{-1} r_i$
    return x_i

Figure 1: Multigrid *V-cycle* Algorithm

3 Our method

We build on an algorithm first proposed by Guillard [16] and independently by Chan and Smith [9]. The purpose, of this algorithm, is to automatically construct a coarse grid, from a finer grid, for use in standard multigrid algorithms. This method is applied recursively to produce a series of coarse grids, and their attendant operators, from a "fine" (application provided) grid. A high level view of the algorithm is as follows:

- The vertex set at the current level (the "fine" mesh) is *evenly* coarsened, using an maximal independent set (MIS) algorithm to produce a much smaller subset of vertices.
- The new vertex set is then automatically remeshed with tetrahedra.
- Standard finite element shape functions for tetrahedra are used to produce the restriction operator (R). The transpose of the restriction operator is used as the interpolation operator.
- The restriction operator is then used to construct the (Galerkin) coarse grid operator from the fine grid operator, i.e. $A_{coarse} \leftarrow RA_{fine}R^T$.

Multigrid requires two types of operators: first, *restriction* and *interpolation* operators, which can be implemented with a rectangular matrix (R and R^T respectively); and second the PDE operator, a sparse matrix, for each coarse mesh (the fine grid matrix is provided by the finite element application). The coarse grid matrix can be formed in one of two ways - either algebraically to form Galerkin coarse grids ($A_{coarse} \leftarrow RA_{fine}R^T$), or by creating a new finite element problem on each coarse grid, thereby letting the finite element implementation construct the matrix. There are advantages and disadvantages to each approach.

The algebraic method has the advantage that it places less demand on the user by not requiring that a problem be completely defined on the coarse mesh. The construction of good quality meshes is a challenging and expensive part of using the finite element method. Requiring good coarse meshes may be an onerous responsibility for the solver to place on the user. Mesh generators, be they automatic or semi-automatic, are not accustomed to approximating the domain automatically (i.e. not strictly maintaining the topology of the domain) which is often required for efficiency - especially on the coarsest grids of problems with complex geometry. The explicit construction of a new finite element problem on the coarse grid may however provide better quality multigrid operators, but we are not aware of any direct comparison of the two methods. We have opted for the algebraic approach - this requires that we construct only the restriction matrices; all of the operators that multigrid requires can be transparently constructed from these restriction operators. Our work thus centers on the construction of good quality restriction operators.

The next section discusses methods and heuristics useful in optimizing the quality of these restriction operators. Our methods use coordinate data available in finite element simulations, and element data that is available when continuum elements are used. We show how to use this data to categorize topological elements of the finite element mesh (i.e. corners, edges[2], surfaces, and interiors), and to use this information in a logical way to modify the graph that is used in the MIS algorithm.

4 Automatic coarse grid creation with unstructured meshes

This section introduces the components that we use for the automatic construction of coarse grids on unstructured meshes. First we state the general purpose of the coarse grids in multigrid algorithms: the goal of the coarse grid function spaces is to approximate the low frequency part of the spectrum of the current grid well. Each successive grid's function space should (with a drastically reduced vertex set) approximate, as best as it can, the lowest frequencies (or eigen functions) of the previous grid. That is, with say 10% of the vertices from the fine grid, it is natural to expect that one could only represent the lowest 10% of the fine grid spectra well. It is not possible to satisfy this criterion directly (on unstructured grids), but a natural heuristic is to represent the *geometry* as well as possible. With a good representation of the geometry (implicitly assumed on structured grid problems) one can hope that the finite element function spaces, of the coarse grids, will approximate the lowest modes of the fine grid well.

Thus, our basic approach is to construct a low order geometric representation of the "fine", or current grid, with a "coarse" grid; and recursively apply this process; use standard finite element function spaces; and use this series of function spaces in one of many standard multigrid algorithms. An alternative, and promising approach, to construct these low order geometric representations, is to use computational geometry techniques to characterize features and algorithms to maintain them on the coarser grids [28]; though the most widely used method, to construct the coarse grids, is to use a *maximal independent set* as a heuristic to evenly coarsen the vertex set. An MIS is not unique in general, and an arbitrary MIS is not likely to perform well, thus we use heuristics to improve performance.

We motivate our geometric approach by first looking at a typical *structured* multigrid example in Figure 2. We can characterize the behavior of multigrid on structured meshes, as shown in a 2D example in Figure 2, as: "select every other vertex", in each dimension, for use in the coarse grid. The use of these grids on structured problems is provably very effective for some problems [10].

To apply multigrid to unstructured meshes it is natural to try to imitate the behavior of the structured algorithm in the hope of imitating its success. Consider that, in addition to evenly coarsening the vertex set, the coarse grids in Figure 2 also emphasize the boundaries. The pioneers of our multigrid algorithm

[2]we use *edges* here to mean a topological feature and *not* a graph edge - the type of "edge" should be obvious from the context in following discussions.

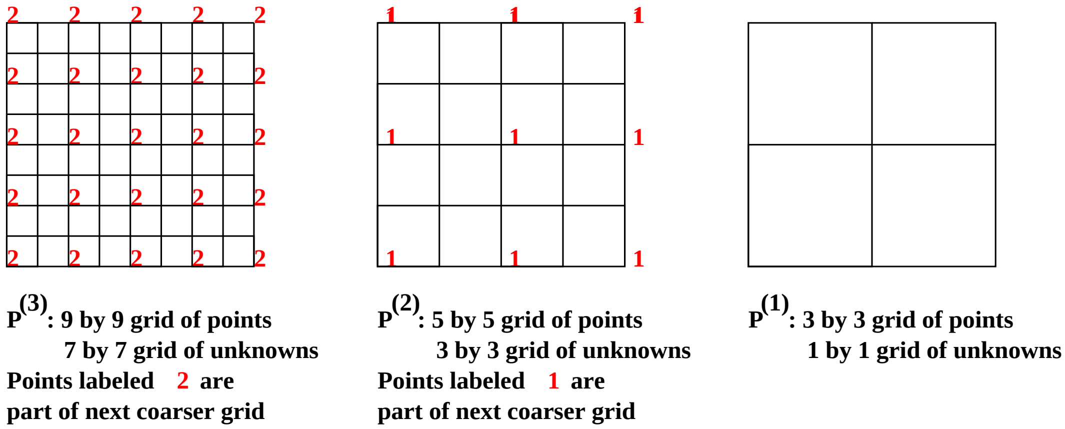


Figure 2: Multigrid coarse vertex set selection on structured meshes

(Guillard [16] and Chan and Smith [9]) use some simple 2D heuristics to preserve boundaries and emphasis corners as well.

One description of multigrid meshes on regular grids is: place each vertex v in a *topological category* of dimension d - for instance, corners ($d = 0$), edges ($d = 1$), surfaces ($d = 2$), and interiors ($d = 3$). Given these categories we have collections of features for each category (e.g. a set of 3D connected surface vertices bounded by edge vertices would be one surface in the set of surfaces). Now the regular mesh in Figure 2 produces an MIS *within* each feature, and hence the number of vertices in features (with dimension d) is reduced by a factor of about $\frac{1}{2^d}$. This section discusses algorithms to implement these observations.

4.1 Maximal independent set algorithms

An *independent set* is a set of vertices $I \subseteq V$ in a graph $G = (V, E)$, in which no two members of I are adjacent (i.e. $\forall v, w \in I, (v, w) \notin E$); a *maximal independent set* (MIS) is an independent set for which no proper superset is also an independent set. Maximal independent sets are a popular device in selecting the "points" for unstructured multigrid methods. The simple greedy MIS algorithm [22, 18], is show in Figure 3.

forall $v \in V$
    if $v.state = undone$ **then**
        $v.state \leftarrow selected$
        forall $v1 \in v.adjac$
            $v1.state \leftarrow deleted$
$I \leftarrow \{v \in V \mid v.state = selected\}$

Figure 3: Greedy MIS algorithm for the serial construction of an MIS

There is a great deal of flexibility in the order that vertices are chosen of the greedy algorithm. Herein lies a simple opportunity to apply a heuristic, as the first vertex chosen is always *selectable* and the probability is high that vertices which are chosen early are also selectable. Thus, if an application can identify vertices that are "important" then those vertices can be ordered first and so that a less important vertex can not delete a more important vertex. For example, Guillard order the boundary vertices first and ordered them is ascending order of their interior angle in 2D examples [16]. We can now decide that corners are more important than edges and edges are more important than surfaces and so on, and order all corner vertices first, then edges, etc. With this heuristic in place and the basic MIS algorithm in Figure 3 we can guarantees that the number of edge vertices on the coarse grid (in each edge segment) satisfies $\left|V_{edge}^{coarse}\right| \geq \frac{\left|V_{edge}^{fine}\right|-2}{3}$

for 2D meshes, whereas a valid MIS could remove *all* edge and corner vertices from the graph, which can be disastrous (see [1, 2] for numerical experiments).

4.2 Parallel maximal independent set algorithms

We use a partition based parallel MIS algorithm which requires that vertices v are given an immutable data member $v.proc$, the processor number that each vertex is partitioned onto, and a list of adjacent vertices $v.adjac$ [3]. The order in which each processor traverses the local vertex list can be governed by our heuristics although the global application of a heuristic requires an alteration to the MIS algorithm. We add an immutable data member to each vertex v: $v.topo$, the *topological weight* of each vertex. Processor p, in the parallel MIS algorithm, can select a vertex v only if $\{\forall v1 \in v.adjac \mid v1.state \neq deleted\}$:

$$v1.topo < v.topo \;\; or \;\; (v1.topo = v.topo \;\; and \;\; v1.proc \leq v.proc).$$

This test is added to the test in the second line of Figure 3, and results in a correct global implementation of any heuristic that is based on vertex ranking, which can be implemented in the serial MIS by simply ordering the vertices accordingly. To complete the parallel algorithm we simply embed the modified greedy in Figure 3 in an outer loop, see [3] for details and complexity bounds.

4.3 Topological classification of vertices in finite element meshes

Our methods are motivated by the intuition that the coarse grids, of multigrid methods, must represent the geometry of the domain well in order to approximate the function space of the fine mesh well. Note, we define domain in a slightly non-standard way to mean a contiguous region of the finite element problem with a particular material property. Thus, for our discussion, the boundary of the PDE proper is augmented with boundaries between different material types.

The first type of classification of vertices is to find the *exterior* vertices - if continuum elements are used then this classification is trivial. For non-continuum elements like plates, shells and beams, heuristics such as minimum degree could be used to find an approximation to the "exterior" vertices, or a combination of mesh partitioners and convex hull algorithms could be used. For the rest of this section we assume that continuum elements are used and so a boundary of the domains, represented by a list of *facets* or 2D polygons, can be defined. The exterior vertices give us our first vertex classification from the last section: *interior* vertices are vertices that are not exterior vertices. Exterior vertices require further classification, but first we need a method to automatically identify *faces* in our finite element problems.

4.4 A simple face identification algorithm

We want to identify *faces*, or flat regions, of the boundaries in the mesh. To describe our face identification algorithm we assume that a list of facets $facet\_list$ has been created with the boundaries of the finite element mesh. Assume that each facet $f \in facet\_list$ has calculated its unit normal vector $f.norm$, and that each facet f has a list of facets $f.adjac$ that are adjacent to it. With these data structures, and a list with $AddTail$ and $RemoveHead$ functions with the obvious meaning, we can calculate a $face\_ID$ for each facet with the algorithm shown in Figure 4. All facets with the same $face\_ID$ will define one face in our algorithms.

This algorithm simply repeats a breadth first search, of trees rooted at an arbitrary "undone" facet, which is pruned by the requirement that a minimum angle (arccos TOL) be maintained by all facets in the tree relative to the root and its parent. This heuristic is a simple way to identify *faces* (or manifolds that are somewhat "flat") of the boundaries in the mesh.

These faces are useful for two reasons:

- Topological categories for vertices, used in the heuristics of section §4.2, can be inferred from these faces:
  - A vertex attached to only one face is in the interior of a *surface*.
  - A vertex attached to only two faces is in the interior of an *edge*.
  - A vertex attached to more than two faces is a *corner*.

forall $(f \in facet\_list)$ $f.face\_ID \leftarrow 0$
$Current\_ID \leftarrow 0$
forall $f \in facet\_list$
  if $f.face\_ID = 0$
    $list \leftarrow \{f\}$
    $norm \leftarrow f.norm$
    $Current\_ID \leftarrow Current\_ID + 1$
    while $list \neq \emptyset$
      $f \leftarrow list.RemoveHead$
      $f.face\_ID \leftarrow Current\_ID$
      forall $f1 \in f.adjac$ - - TOL is a user selected tolerance $-1 < TOL \leq 1$ (e.g. $TOL = 0.5$)
        if $norm^T \cdot f1.norm > TOL$ and $f.norm^T \cdot f1.norm > TOL$ and $f1.face\_ID = 0$
          $list.AddTail(f1)$

Figure 4: Face identification algorithm

- Vertices, of the same feature class, though not associated with the same feature should not interact with each other in the MIS algorithm.

This first item gives us the classifications that we have discussed above, the second item is discussed in the next section.

4.5 Modified maximal independent set algorithm

We now have all of the pieces that we need to describe the core of our method. First we classify vertices and ensure that a vertex of *lower rank* does not suppress a vertex of *higher rank*. Second we want to maintain the integrity of the "faces" in the original problem as best we can. The motivation for this second criterion can be seen in Figure 5.

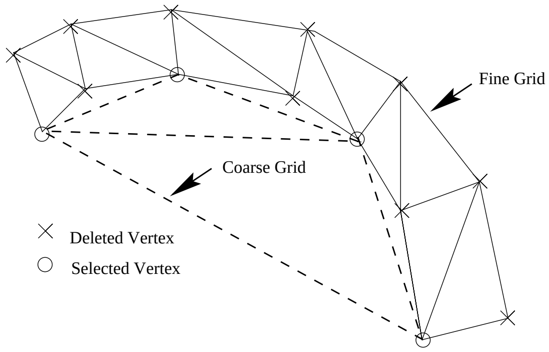


Figure 5: Poor MIS for multigrid of a "thin" body

If the finite element mesh has a thin region then the MIS as described in §4.1 can easily fail to maintain a cover of the vertices in the fine mesh. This comes from the ability of the vertices on one face to decimate the vertices on an opposing face as shown in Figure 5. This phenomenon could be mitigated by randomizing the order that the vertices are added to the MIS, at least within a vertex type. But randomization is not good enough as these skinny regions tend to lower the convergence rate of iterative solvers, and so we need to pay special attention to these thin body features.

The problem in Figure 5, is that vertices are allowed to suppress vertices in the same class - but in a different feature. This problem does not occur on logically square domains as when the grid is coarse enough, for surface vertices to "punch through" the domain, the coarsening stops. On general domains one must continue coarsening, even when one dimension, of some parts of the problem, has coarsened-all-the-way-through, because the problem may still be too large to solve cheaply with a direct solver.

We claim that by *removing all edges between vertices that do not share a common face*, we force the MIS to be a more "logical" and economical representation of the fine mesh (in terms of solver performance). This simple modification (once we have identified faces) will prevent a corner from deleting an edge vertex with which it does not share a face, and likewise and edge vertex from deleting another edge vertex, or a surface vertex, with which it does not share a face. Also, we do not allow corners to be deleted at all; this can be problematic on meshes that have many initial "corners" (as defined by our algorithm) - to mitigate this problem by reclassify vertices on the coarser grids (we generally reclassify the grids above the first two or three).

We are now free to run our MIS algorithm on this modified graph, Figure 6 and 7 shows an example of a possible MIS and remeshing.

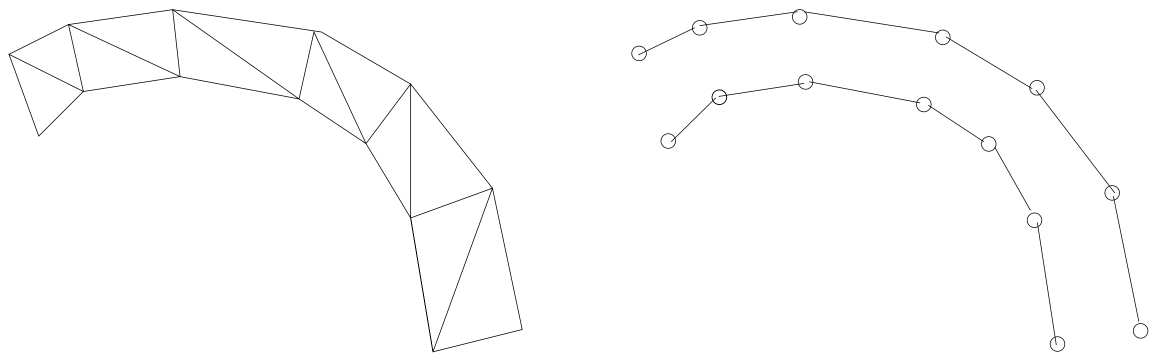


Figure 6: Original and fully modified graph

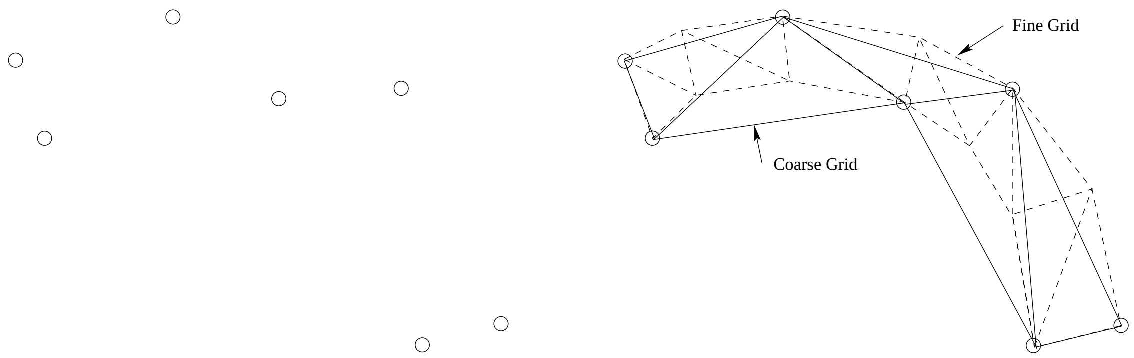


Figure 7: MIS and coarse mesh

4.6 Vertex ordering in MIS algorithm on modified finite element graphs

An additional degree of freedom, in the MIS algorithm, is the order of the vertices within each category. Thus far we have implicitly ordered the vertices by topological category - the ordering within each category can also be specified. Two simple heuristics can be used to order the vertices: random order, and a "natural" order. Meshes may be initially ordered in a block regular order (i.e. an assemblage of logically regular blocks), or ordered in a cache optimizing order like Cuthill-McKee [29]. Both of these ordering types are what we call *natural* orders, and we assume that the "initial" order of our mesh is of this type (if not then we can make it so). The MISs produced from natural orderings tend to be rather dense, random ordering on the other hand tend to be more sparse. That is, the MISs with natural orderings tend to be larger than those produced with random orders. For a uniform 3D hexahedral mesh, the asymptotics of the size of the

MIS is bounded from above by $1/2^3$, and from below by $1/3^3$ as the largest MIS picks every second vertex and the smallest MIS selects every third vertex, in each dimension - natural and random orderings are simple heuristics to approach these bounds.

Small MISs are preferable as there is less work in the solver on the coarser mesh, also fewer levels are required before the coarsest grid is small enough to solve directly, but care has to taken to not degrade the convergence rate of the solver by compromising the quality if the coarse grid representation. In particular, as the boundaries are important to the coarse grid representation it may be advisable to use natural ordering for the exterior vertices and a random ordering for the interior vertices.

4.7 Meshing of the vertex set on the coarse grid

The vertex set for the coarse grid remains to be meshed - this is necessary in order to apply finite element shape functions to calculate the restriction operator. We use a standard Delaunay meshing algorithm to give us these meshes [13]. This is done by putting the mesh inside of a bounding box, thus adding dummy vertices to the coarse grid set, and then meshing this to produce a mesh that covers all fine grid vertices. The tetrahedra attached to the bounding box vertices are removed and the fine grid vertices within these deleted tetrahedra are added to a list of "lost" vertices ($lost\_list$).

We continue to remove tetrahedra, from the mesh, that connect vertices that were not "near" each other on the fine mesh (recall the vertex set are still nested), and that do not have any vertices that lie "uniquely" within the tetrahedron. Define a vertex v to lie uniquely in a tetrahedron, if v lies completely within the tetrahedron and not on its surface, or there is no adjacent tetrahedra to which v can be added. More precisely if a vertex's shape function values are all larger than some small tolerance ϵ (we use only linear shape functions), or there is not an adjacent tetrahedra that can "accept" the vertex, then that tetrahedron is deemed necessary and not removed. We also use a more aggressive phase in which we use a larger, though still small, tolerance, to try to remove more tetrahedra - but the "orphaned" vertices are added to the $lost\_list$. The resolution of the vertices in the $lost\_list$ is discussed in section §4.8.

4.8 Coarse grid cover of fine grid

The final optimization that we would like to employ is to improve the cover of the coarse mesh on the fine grid vertex set. With the coarse grids constructed, the interpolation operators are calculated by evaluating standard finite element element shape functions of the element to which the fine grid vertex is *associated*. Each fine grid vertex is associated with an element on the coarse grid - the element that covers the fine grid vertex. In general however some fine grid vertices (the $lost\_list$ from the previous sections) fail to be covered by the coarse grid as shown in Figure 7. This problem can be solved in one of two ways: find a nearby element and use it (thus extrapolate), or move the vertices on the coarse grid so as to cover all fine grid vertices. We can use the extrapolation of an element that does not cover a fine mesh point, the extrapolation values are simply not all be between zero and one (we use only linear shape functions). Intuition tells us however that interpolation will be of higher quality than extrapolation. Alternatively one can move the coarse grid vertex positions to cover the fine vertices in $lost\_list$.

The optimal coarse grid vertex positions (or an approximation to them) could perhaps be constructed with the use of interpolation theory to provide *cost functions*, and linear or nonlinear programming. We have instead opted for a simple, greedy algorithm that iteratively traverses the exterior vertices of the coarse mesh and applies a simple algorithm to *try* to cover the uncovered vertices that are near it. First we define a list ext_neigh_c, for each coarse grid vertex c, that contains all of the vertices attached to all of the exterior facets to which c is attached; define A to be the cumulative area of these facets ($\sqrt{A}$ is used as a characteristic area of the patch being modified during each vertex move). We define a list $lost\_list_c$ for each coarse grid vertex c, and put the vertices v in $lost\_list$ into the list of the coarse grid vertex to which v is closest; $lost\_list_c$ is then expanded to include the vertices in $lost\_list_i$ for each vertex $i \in ext\_neigh_c$. Given a maximum number of outer iterations M, a maximum distance tol_{max} that a vertex can be move, and a larger tolerance tol_{delete} to prune the list of fine grid vertices that a coarse grid vertex will try to cover. The algorithm is as follows.

- **do** M **times**: **forall** c on the exterior of the coarse grid

- Calculate a vector ϕ: the weighted average of the outward normals of the facets connected to the coarse grid vertex ($c.facet\_list$), weighted by the facet area.
- The normal of each facet that does not have a positive inner product (with this ϕ vector) is added to ϕ until $\{f \in c.facet\_list \mid f.norm^T \cdot \phi < 0\} = \emptyset$, then ϕ is normalized to unit length.
- $\omega \leftarrow 0$
- **forall** $v \in lost\_list_c$: **forall** $f = (a, b, c) \in c.facet\_list$
  * Solve for α in
$$\begin{vmatrix} a.x & a.y & a.z & 1 \\ b.x & b.y & b.z & 1 \\ c.x + \alpha \cdot \phi.x & c.y + \alpha \cdot \phi.y & c.z + \alpha \cdot \phi.z & 1 \\ v.x & v.y & v.z & 1 \end{vmatrix} = 0.0$$
  * if $\alpha > tol_{delete} \cdot \sqrt{A}$ remove v from $lost\_list_c$
  * else if $\alpha > tol_{max} \cdot \sqrt{A}$: $\omega \leftarrow tol_{max}$
  * else if $\alpha > \omega$: $\omega \leftarrow \alpha$
- if $\omega > 0$
  * $c.cood \leftarrow c.cood + \omega \cdot \phi$
  * Recalculate the shape functions for all of fine grid vertices that are associated with an element in a list of elements connected to c ($c.elems$).
  * For all $e \in c.elems$, For all $v \in lost\_list_c$: calculate the shape function for v in element e
    · If all shape values are greater than $-\epsilon$ for some small number ϵ, then
    · Add v to e and remove v from $lost\_list_i$ for all $i \in ext\_neigh_c$

Figure 8 shows an illustration of what our algorithm might do on our running example.

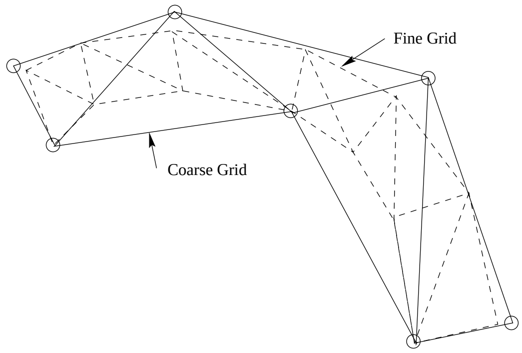


Figure 8: Coarse grid after vertices have been moved to cover all fine grid vertices

Figure 9 shows an example of our methods applied to a problem in 3D linear elasticity. The fine (input) mesh is shown with three coarse grids used in the solution.

See [1] for numerical experiments with (and without) our heuristics on a series of model problems in linear elasticity.

5 Parallel issues

There are two basic issues in the parallelization of our multigrid algorithm: the setup or construction of the restriction operators for our particular algorithm and issues of parallel multigrid methods in general. This section first discusses issues of parallelizing our particular algorithms, and then discusses parallel issues of multigrid solvers in general on todays parallel machines.

Recall the basic steps in our method from section §3:

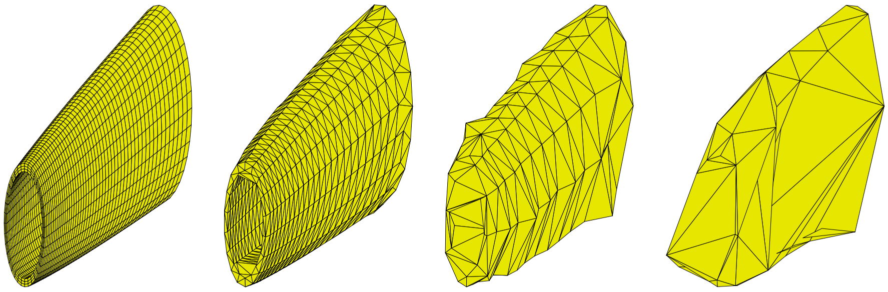


Figure 9: Fine (input) grid and coarse grids for problem in 3D elasticity

- Coarsen the vertex set at the current level (e.g. with an MIS algorithm).
- Mesh this vertex set.
- Use standard finite element shape functions to produce the restriction operators R_i.
- Construct the coarse grid operator: $A_{i+1} \leftarrow R_i A_i R_i^T$.

Parallel maximal independent set algorithms are well documented [3, 22, 18]; the parallel evaluation of the finite element shape functions is trivial; the sparse matrix triple product is difficult to implement efficiently though is a straight forward algorithm [2]. Efficient 3D parallel Delaunay meshing is, however, an open problem - though the 2D problem has been addressed [6]. We are, however, able to avoid computing a complete 3D Delaunay mesh for the coarse grids, as each processor has copies of "ghost" vertices that "surround" all vertices, that a processor is responsible for. We are able to mesh only the local subdomain problem and thus construct the restriction operators locally, though interpolated with non-local (ghost) vertices. Our experiments have, to date, shown little or no adverse effects (on the convergence rate) of our approximate coarse grids. Thus, we are able to avoid the problem of constructing a global (valid finite element) mesh, and make due with locally constructed coarse grids on each processor.

The second class of parallel issues, for our algorithm, are those that relate to general multigrid algorithms. We do not discuss the effective parallel implementation of the components of multigrids as they are all standard linear algebraic operations with sparse matrices and dense vectors - e.g. matrix vector products, matrix triple products, dot products, etc. We, however, do discuss some algorithmic issues that must be addressed for the efficient application of multigrid solvers on large scale problems on typical parallel machines of today.

5.1 Processor subdomain agglomeration

Subdomain agglomeration refers to reducing the number of active processors before a coarse grid is partitioned for the actual solves. Thus, subdomain agglomeration can be seen as a preprocessor to the optimization process in a mesh partitioner; this is required as mesh partitioner are not now sophisticated enough to optimize the "entire problem" and must be used as a tool to optimize communication and load balance - with a given number of processors [21, 17]. Subdomain agglomeration is valuable for two different reasons, first for performance and second for algorithmic considerations. We partition many vertices (i.e. about 10^4) to each processor - this allows all of the processors to do useful work in much, but not all, of the multigrid algorithm - for the "large" problems of today (e.g. 10^7 - 10^8 vertices). The difficulty is that, as the number of vertices per processor dwindles the ability to do work efficiently decreases - this problems will also be come more acute as the problem size increases (and more processors are used). At some point in the grid hierarchy it is *efficient* to let some processors remain idle and agglomerate the work to fewer processors - that is, the time spent on a grid *decreases* if fewer processors are used.

A second reason for the use of processor agglomeration comes from the multigrid algorithm that we use, both in terms of mathematics and practical implementation issues. Mathematically, when any multigrid algorithm uses a block Jacobi preconditioner in the smoother you no longer have the "same solver" in parallel, as on one processor, since one can no longer maintain the same block sizes on the coarser grids (assuming that a serial solver is used on the subdomains).

Agglomeration should take place when the value of the global floating point operations (flop) rate (e.g. Figure 10 (left)), of the *operator* that one wishes to optimize, using the current number of *active* processors, falls below that of a smaller integer number of processors. We need a method to pick the number of processors to use, at each level of multigrid, automatically at run time.

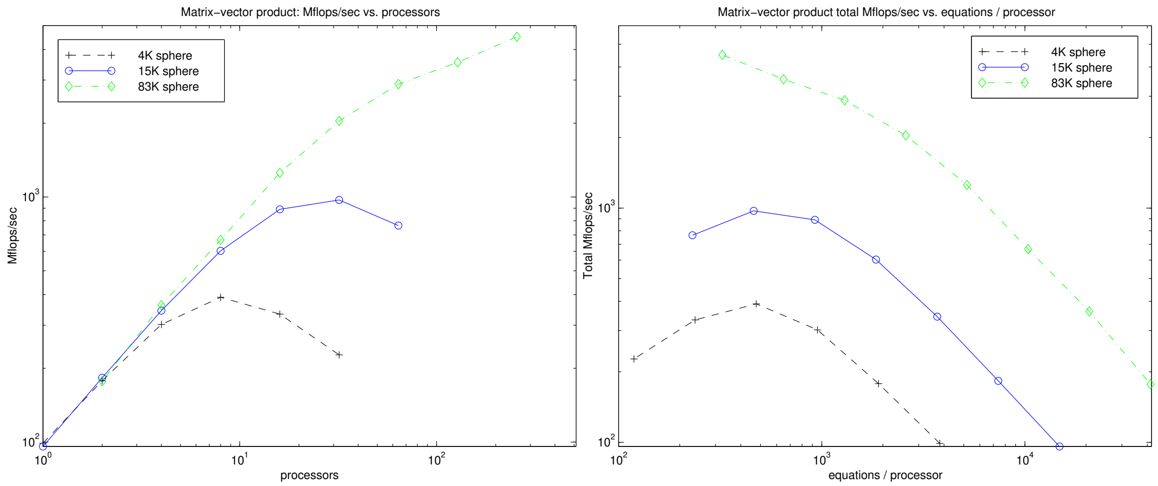


Figure 10: Matrix vector product: per processor, and total, Mflop/sec on a Cray T3E

5.2 Subdomain agglomeration method

There are many approaches that one could take in constructing an algorithm, for picking the number of processors to use at any given level, to optimize the time for each iteration [2]. The complexity of todays parallel computers, and of the multigrid algorithm, relegates us to measuring test problems and curve fitting this data to construct functions that can be evaluated at run time.

We currently work with *empirical* models, based on curve fitting, that use measurements from actual solves, to decide on the number of processors to use on each grid. This method has the disadvantage that model parameters have to be calculated for each machine that the code is to be run on, and may need to be recalculated for different problem types, but it has the advantage that it is reasonably accurate and simple to construct. We can start by looking at the *form* of the function f that we want in Figure 11. The concave shape of the curve, for f, is derived from the intuition that as more processors are in use the $\log(P)$ term in the dot products, and any other source of parallel inefficiency, will push for the use of fewer processors. For convenience we transpose this function to get the convex function $n = g(P)$.

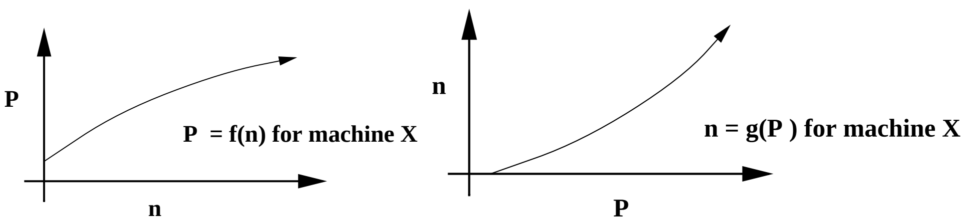


Figure 11: Cartoon of cost function, and its transpose

Now it is natural to *begin* to approximate this function with a quadratic polynomial $n = AP^2 + BP$, or alternatively

$$\frac{n}{P} = A \cdot P + B$$

where $\frac{n}{P}$ is the problem size per processor. A and B are machine and problem dependent parameters that are determined by experimental observation on actual multigrid solves of typical problems, or from experience with other problems on the target machine. We have selected A and B by starting with an initial guess and using a large problem to "search" for the optimal solve time by perturbing A, measuring the total performance (the only quantity that we can effectively measure), and "search" for an "optimal" A; we repeat this process for B, and go back to A, and so on until we find the minimal solve time for the problem.

This method is rather *ad hoc*, though appropriate given the indeterminacy in our current solver implementation. Collecting data, for this type of curve fitting procedure, required that we can either collect reproducible (i.e. consistent) measurements or can collect data and construct statistical averages. Our solver performance is not highly reproducible; we see fluctuation in the total Mflop rate of the solver in a range of about 10%. This is due to, among other things perhaps, the fact that our mesh partitioner is not deterministic, the mesh partitioner does not enforce any particular layout on the machine, and we use machines that are used by other users so that, although we have exclusive use of processors, the communication system is shared and we have no control of the topology of our processor set. Thus, the function that we wish to measure and model (solution times) can not be measured well for use in modeling.

In a production setting one could automate this process for selecting the coefficients, e.g. A and B in equation (1), by running parametric experiments with a large representative problem. One could then simply select the A and B used in the experiment with the fastest solve time, or use curve fitting to construct a function that can be minimized. The use of many more processors would likely require that a higher order polynomial or a more complex function be used. Note, for more accuracy one should also use the number of non-zeros in the matrix in addition to the number of equations n, as this is a more direct measure of the matrix-vector multiply cost, and does *not*, in general, remain constant on all grids. With A and B we can now, at run time, compute the "ideal" number of processors to use:

$$\hat{P} = \frac{-B + \sqrt{B^2 - 4A\frac{n}{P}}}{2A}. \tag{1}$$

After the number of vertices n on a grid has been determined at run time, A, B, and equation (1) are used to compute $\hat{P}$, and then a nearby integer can be selected as the number of processors to use. Note, our current implementation is limited in that each grid must use an integer divisor of number of processors used on the previous (finer) grid.

6 Parallel finite element architecture

Iterative solver development requires constant testing on many types of problems as iterative equation solvers are very sensitive to many features found in the accurate simulation of complex physical systems (e.g. large jumps in material coefficients, thin body domains, complex geometries, and challenging material constitution such as incompressibility). Thus, the development of robust iterative solvers requires testing on challenging problems at all stages of development. We have constructed a parallel finite element system, ParFeap, that is built on a serial research finite element implementation: Finite Element Analysis Program ($FEAP$) [12]. ParFeap provides us with a powerful computational substrate that is invaluable for this research; ParFeap is composed of three basic components:

- **Athena** is a parallel finite element partitioner that uses ParMetis [21] to partition the finite element graph, then constructs a fully valid "serial" finite element problem - to provide a well specified serial finite element problem on each processor.
- **Epimetheus** is an "algebraic" multigrid solver infrastructure that provides a solver to Athena, a driver for Prometheus, an interface to PETSc [5], and numerical primitives not provided by PETSc (e.g. the sparse matrix triple product).
- **Prometheus** is our restriction operator constructor and is the implantation of the coarse algorithms discussed in this paper.

Athena, Epimetheus, and Prometheus are implemented with about 30,000 lines of C++ code, PETSc and ParMetis are implemented in C, and $FEAP$ is implemented in FORTRAN.

6.1 Athena

We have developed a highly parallel finite element code, built on an existing serial research "legacy" finite element implementation. Testing iterative solvers convergence rates requires that challenging test problems be used - problems that test a wide range of finite element techniques. We needed a full featured finite element code, but full featured finite element codes are inherently large, complex, and not easily parallelized. Thus, by necessity, we have developed a domain decomposition based parallel finite element approach, in which a *complete* finite element problem is built on each processor. This abstraction allows for a very simple interface which required only very simple modifications to $FEAP$ (we will denote this modified version of $FEAP$ as $FEAP_p$).

Athena uses ParMetis [21] to calculate a mesh partitioning, and then constructs a fully valid finite element problem for each processor. In addition to the "local" vertices for processor p, prescribed by the vertex partitioning V_p^L, duplicate vertices are added V_p^G (i.e. "ghost" vertices that are required for local computation of residuals and stiffness matrices but are not in V_p^L), define an *extended* vertex set $V_p^E = V_p^L \cup V_p^G$. An extended element set E_p^E for each processor is defined as all elements that touch any vertex in V_p^L (note, the set of vertices in E_p^E is precisely the set V_p^E). Elements can be partitioned in the same manner as vertices to form a set E_p^L of elements to be computed on each processor. In general E_p^L need not be related to E_p^E in any way, though in practice it is natural and most likely optimal to have $E_p^L \subset E_p^E$ and we assume that this is the case. The displacements or Dirichlet boundary conditions must be applied redundantly (i.e. on ghosts) so that elements can compute correct and consistent residuals and element stiffness matrices, whereas the loads (Neumann boundary conditions) must be applied uniquely to maintain the semantics of the problem (as residuals or forces are added into a global vector). We, however, redundantly compute elements so as to avoid communication in the construction of the residual and stiffness matrix; thus, residuals for ghost vertices are ignored and all elements in E_p^E are computed by processor p.

A *slightly* modified serial finite element code runs on each processor. Though the serial code is modified it does not have *any* parallel constructs or knowledge of the global problem - this is useful for debugging and the continued independent development of $FEAP$ (we will denote the serial stand-alone version of FEAP as $FEAP_s$). Parallelism is introduced by providing the finite element code with a matrix and vector assembly routine, solver setup routines, and a solve routine, (additional support functions are provided for expressiveness and performance but are not strictly necessary for many classes of finite element problems). This simple interface could also be adequate for a simple explicit method, where the solver simply needs to invert a diagonal mass matrix and do a component by component vector-vector product - and then *communicate* the solution, on local vertices, to neighbor processor's ghost vertices. Any method that requires other global operations can be added as needed - thus this interface provides the *kernel* for a parallel finite element implementation. The advantage of this method is that the serial finite element code is *completely* parallel - and has a very small interface with Epimetheus.

6.2 Epimetheus

Prometheus provides Epimetheus with restriction operators, and $FEAP_p$ uses Epimetheus to solve a series of linear systems of equations. Epimetheus uses METIS [20] to determine the block Jacobi subdomains and PETSc for the parallel numerical library and programming development environment.

6.3 Prometheus

$FEAP_p$ provides Prometheus with the local finite element problem (that was originally constructed by Athena) i.e. coordinates, element connectivities, material identifiers, and the boundaries conditions. Prometheus constructs the global restriction operators for each grid, and is the core of the algorithmic contribution of this paper. Future work may include adding an interface to the parallel unstructured multilevel grids that we use to construct our coarse grids as these are generally useful for anyone building a parallel "algebraic" multigrid code, or in fact any parallel algorithm on unstructured multilevel grids, similar to how Kelp [4] and Titanium [31] provide parallel multilevel *structured* grid primitives.

Figure 12 shows a graphic representation of the overall system architecture.

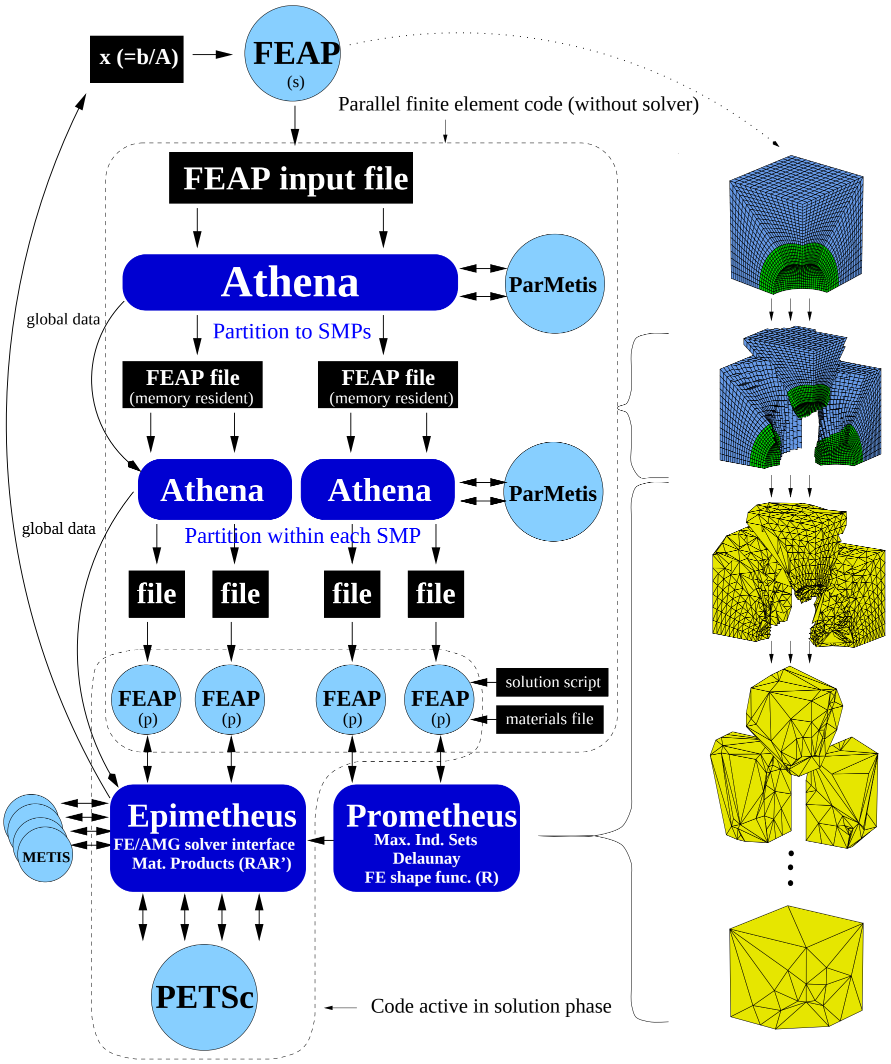


Figure 12: Code Architecture

7 Numerical results

The primary motivation for investigating iterative solvers is to solve challenging large scale problems effectively, we test a problem in large deformation elasticity and plasticity with somewhat complex geometry and material coefficients. Our test problem is of a series if thin concentric spheres enclosed in "soft" material (with symmetric boundary conditions so that only one octant need be modeled). The sphere is constructed of seventeen alternating layers of "hard" and "soft" materials described below. The loading and boundary conditions are an imposed uniform displacement (down), on the top surface. Table 7 shows a summary of the constitution of our two material types.

The hard material is a J_2 plasticity, small deformation material, with a mixed formulation and kinematic hardening [26]. The soft material is a large deformation (Neo-Hookean) hyperelastic material with a mixed formulation [32].

The mesh is parameterized so that we can construct several versions of the problem with different scales of discretization, Figure 7 shows the smallest (base) version of the problem with 80 k (k=1000 degrees of freedom) . Each successive problem has one more layer of elements through each of the seventeen shell layers, with an appropriate (i.e. similar) refinement in the other two directions, and in the outer soft domain - resulting in problems of size: 80 k, 621 k, 2,086 k, 4,924 k, 9,595 k, 16,554 k, and 26,257 k degrees of

Material	Elastic mod.	Poisson ratio	deformation type	yield stress	hardening mod.
soft	10^{-4}	0.49	large	∞	NA
hard	1	0.3	small	0.002	0.002

Table 1: Nonlinear materials

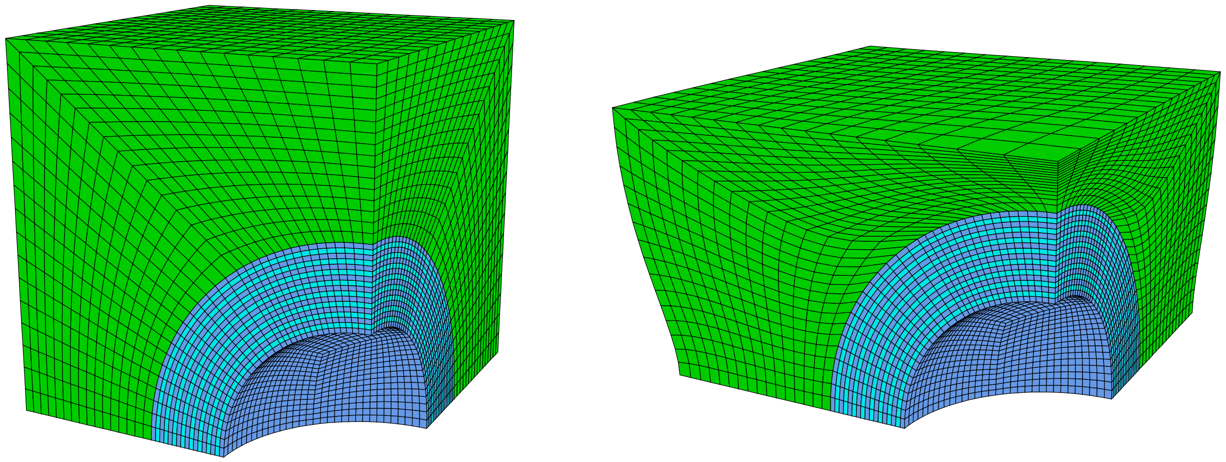


Figure 13: 79,679 dof concentric spheres problem

freedom.

7.1 Linear and nonlinear solver

We use a full Newton nonlinear solution method. Convergence is declared when the energy norm of the correction is 10^{-16} that of the first correction. This means in Newton iteration m, we declare convergence when $\left|x_m^T \cdot (b - Ax_m)\right| < 10^{-16} \cdot \left|x_0^T \cdot (b - Ax_0)\right|$. Our linear solver, within each Newton iteration, is conjugate gradient preconditioned by our multigrid solver, with a block Jacobi preconditioned conjugate gradient smoother. We use 6 blocks for every 1,000 unknowns in the block Jacobi preconditioner.

$FEAP_p$ calls our linear solver at each Newton iteration, with the current residual $r_m = b - Ax_m$, thus the linear solve is for the increment $\Delta x \approx A^{-1} r_m$. We use a dynamic convergence tolerance ($rtol$) for the linear solve in each Newton iteration of: $rtol_1 = 10^{-4}$ in the first iteration, $rtol_2 = 10^{-3}$ in the second iteration, and $rtol_m = min(10^{-3}, \frac{\|r_m\|}{\|r_{m-1}\|} \cdot 10^{-1})$ on all subsequent iterations ($m > 2$). This heuristic is intended to minimize the number of total iterations required in the Newton solve for each time step by only solving each linear equation set to the degree that it "deserves". That is, if the true (nonlinear) residual is not converging quickly then solving the linear system to an accuracy far in excess of the reduction in the residual, that is expected in the outer Newton iteration, is not likely to be economical.

The reason for hardwiring the tolerance for the second Newton iteration, is that the residual for the first iteration of this particular problem tends to drop by about three orders of magnitude. The second step of this problem tends to have the residual reduced by about one order of magnitude or less and then continues with super-linear, but not quadratic convergence rate (as we use a non-exact solver). Our dynamic tolerance heuristic ($min(10^{-3}, \frac{\|r_m\|}{\|r_{m-1}\|} \cdot 10^{-1})$) specifies too small of a tolerance, on the second iteration of this problem, so we hardwired the tolerance for the sake of efficiency.

7.2 Cray T3E

We run each linear solve with about 41,000 degrees of freedom per processor, on 2 to 640 processors; thus, the problems range in size from 80,000 to over 26 million degrees of freedom. For these experiments we use a convergence tolerance of 10^{-4} (the first linear solve tolerance in the nonlinear solver), so as to investigate the efficiency of one linear solve, in isolation of the issue of nonlinear performance discussed in section §7.3. We want to show our solver in its best light by running with as many equations per processor as possible, as parallel efficiency will in general increase as the number of degrees of freedom per processor goes up. These experiments are performed on a Cray T3E with 640 single 450 MHz. processors, 900 Mflop/sec theoretical peak, 256 MB memory per processor, and a peak Mflop rate of 662 Mflop/sec (1/2 of 2 processor Linpack R_{max}). Figure 14 (left) shows the times for the major subcomponents of the solver, and Figure 14 (right) shows the efficiency diagram of the same data.

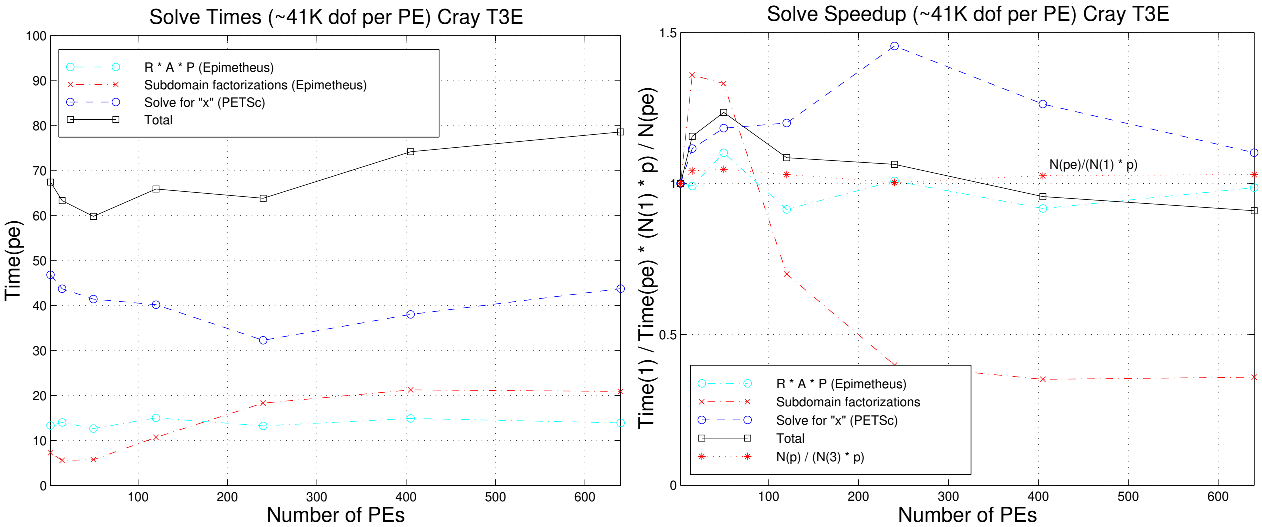


Figure 14: 41,000 dof per processor, included concentric sphere times (left), efficiency (right), on a Cray T3E

Processors	2	15	50	120	240	405	640
Number of dof	79,679	622,815	2,085,599	4,924,223	9,594,879	16,553,759	26,257,055
Number of iterations	22	22	19	17	14	14	15

Table 2: Number of iterations for first linear solve in the nonlinear solve

We see super-linear efficiency in the solve times, in Figures 14, for two reasons. First, we have super-linear convergence rates, as shown in Table 7.2. Second, the vertices added in each successive scale problem have a higher percentage of *interior* vertices than the base (two processor) problem, leading to higher rates of vertex reduction in the coarse grids. This is because, as the number of unknowns n increases, the "surface area" increases by $n^{\frac{2}{3}}$ whereas the interior increases by n; thus, the ratio of interior vertices to surface vertices increases as the scale of discretization decreases (n increases) on the larger problems. Our heuristics (section §4) try to articulate the surfaces (boundary and material interfaces) well, resulting in a higher ratio of surface vertices being promoted to the coarse grid. Thus, the *rate* of vertex reduction is higher on the larger problems as they have proportionally more interior vertices, hence proportionally larger reduction rates leading to less work per processor per fine grid vertex on the large problems, as can be seen in Figure 15 (left). For instance, the total reduction factor of the first coarse grid is about seven on the base case (80 k dof problem) and about thirteen on the larger problems. Therefore, there are in fact fewer flops in each full multigrid iteration - per fine grid vertex - on the coarse grids for the larger problems.

Figures 14 also show that "subdomain factorizations" perform poorly. On inspection of the PETSc output we see that the copying of the subdomain data, from the grid stiffness matrix, and the symbolic factorizations of these submatrices are the cause of these large and increasing (with number of processors) times. We are not able to explain the poor performance in this data, though data on problems with fewer equations per processor, show much smaller times that are in line with our expectations. The Cray does not have virtual memory and so paging is not an issue; thus, we suspect that this poor performance is due

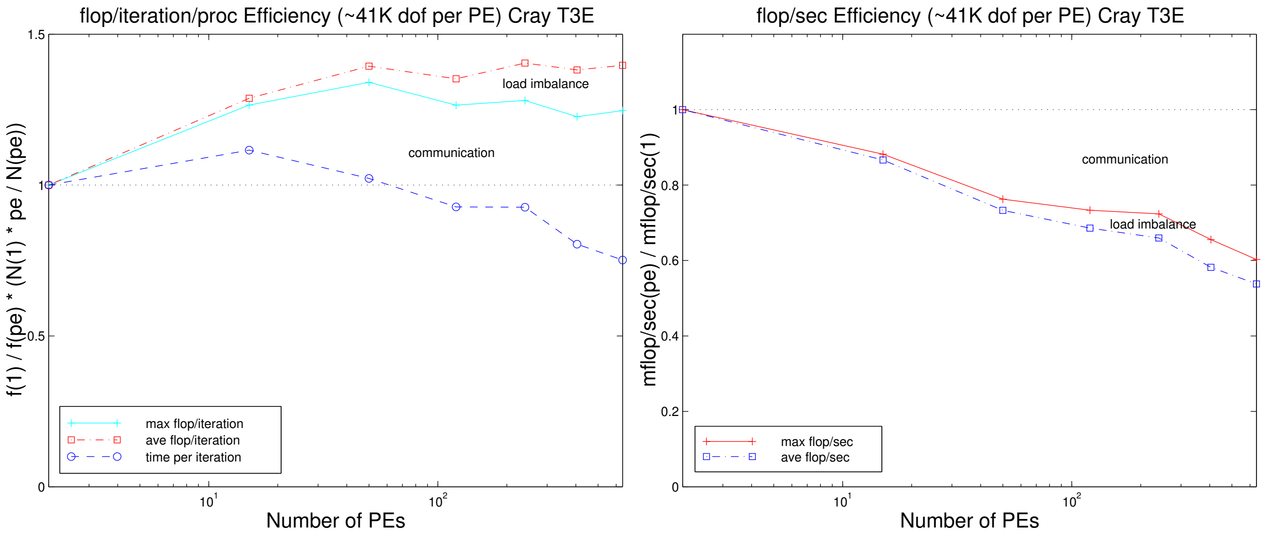


Figure 15: 41 k dof per processor, concentric sphere, flop efficiency on a Cray T3E

to cache effects as there is no need for communication in the subdomain factorization phase, hence the 25% parallel efficiency in Figure 14 (right) is not well understood.

This data shows that the overall solution times are staying about constant. Additionally, the base case (2 processor case) solve ran at 144 Mflops, and the 640 processor case ran at 24,792 Mflops (about 54% efficiency), in the actual solve.

7.3 IBM PowerPC cluster

The full nonlinear problem has run five time steps that result in a vertical displacement of 3.6 inches down; the cube is 12.5 inches on a side, and the top "soft" section is 5 inches at the central (z) axis. We keep about 80,000 degrees of freedom per processor, and run problems of size 80 k (on 1 processors) up to 26,257 k (on 336 processors, 84 nodes). Figure 16 show the number of multigrid iterations, stacked on one another to show the total number of multigrid iterations to solve each problem.

From this data we can see that the total number of iterations is staying about constant as the scale of the problem increases. Figure 14 (left) shows that the number of iterations, to reduce the residual by a fixed amount, in the first solve of the first time step, *decreases* as the problem size increases. Thus, the data in Figure 16 suggests that the nonlinear problem is getting *harder* to solve as the discretization is refined; this is not a surprising result as there is likely more nonlinear behavior in the finer discretizations, but more work remains to investigate this issue further.

Figure 17 show the total "wall-clock" times. From this we can see that the overall time to solve a problem, after the problem as been constructed, is scaling well, and that the parallel partitioning and multigrid setup time is relatively modest when nonlinear problems are being solved (i.e. the setup costs are amortized by the many linear solves in nonlinear time stepping problems)

8 Conclusion

We have developed a promising method for solving the linear set of equations arising from implicit finite element applications in solid mechanics. Our approach, a 3D and parallel extension to an existing serial 2D algorithm, is to our knowledge unique in that it is a fully automatic (i.e. the user need only provide the fine grid, which is easily available in most finite element codes) standard geometric multigrid method for unstructured finite element problems.

We have developed a fully parallel and portable prototype solver that shows promising results, both in terms of convergence rates and parallel efficiency, for some modestly complex geometries with challenging

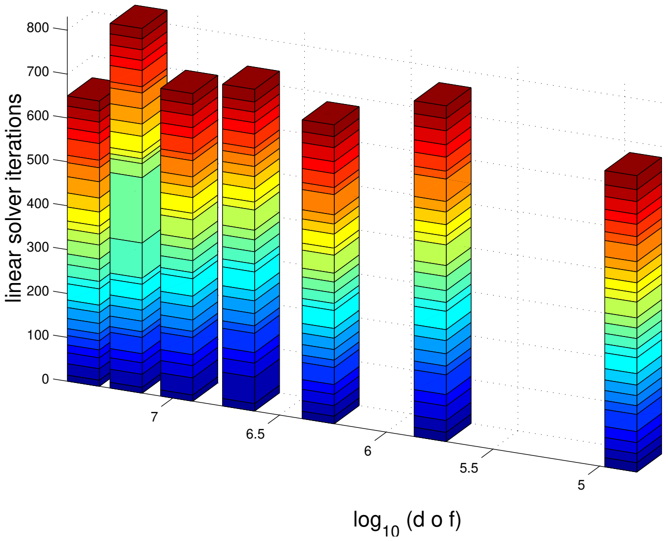


Figure 16: Multigrid iterations per Newton iteration

materials in large deformation elasticity and plasticity. Our prototype has solved problems of up to 26.3 million degrees of freedom on a Cray T3E with up to 640 processors, and on an IBM PowerPC cluster with up to 84 4-way SMP nodes processors. We have also developed agglomeration strategies for the optimal selection of active processor sets on the coarse grids of multigrid, as this is essential for optimal scalability.

We have also developed a highly parallel finite element implementation, built on an existing state-of-the-art serial research "legacy" finite element implementation that uses our parallel solver. By necessity we have developed a, domain decomposition based parallel finite element method, that builds a complete finite element problem on each processor as its primary abstraction.

The implementation of our prototype system (ParFeap) required about 30,000 lines of our own C++ code, plus several large packages: PETSc (160,000 lines of C), FEAP (105,000 lines of FORTRAN), METIS/ParMetis (30,000 lines of C), and geometric predicates (4,000 lines of C) [25]. This complexity was required because

- efficient parallel multigrid solvers for unstructured meshes are inherently complex,
- algorithm development and verification of success in this area is highly *experimentally* based, and thus requires a flexible full featured computational substrate to enable this type of research, and
- we require portable implementations, so we use explicit message passing (MPI), as our parallel programming paradigm, while accommodating both "flat" and hierarchical memory architectures (clusters of SMPs, CLUMPs).

Additionally this work is rather broad, in the sense that the emphasis has been on getting a prototype of the best finite element linear equation solver possible - this has limited the amount of time that could be devoted to investigating more optimized approaches to each aspect of our algorithm. In fact the vary exploratory nature of this work *demands* a non-optimal implementation, in that there is no sense in optimizing any system (e.g. a parallel linear solver for unstructured finite element problems) before the overall practical quality of the algorithm and a particular application area have been determined. Thus, there is much more work to be done.

To be useful to a more general finite element community we need to extend the algorithms and features. Some areas of future work are: investigate more sophisticated face identification algorithms, to increase robustness of solver on arbitrary complex domains; incorporate a parallel Delaunay tessellation algorithm [6] so as to develop more robust and globally consistent implementations; extend the implementation for more element types: shells, beams, trusses, etc.; accommodate higher order elements such as, supporting higher dof

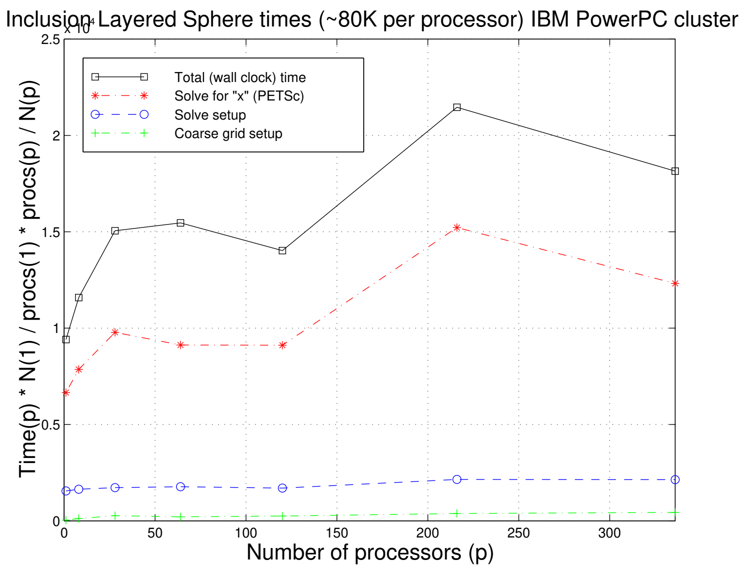


Figure 17: End to end times of nonlinear solve with 80,000 dof per processor on IBM

per vertex for p-adaptive methods and multi-physics problems. To extend the depth of problems that we can address, such as, investigate non-CG Krylov subspace methods for indefinite systems from large deformation elasticity and plasticity, or regularization of these problems; and develop parallel and preconditioned Uzawa solvers for indefinite systems from constrained problems with Lagrange multipliers, or develop multigrid methods to address these problems.

Acknowledgments. We would like to thank the many people that have contributed libraries to this work: Robert L. Taylor for providing FEAP, The PETSc team for providing PETSc, George Karypis for providing ParMetis. I would also like to thank Steve Ashby and the ALE3D group at LLNL for proving support. Lawrence Livermore National Laboratory for providing access to its computing systems and to the staff of Livermore Computing for their support services. Lawrence Berkeley National Laboratory for the use of their Cray T3E, and their very helpful support staff - this research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Energy Research of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098. This research was supported in part by the Department of Energy under DOE Contract No. W-7405-ENG-48 and DOE Grant Nos. DE-FG03-94ER25219, DE-FG03-94ER25206.

References

[1] Mark Adams. Heuristics for the automatic construction of coarse grids in multigrid solvers for finite element matrices. Technical Report UCB//CSD-98-994, University of California, Berkeley, 1998.

[2] Mark Adams. *Multigrid Equation Solvers for Large Scale Nonlinear Finite Element Simulations*. Ph.D. dissertation, University of california, Berkeley, 1998. Tech. report UCB//CSD-99-1033.

[3] Mark Adams. A parallel maximal independent set algorithm. In *Proceedings 5th copper mountain conference on iterative methods*, 1998. Best student paper award winner.

[4] S. B. Baden. Software infrastructure for non-uniform scientific computations on parallel processors. *Applied Computing Review, ACM*, 4(1):7–10, Spring 1996.

[5] S. Balay, W.D. Gropp, L. C. McInnes, and B.F. Smith. PETSc 2.0 users manual. Technical report, Argonne National Laboratory, 1996.

[6] Guy Blelloch, Gary L Miller, and Dafna Talmor. Design and implementation of a practical parallel Delaunay algorithm. *To appear in Algorithmica*, 1998.

[7] W. Briggs. *A Multigrid Tutorial.* SIAM, 1987.

[8] V.E. Bulgakov and G. Kuhn. High-performance multilevel iterative aggregation solver for large finite-element structural analysis problems. *International Journal for Numerical Methods in Engineering*, 38, 1995.

[9] Tony F. Chan and Barry F. Smith. Multigrid and domain decomposition on unstructured grids. In David F. Keyes and Jinchao Xu, editors, *Seventh Annual International Conference on Domain Decomposition Methods in Scientific and Engineering Computing.* AMS, 1995. A revised version of this paper has appeared in ETNA, 2:171-182, December 1994.

[10] James Demmel. *Applied Numerical Linear Algebra.* SIAM, 1997.

[11] M.C. Dracopoulos and M.A Crisfield. Coarse/fine mesh preconditioners for the iterative solution of finite element problems. *International Journal for Numerical Methods in Engineering*, 38:3297–3313, 1995.

[12] FEAP. http://www.ce.berkeley.edu/∼ rlt.

[13] D. A. Field. Implementing Watson's algorithm in three dimensions. In *Proc. Second Ann. ACM Symp. Comp. Geom.*, 1986.

[14] J. Fish, V. Belsky, and S. Gomma. Unstructured multigrid method for shells. *International Journal for Numerical Methods in Engineering*, 39:1181–1197, 1996.

[15] J. Fish, M. Pandheeradi, and V. Belsky. An efficient multilevel solution scheme for large scale non-linear systems. *International Journal for Numerical Methods in Engineering*, 38:1597–1610, 1995.

[16] Herve Guillard. Node-nested multi-grid with Delaunay coarsening. Technical Report 1898, Institute National de Recherche en Informatique et en Automatique, 1993.

[17] Bruce Hendrickson and Robert Leland. A multilevel algorithm for partitioning graphs. In *Proc. Supercomputing '95*, 1993. Formerly, Technical Report SAND93-1301 (1993).

[18] Mark T. Jones and Paul E. Plassman. A parallel graph coloring heuristic. *SIAM J. Sci. Comput.*, 14(3):654–669, 1993.

[19] S. Kacau and I. D. Parsons. A parallel multigrid method for history-dependent elastoplacticity computations. *Computer methods in applied mechanics and engineering*, 108, 1993.

[20] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, To appear.

[21] George Karypis and Kumar Vipin. Parallel multilevel k-way partitioning scheme for irregular graphs. *Supercomputing*, 1996.

[22] M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comput.*, 4:1036–1053, 1986.

[23] D.R.J. Owen, Y.T. Feng, and D Peric. A multi-grid enhanced GMRES algorithm for elasto-plastic problems. *International Journal for Numerical Methods in Engineering*, 42:1441–1462, 1998.

[24] J. Ruge. AMG for problems of elasticity. *Applied Mathematics and Computation*, 23:293–309, 1986.

[25] Jonathan Richard Shewchuk. Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. *Discrete & Computational Geometry*, 18(3):305–363, October 1997.

[26] J.C. Simo and T.J.R. Hughes. *Computational Inelasticity.* Springer-Verlag, 1998.

[27] Barry Smith, Petter Bjorstad, and William Gropp. *Domain Decomposition.* Cambridge University Press, 1996.

[28] Dafna Talmor. *Well spaced points for numerical methods.* PhD thesis, Carnegie Mellon University, Pittsburgh PA 15213-3891, 1997.

[29] Sivan Toledo. Improving memory-system performance of sparse matrix-vector multiplication. In *Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing*, March 1997.

[30] P. Vanek, J. Mandel, and M. Brezina. Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems. In *7th Copper Mountain Conference on Multigrid Methods*, 1995.

[31] Yelick, Semenzato, Pike, Miyamoto, Liblit, Krishnamurthy, Hilfinger, Graham, Gay, Colella, and Aiken. Titanium: A high-performance Java dialect. In *Workshop on Java for High-Performance Network Computing, Stanford, California*, February 1998.

[32] O.C. Zienkiewicz and R.L. Taylor. *The Finite Element Method*, volume 1. McGraw-Hill, London, Fourth edition, 1989.