Parallel Multigrid Solver for 3D Unstructured Finite Element Problems

Mark Adams [*] James W. Demmel [†]

Abstract

Multigrid is a popular solution method for the system 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) as a mechanism 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 parallel algorithms, based on geometric heuristics, to optimize the quality of MISs and the meshes constructed from them, for use in multigrid solvers for 3D unstructured problems. We conduct scalability studies that demonstrate the effectiveness of our methods on a problem in large deformation elasticity and plasticity of up to 40 million degrees of freedom on 960 processor IBM PowerPC 4-way SMP cluster with about 60% parallel efficiency.

Key words: unstructured multigrid, parallel sparse solvers, parallel maximal independent sets

1 Introduction

This work is motivated by the success of the finite element method in effectively simulating complex physical systems in science and engineering, coupled with the wide spread availability of ever more powerful parallel 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 their solve time is unaffected by the condition number of the matrix. However, direct methods possess sub-optimal time and space complexity, as the scale of the problems increase, when compared to iterative methods. Thus, as larger and faster computers become more widely available, the use of iterative methods is becoming increasingly attractive.

Multigrid is one of a family of optimal multilevel domain decomposition methods [23], and is known to be an effective method to solve finite element matrices [12, 15, 25, 5, 8, 18]. The general application of multigrid to unstructured meshes, which 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 scalable algorithms for unstructured finite element problems that can be easily integrated with existing finite element codes (ie, requiring only data that is easily available in most finite element applications), is an open problem. This paper discusses one promising approach to the development of scalable and modular linear equation solvers for unstructured finite element problems; a more detailed presentation can be found in [1].

This paper proceeds as follows: Section §2 briefly introduces multigrid; section §3 introduces our basic algorithm; and section §4 describes our parallel methods to optimize the algorithm for finite element problems. Parallel finite element and multigrid algorithmic issues are discussed in section §5 and performance measures are discussed 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 almost 40 million degrees of freedom on 240 4-way SMP IBM PowerPC nodes (with about 60% parallel efficiency). We conclude in section §8 with potential directions for future work.

[*] Department of Computer Science, University of California Berkeley, Berkeley CA 94720 (madams@cs.berkeley.edu). This work is supported by DOE grant No. W-7405-ENG-48

[†] Department of Computer Science, University of California Berkeley, Berkeley CA 94720

2 Multigrid

Multigrid is known to be the asymptoticly optimal solution method for the discrete Poisson equation in serial. The FFT is competitive with multigrid in parallel [7], however, unlike the FFT multigrid has been applied to unstructured second order finite element problems in elasticity [20, 6] and plasticity [12, 15, 18], as well as fourth order finite element problems [11, 25].

Simple (and inexpensive) iterative methods like Gauss-Seidel, Jacobi, and block Jacobi [7] are effective at reducing the high frequency error, but are ineffectual in reducing the low frequency error. These simple solvers are called *smoothers* as they "smooth" the error (actually they reduce high *energy* components, 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 (coarse) 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 approximate solution. This coarse grid correction is then added to the current solution. Thus, the goal of a multigrid method is to construct, and compose, a series of function spaces in which iterative solvers and small direct solves can work together to economically reduce the entire spectrum of the error.

Figure 1 shows the multigrid *V-cycle*, using a smoother $S(A, b)$, restriction operator R_{i+1} that maps residuals from the fine grid i to the next coarse grid $i + 1$, and prolongation operator R^T_{i+1} to map corrections from coarse grid $i + 1$ to fine grid i.

```
function $MGV(A_i, r_i)$
    if there is a coarser grid
        $x_i \leftarrow S(A_i, r_i)$                                  // pre-smooth
        $r_i \leftarrow r_i - Ax_i$                                   // compute residual
        $r_{i+1} \leftarrow R_{i+1}(r_i)$                             // restrict residual to coarse grid
        $x_{i+1} \leftarrow MGV(R_{i+1}A_iR^T_{i+1}, r_{i+1})$        // compute coarse grid correction
        $x_i \leftarrow x_i + R^T_{i+1}(x_{i+1})$                     // prolongate coarse grid correction
        $r_i \leftarrow r_i - A_ix_i$                                 // compute residual
        $x_i \leftarrow x_i + S(A_i, r_i)$                            // post-smooth
    else
        $x_i \leftarrow A_i^{-1}r_i$                                  // solve coarsest problem directly
    return $x_i$
```

Figure 1: Multigrid *V-cycle* Algorithm

Multigrid algorithms compute an approximate coarse grid correction, and then smooth the remaining error; the *V-cycle* adds a pre-smoothing step to symmetrize the operator. Many multigrid algorithms have been developed. Figure 1 shows one iteration of "multiplicative" multigrid; we use the "full" multigrid algorithm (FMG) [4], in our numerical experiments. One full multigrid cycle applies the *V-cycle* to each grid, starting with the coarsest grid, then adds the result to the current solution, projects the new solution to the next finer grid, computes the residual, applies the *V-cycle* to this finer grid, and so on until the finest grid is reached.

3 Our method

Given a smoother, the only operators required by multigrid (Figure 1) are the restriction operators, the rows of which define the coarse grid spaces. These coarse grid spaces can be constructed algebraically [25] or geometrically - we employ a geometric approach. Traditionally geometric approaches have required that the user provide the coarse grids [12, 15, 18]; requiring coarse meshes may be an onerous responsibility for the solver to place on the user, and thus we wish to do this automatically within the solver. We build on a 2D serial algorithm first proposed by Guillard [13] and independently by Chan and Smith [6]. The purpose of this algorithm is to automatically construct a coarse grid from a finer grid for use in standard multigrid algorithms. A high level view of the algorithm, at each level, is as follows:

- The vertex set at the current level (the "fine" mesh) is *evenly* coarsened, using a maximal independent set (MIS) algorithm (§4.1) to produce a much smaller subset of vertices.

- The new vertex set is automatically remeshed with tetrahedra.
- Standard linear finite element shape functions for tetrahedra are used to produce the restriction operator (R). The transpose of the restriction operator is used as the prolongation operator.
- The restriction operator is then used to construct the (Galerkin) coarse grid operator from the fine grid operator: $A_{coarse} \leftarrow RA_{fine}R^T$.

This method is applied recursively to produce a series of coarse grids, and their attendant operators, from a "fine" (application provided) grid.

The coarse grid operators can be formed in one of two ways - either algebraically to form a Galerkin coarse grid ($A_{coarse} \leftarrow RA_{fine}R^T$), or by creating a new finite element problem on each coarse grid and letting the finite element implementation construct the matrices. The algebraic method has the advantage that it places less demand on users by not requiring that they construct the coarse grid operators, thus allowing for modular software design. The algebraic approach also has the advantage that strain localizations, in nonlinear material problems, influence the coarse grid operators, thereby potentially providing better operators.

An additional reason for constructing the coarse grid operators algebraically is that mesh generators, be they automatic or semi-automatic, are not accustomed to approximating the domain automatically (ie, not strictly maintaining the topology of the domain) which is often required for efficiency - especially on the coarsest grids of large problems with complex geometry. Thus, standard mesh generators may not be optimal on the coarsest grids of large complex problems as they may not be able to reduce the complexity of the problems (as they are constrained to fully representing the geometry of the problem) to the degree that the coarse grid can be solved efficiently with a direct solver. Because of the difficulty in generating the coarse grids, the quality of the coarse grids - as a finite element mesh - may be poor especially on the coarsest grids (as can be seen from our coarse grids, §7) which can lead to robustness problems if the finite element application is required to operate on these coarse grids.

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.

4 Automatic coarse grid creation with unstructured meshes

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 eigenfunctions) of the previous grid. Our algorithms, introduced above and described below, can be viewed as attempting to approximate the geometry of the problem, to a uniform degree, on each coarse grid so as to approximate the eigenvectors efficiently in a general purpose (non-operator specific) way.

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 (ie, $\forall v, w \in I, (v, w) \notin E$); 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 [17, 14], is show in Figure 2, in which we provide vertices with a state variable which is initialized to the "undone" state.

```
forall $v \in V$
    if $v.state = undone$ then
        $v.state \leftarrow selected$
        forall $v1 \in v.adjac$        // $v.adjac$ is a list of vertices adjacent of $v$ in $G$
            $v1.state \leftarrow deleted$
$I \leftarrow \{v \in V \mid v.state = selected\}$
```

Figure 2: Greedy MIS algorithm for the serial construction of an MIS

4.2 Parallel maximal independent set algorithms

We use a partition based parallel MIS algorithm which requires that vertices v be given an data member $v.proc$, the unique processor number that each vertex is assigned to, and a list of adjacent vertices $v.adjac$ [2]. 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: $v.rank$. In the parallel MIS algorithm, processor p can select a vertex v only if all $v1 \in v.adjac$ are deleted (ie, $v1.state = deleted$) or

$$v.rank > v1.rank \ \ or \ \ (v.rank = v1.rank \ \ and \ \ v.proc \geq v1.proc).$$

This test is added to the test in the second line of Figure 2, and results in a correct global implementation of any heuristic that is based on vertex ranking. To complete the parallel algorithm we simply embed the modified greedy algorithm in Figure 2 in an outer loop and send appropriate data to other processors on distributed memory machines (see [2] for details). We use "topological classification" to compute vertex ranks in our algorithm as described below.

4.3 Topological classification of vertices in finite element meshes

Our methods are motivated by the intuition that the coarse grids of multigrid methods should represent the geometry of the domains well in order to approximate the lower eigenvectors well, and hence be effective in multigrid solvers. We define a domain as a contiguous region of the finite element problem with a particular material property. We rank vertices with classifications derived from geometric features.

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 domain represented by a list of *facets* can be defined and easily constructed. The exterior vertices give us our first vertex classification from the last section: *interior* vertices are vertices that are not exterior vertices. We further classify exterior vertices, but first we need a method to automatically identify *faces* in our finite element problems, from which we can construct features, and then define vertex classifications.

4.4 A simple face identification algorithm

We want to identify *faces*, or "flat" regions, of the boundaries in the mesh; features can then be naturally constructed from these faces. Assume that a list of facets $facet\_list$ has been created from all of the element facets that are on a boundary of the problem (these include boundaries between material types). 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 3. All facets with the same $face\_ID$ define one face.

This algorithm simply repeats a breadth first search of trees rooted at an arbitrary "undone" facet, which is terminated by the requirement that a minimum angle (arccos TOL) be maintained by all facets in the tree with the root and with its neighbors. This is a simple algorithm to partition the boundaries of the mesh into faces (or manifolds that are somewhat "flat"). These faces are used in two ways:

1. Topological categories for vertices, used in the heuristics of section §4.2, can be inferred from these faces:
   - A vertex attached to exactly one face is a *surface* vertex.
   - A vertex attached to exactly two faces is an *edge* vertex.
   - A vertex attached to more than two faces is a *corner* vertex.
2. Feature sets of vertices can be constructed from these faces, eg, an edge is the set of all vertices that touch the same two, and only two, faces. These feature sets are used to modify the graph that is used in the MIS algorithm so as to insure that vertices of the same feature class, though not in the same feature, do not interact with each other in the MIS algorithm; section §4.6 discusses the reasons for this criteria.

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

Figure 3: Face identification algorithm

Item 1) gives us the classifications that we have discussed above, item 2) is discussed in the §4.6. We can now assign vertex ranks ($v.rank$ in §4.2) as $rank = 0$ for interior vertices, $rank = 1$ for surface vertices, $rank = 2$ for edge vertices, and $rank = 3$ for corner vertices.

4.5 Parallel face identification algorithm

To parallelize this algorithm (Figure 3), each processor p constructs a list E_p of all elements that touch any vertex that it is responsible for (given by a vertex partitioning onto processors); vertices in elements of E_p that are not in processor p's vertex set are called "ghost" vertices. A list of facets F_p on processor p is generated from E_p; F_p is pruned by eliminating facets with all ghost vertices. Also, $Current\_ID$ (in Figure 3) is really a tuple $\langle p, Current\_ID\rangle$ so that each $face\_ID$ that a processor creates is unique.

Each processor p waits to receive "seed" facets from all processors that are responsible for ghost vertices on facets in F_p and that have a higher processor number; these seed facets are used in the local algorithm (Figure 3). A processor with the highest processor number (or with no higher neighbor) starts the process by simply running the algorithm in Figure 3 and then sending its ghost facets f, along with the $f.face\_ID$ and the "root" normal ($root\_norm$ in Figure 3), to all processors (with a lower processor number) that own a vertex on the facet - these become the seed facets for the lower processors. Seed facets f that are given a new $f.face\_ID$ in the course of the algorithm generate an edge in a $face\_ID$ graph G_{fid} between the old and the new $f.face\_ID$. A global reduction is performed at the end of the algorithm so that all processors have a copy of G_{fid} (note, this is not a scalable construct but the constants are very small). All facets f are then given the largest $face\_ID$ that $f.face\_ID$ can reach in G_{fid}.

This algorithm does not preserve the semantics of the serial algorithm (ie, the resulting faces are not guaranteed to be faces that the serial algorithm could have produced) but the results are close enough for the current development of our solver since we do not see deterioration in convergence rates with the use of multiple processors. Future work may include tightening up the semantics of the parallel algorithm as well as investigating more sophisticated serial algorithms.

4.6 Modified graphs for maximal independent sets

We now have all of the pieces that we need to describe the core of our method. First we classify vertices, to generate vertex ranks, 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 the 2D example in Figure 4. 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 4. This phenomenon could be mitigated by

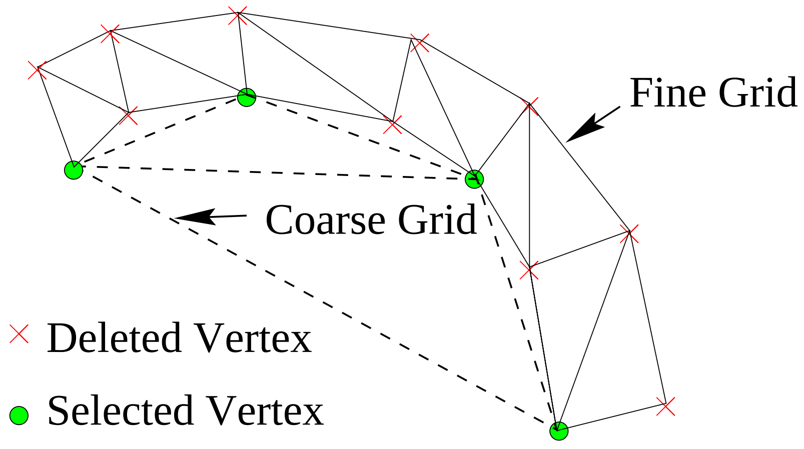


Figure 4: Poor MIS for multigrid of a “thin” body

randomizing the order that the vertices are added to the MIS, but thin regions tend to lower the convergence rate of iterative solvers, and so we want to pay special attention to them.

The problem (in Figure 4) is that vertices are allowed to suppress vertices in the same feature class (eg, edges) - but in a different feature. This problem does not occur on logically square domains because when the grid is coarse enough for elements to “punch through” the domain the coarsening stops. On general domains, however, 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.

Our simple modification (once we have identified faces) is to delete edges connecting nodes that do not share a face; this prevents a corner vertex from deleting an edge vertex with which it does not share a face and surface vertices from deleting surface vertices on different surfaces. 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); we mitigate this problem by reclassifying vertices on the coarser grids. We generally reclassify the third and subsequent grids (ie, we let the second grid vertices retain the type of the fine grid vertex from which it was derived). Note, this heuristic is problem dependent and should be a user defined parameter.

Figure 5 shows the problem in Figure 4 and the modified graph with edges removed as described above.

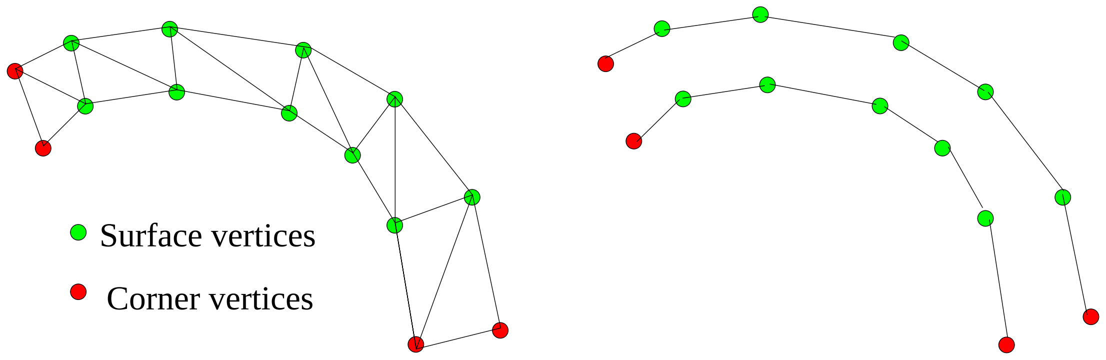


Figure 5: Original and modified graph

Figure 6 shows a possible MIS and coarse grid for this problem.

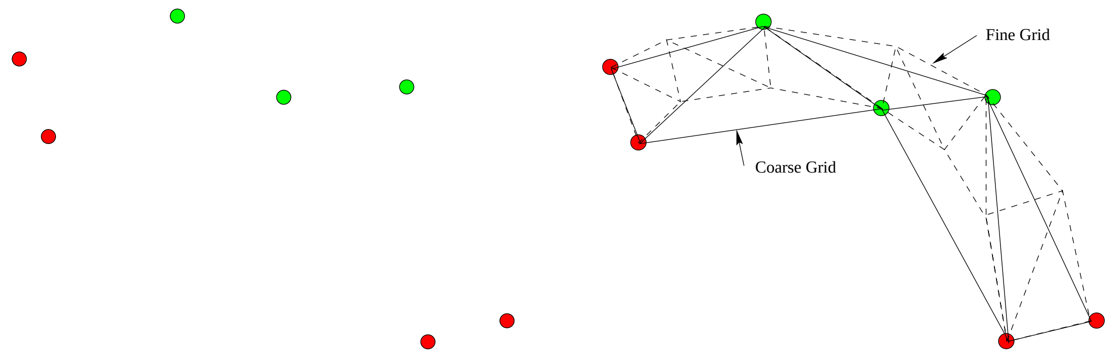


Figure 6: MIS and coarse mesh

4.7 Vertex ordering in MIS

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 (or rank) - the ordering within each category can also be specified. Two simple heuristics can be used to order the vertices: a "natural" order and a random order. Meshes may be initially ordered in a block regular order (ie, an assemblage of logically regular blocks), or ordered in a cache optimizing order like Cuthill-McKee [24]; both of these ordering types are what we call *natural* orders. 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 ratio of the MIS size to the vertex set size is bounded by $1/2^3$ and $1/3^3$, 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, and fewer levels are required before the coarsest grid is small enough to solve directly, but care must be taken not to degrade the convergence rate of the solver. 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.

4.8 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 construct the finite element coarse grid space of our method. We use a standard Delaunay meshing algorithm to give us these meshes [10]. This is done by placing a bounding box around the coarse grid vertices (on each processor) 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 that connect vertices that were not "near" each other on the fine mesh and that do not have any fine grid vertices that lie "uniquely" within the tetrahedron. Define a vertex v to lie uniquely in a tetrahedron if there is no tetrahedron that can provide the fine grid vertex with interpolates that are all above -ϵ (for some small number ϵ). Finally, for each vertex v in $lost\_list$, we find a nearby element to use for the interpolants for v. Figure 7 shows an example of our methods applied to a problem in 3D elasticity. The fine (input) mesh is shown with three coarse grids used in the solution.

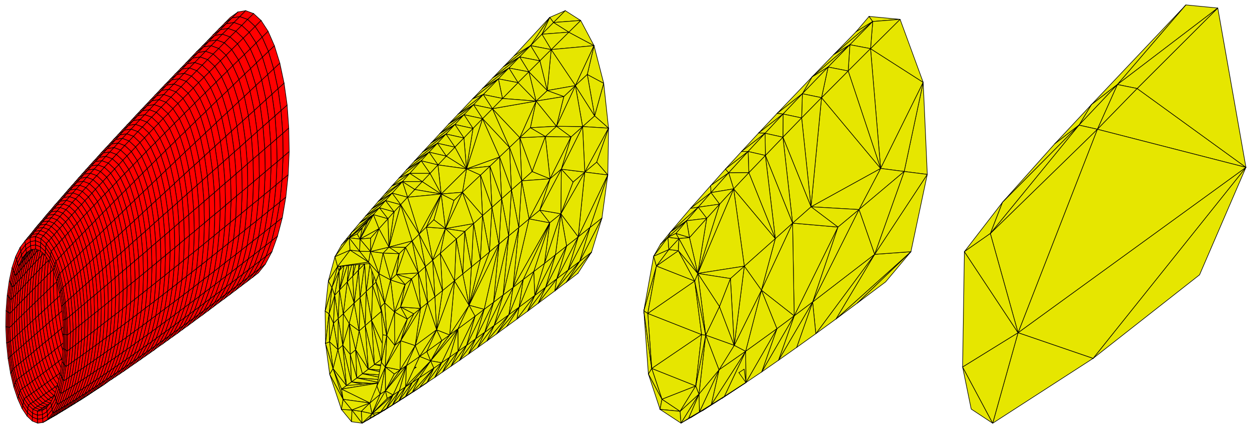


Figure 7: Fine (input) grid and coarse grids for problem in 3D elasticity

5 Parallel architecture

We have developed a highly scalable implementation of our algorithms and a parallel finite element application for solid mechanics problems to effectively test our solver. Our parallel finite element system is composed of two basic parts: *Athena*, a parallel finite element program built on a serial finite element code (FEAP [9]) and a parallel mesh partitioner (ParMetis [16]), and our solver *Prometheus*. Prometheus can be further decomposed into two parts: *Epimetheus*, general unstructured multigrid support (built on PETSc [3]); and our particular multigrid algorithm Prometheus (whose sole responsibility is to construct the restriction operators between each grid). Prometheus and Epimetheus are not implemented separately and constitute the publicly available library [19].

Athena reads a large "flat" finite element mesh input file in parallel (ie, each processor seeks and reads only the part of the input file that it, and it alone, is responsible for), uses ParMetis to partition the finite element graph, and then constructs a complete finite element problem on each processor. These processor sub-domains are constructed so that each processor can compute all rows of the stiffness matrix, and entries of the residual vector, associated with vertices that have been partitioned to the processor. This negates the need for communication in the finite element element evaluation at the expense of some redundant work.

We use explicit message passing (MPI) for performance and portability, parallelize all parts of the algorithm for scalability. All components of multigrid can scale reasonably well (except for the coarsest grids, whose size remains constant as the problem size increases and is thus not a hindrance to scalability).

We target clusters of symmetric multi-processors (SMPs), which we call CLUMPs, as this seems to be the architecture of choice for the next generation of large machines. We accommodate CLUMPs by first partitioning the problem onto the SMPs and then the local problem is partitioned on to each processor. This approach implicitly takes advantage of any increase in communication performance within each SMP, though the numerical kernels (in PETSc) are "flat" MPI codes. Figure 8 shows the a diagram of the overall system architecture.

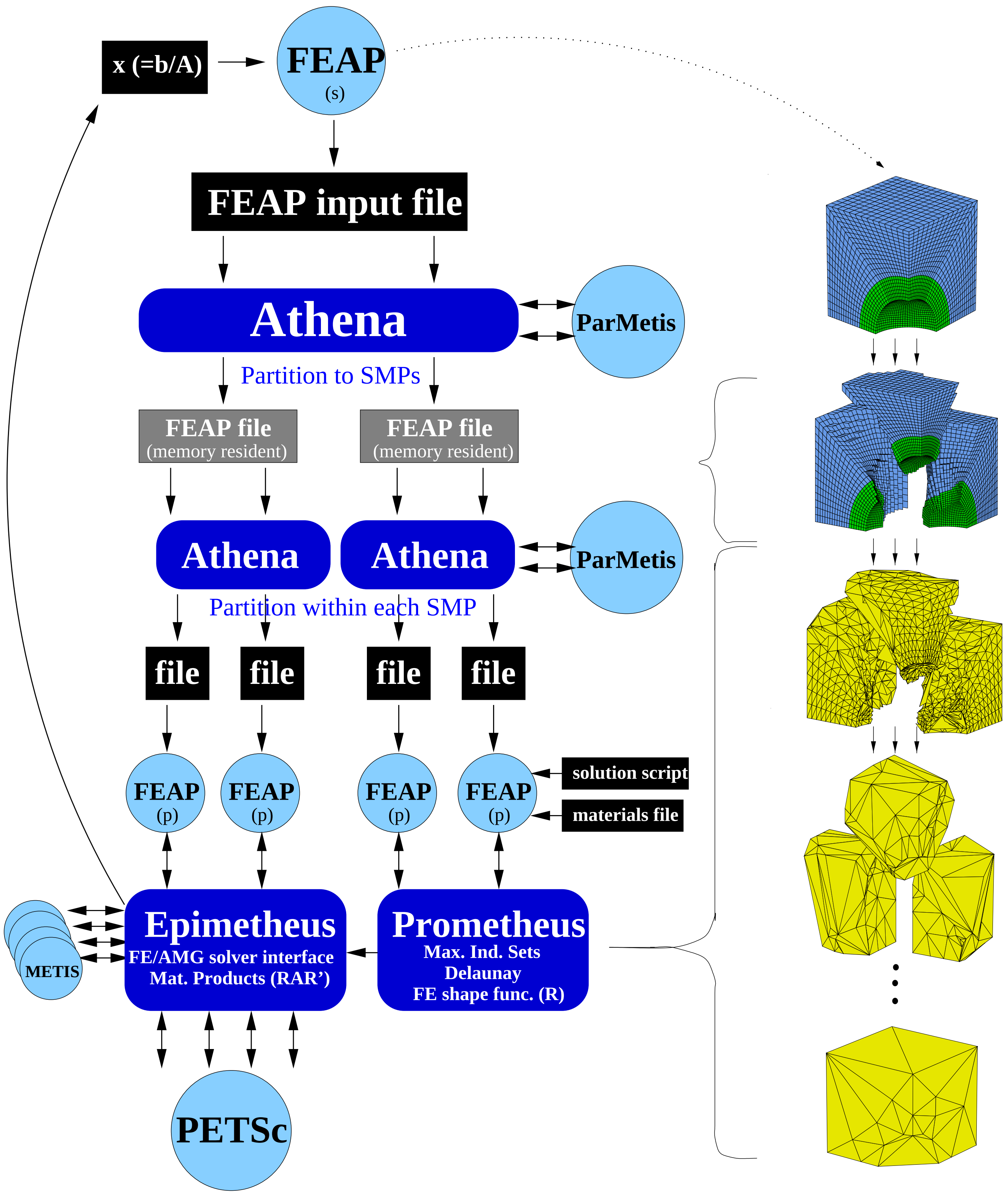


Figure 8: Athena/Prometheus Architecture

6 Performance measures

The section defines the methods, goals, and terminology of our numerical experiments. The goal of our numerical experiments is to measure the degree of *scalability* of our algorithms and implementations. We use *efficiency* as the primary metric in the analysis of our experimental data. Perfect efficiency is defined as 1.0 and of course the higher the efficiency the better. Time efficiencies are of the form: "optimal" / "measured" (ie, $\frac{T(c,1)}{T(c \cdot P, P)}$ where $T(n, P)$ is the time to solve a problem with n equations on P processors) and flop rate efficiencies are of the form: "measured" / "optimal". Orthogonal sources of (in)efficiencies (eg, iteration counts and flop rates) can be multiplied together to give the total efficiency, which is often the most easily measured term - other efficiencies can be back calculated from the total efficiency with a model of the computation.

Efficiencies are useful as 1) they provide a uniform metric for measuring the many sources of slowdown in a code, 2) they provide a bound on the benefit that can be gained by optimizing a particular aspect of a code, 3) they help identify scalability bottlenecks in the application. We decompose efficiency into *uniprocessor efficiency* and *parallel efficiency* - the total efficiency being the product of the two. Uniprocessor efficiency is defined below; parallel efficiency e is defined as the time to solve a problem of size n_1 on one processor divided by the time to solve a refined discretization of the problem, with $n_P = P \cdot n_1$ equations on P processors.

The remainder of this section discusses the decomposition of efficiency, can be used as a reference for the numerical results sections that follow, and can be skipped in the first reading. In general efficiency, or sources of inefficiency, can be decompose into the following components:

- **uniprocessor efficiency** or uniprocessor efficiency e_u: the fraction of some "peak" megaflop rate (Mflop/sec) $e_u = \frac{f(1)/sec}{f(1)_{PEAK}/sec}$ of the uniprocessor implementation, where $f(1)/sec$ is the uniprocessor flop rate. Peak can be defined as:
  - Theoretical peak (ie, clock rate times flop issues per cycle).
  - Dense matrix matrix-multiply - the fastest megaflop rate that any numerical code is likely to be able to sustain.
  - Sparse matrix-vector multiply (with A in $Ax = b$) - the source of most of the flops in most iterative methods - the fastest megaflop rate that any iterative solver is likely to be able to sustain in the solve phase.

- Parallel efficiency is the product of the four efficiencies described below:
  - **work efficiency** e_w: the fraction of flops in the parallel implementation that are not redundant, ie, the number of flops to solve the problem on one processor divided by the number of flops to solve the *same* problem with P processors. Thus, $e_w = \frac{f_P(1)}{f(P)}$ where $f_P(1)$ is the number of flops used to solve the n_P unknown problem on one processor and $f(P)$ is the number of flops used to solve the n_P unknown problem with P processors.
  - **scale efficiency** e_s: this is the scalability of the algorithm with respect to non-redundant flops per unknown (ie, the non-redundant flops per unknown to solve the problem as the problem size increases). For iterative solvers, it is convenient to further decompose scale efficiency.
    - **iteration scale efficiency** e_s^I: the efficiency in the number of iterations required; $e_s^I = \frac{Iterations(1)}{Iterations(P)}$, where $Iterations(P)$ is the number of iterations for the problem of size n_P on P processors.
    - **flop scale efficiency** e_s^F: the efficiency in the non-redundant flops per unknown (or processor) per iteration. We define the flops per iteration of the n_P unknown problem by $f^I(P)$ and the non-redundant flops as $\hat{f}(P)$, and $e_s^F = \frac{P \cdot f^I(1)}{\hat{f}^I(P)}$. Recall, $f(P)$ is the toatl flops and so $\hat{f}(P) = e_w \cdot f(P)$. Work efficiency e_w is similar to e_s in that it relates to flop efficiency, though distinct from e_s as: e_w is related only to the number of processors used P; scale efficiency e_s is related only to the size of the problem.
  - **load balance** l: the ratio of the average to the maximum amount of work (flops) that a processor does in an operation, $e_l = \frac{f_{average}}{f_{maximum}}$. This is easily measured (and defined) as we do not use any non-uniform algorithmic constructs (ie, all processors are doing the same operation on the same mathematical object all of the time). Our load balance can be seen graphically in Figure 11.

– **communication efficiency** e_c: the highest percentage of time that a processor is *not* waiting, processing, packing data, or any other form of work associated with interprocess communication. Communication efficiency e_c can be measured with flop rate efficiency, $e_c = \frac{f(P)/sec}{P \cdot f(1)/sec}$, as in Figure 11 (right), if there is no load imbalance induced blocking.

Our solver has perfect work efficiency e_w, though we have some redundant work in the construction of the fine grid matrix A_0 in Athena (§5) as can be seen in Figure 12 (right) - we do not discuss work efficiency further and can assume that $\hat{f}(P) = f(P)$. Our load balance is very good (see Figure 11) and we do not discuss load balance further.

We focus on communication efficiency e_c and scale efficiency e_s; communication efficiency e_c can be represented with the flop rate efficiency. Scale efficiency $e_s = e_s^I \cdot e_s^F$ (flops per unknown) is the product of the number of iterations required to converge and the number of flops per unknown per iteration. Thus, the efficiency $e(P)$ on P processors (and n_P unknowns as defined above) can be effectively represented as

$$e(P) \approx \frac{Iterations(1)}{Iterations(P)} \cdot \frac{P \cdot f(1)}{f(P)} \cdot \frac{f(P)/sec}{P \cdot f(1)/sec} = e_s^I \cdot e_s^F \cdot e_c$$

with the number of iterations $Iterations(P)$, flops iteration $f(P)$, and flop rate $f(P)/sec$. $Iterations(P)$ is tabulated in Table 2, $\frac{P \cdot f(1)/sec}{f(P)/sec}$ is shown if Figure 11 (left), and $\frac{f(P)/sec}{P \cdot f(1)/sec}$ shown if Figure 11 (right).

This paper focuses on parallel efficiency but a few words about uniprocessor efficiency e_u are warranted. Uniprocessor efficiency can be measured against the theoretical peak megaflop rate of the processors; this is simple to define though it is more effective for measuring the performance of the processor in question rather than the algorithm and implementation. The megaflop rate of dense matrix-matrix multiply is more useful than peak speed as this invariably bounds the performance of numerical applications. Sparse matrix-vector multiply, with the fine grid matrix, is most useful as it is the source of most of the flops in most scalable solvers. We report data for the first and third uniprocessor efficiency (ie, theoretical peak and fine grid sparse matrix-vector multiply) in the following sections.

Good scalability can be defined as good parallel efficiency ($e \geq 1.0$ or $e \geq C \gg 0$ for some constant C independent of the number of processors) for the time to compute the solution $\hat{x}$ that reduces the 2-norm of the residual by a fixed constant $rtol$ (ie, $\frac{\|A\hat{x}-b\|}{\|b\|} \leq rtol$). We could alternatively define scalability by solving the linear system to the finite element discretization error; this is more attractive as it insures that perfect parallel efficiency is bounded from above by 1.0 and reflects the rational use of a solver. Discretization error is not used as the convergence tolerance because it is difficult to define (compute). Note, the PRAM complexities of all iterative solver algorithms are, to our knowledge, bounded by $O(\log(n))$, thus optimal parallel efficiency will include a $\log(n)$ term.

An additional note, there are three main phases in the solution of a linear system: setup for the mesh (non-zero structure of the fine grid), setup for each matrix (more than one for nonlinear problems), and the solve for x with a provided right hand side (RHS) b. For direct solvers these three phases correspond to symbolic factorization, numerical factorization, and the front solve and back substitution. The first phase is amortized in nonlinear and transient problems and is thus not as important as the latter stages, unless the application solves just one linear system of equations for each mesh configuration. The second phase, setup for each matrix, includes the coarse grid operator construction and smoother setup in our solver, and is important for fully nonlinear problems as it is required for each RHS and thus its cost is not amortized by multiple solves. The final stage, the time in the actual multigrid iterations, is the most important as this is always done in the solve of a system of linear equations. We focus on the solve phase (the last phase), though we have fully scalable implementations of all three phases and we report times for all stages (Figure 10).

7 Numerical results

We use a model problem in solid mechanics to conduct scalability studies of our solver. Our problem is a sphere embedded in a cube; the sphere is constructed of seventeen alternating "hard" and "soft" layers and the cube is a "soft" material. Think of a spherical steel-belted radial inside a rubber cube. Symmetry can be used to model only one octant. The loading and boundary conditions are an imposed uniform displacement (crushing), on the top surface and symmetric boundary conditions on the three cut faces. The hard material is a J_2 plasticity material with a mixed formulation and kinematic hardening [22]. The soft material is a large deformation (Neo-Hookean) hyperelastic material with a mixed formulation [26]. Table 1 shows a summary of the constitution of our two material types.

Material	Elastic mod. (E)	Poisson ratio	deformation type	yield stress	hardening mod.
soft	10^{-4}	0.49	large	∞	NA
hard	1	0.3	large	0.001 E	0.002

Table 1: Nonlinear materials

The hexahedral discretization is parameterized so that we can perform scalability experiments. Figure 9 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 (ie, 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, 26,257 K, and 39,161 K degrees of freedom. We run this problem with about 40 K degrees of freedom per processor, on 2 to 960 processors.

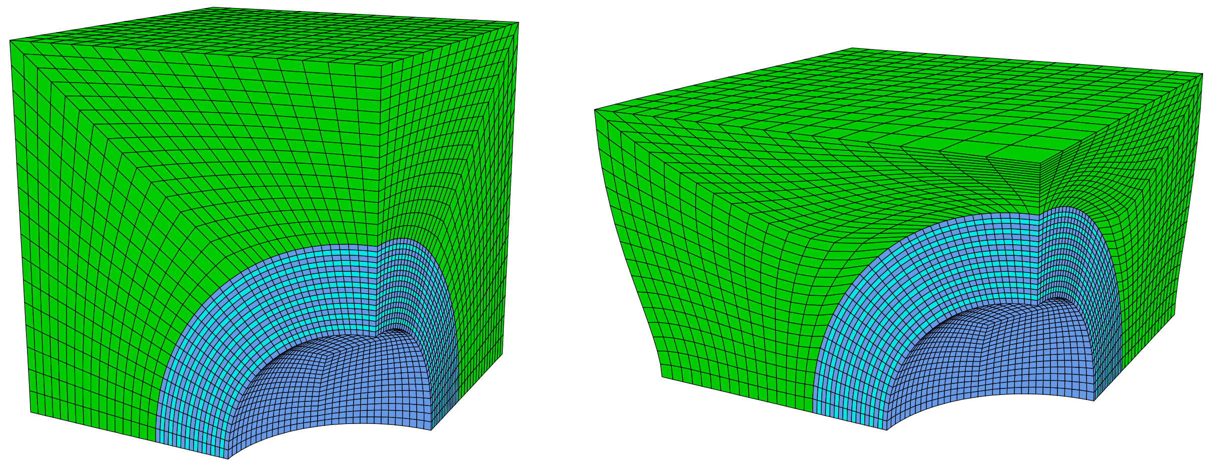


Figure 9: 79,679 dof concentric spheres problem and final configuration

These experiments are performed on an IBM PowerPC cluster with 240 4-way SMP nodes at Lawrence Livermore National Laboratory. Each node has four 332 MHz PowerPC 604e processors, with 1.5 Gbytes of memory, and a theoretical peak Mflop rate of 664 Mflop/sec per processor.

Single processor (PETSc) sparse matrix-vector products (on the fine grid of our problems) run at 36 Mflop/sec, and the multigrid solves run at 34 Mflop/sec; two processor (the base case for these experiments) matrix-vector products run at 66 Mflop/sec, and the multigrid solves run at 63 Mflop/sec; the 960 processor case runs at 26.5 Gflop/sec in the matrix-vector products and 19.3 Gflop/sec in the solves. Thus, we have 59% ($=\frac{19,300}{960 \cdot 34}$) parallel efficiency in the solve in these experiments. We have also run these experiments (up to the 24 million degrees of freedom problem) on a 640 processor Cray T3E with 57% parallel efficiency as well, and about twice the total Mflop rate as the corresponding IBM experiment.

7.1 Scalability study of one linear solve

We look at data from the first linear solve in these problems (with a convergence tolerance of 10^{-4}, the first linear solve tolerance in our nonlinear solver in section §7.2) in detail. Figure 10 (left) shows the times for the major subcomponents of the solver, and Figure 10 (right) shows the times for the setup of the finite element problem (Athena), unmeasured code (mostly in the FEAP setup phase), the coarse grid construction (Prometheus), the fine grid creation (FEAP), the matrix setup phase (Epimetheus and PETSc), and the actual solve time in multigrid (PETSc).

Note, the times for the "matrix setup" are taken from a separate (two solve) run, and the second setup time is used. We report the time for the second application of the "matrix setup" because there is small amount of overhead on the first call to the matrix triple product routine that is not measured separately. Thus, time for the second call to these "matrix setup" routines is more indicative of the asymptotic cost, per matrix, of the "matrix setup". The difference between the first and all subsequent calls to the "matrix setup" routines is about 15%.

It is important to note that this (linear solve) experiment measures one mesh setup, one matrix setup, and one solve (for each right hand side). Linear transient analysis would require multiple solves that would amortize the cost the setup phases, and nonlinear analysis (§7.2) would require multiple matrix setups and solves which would amortize the mesh setup phase (restriction operator construction). The unmeasured code ("FEAP setup + misc." in Figure 10) is mostly the time that FEAP reads the input file, plus miscellaneous items (eg, MPI/PETSc initialization and finalization, diagnostic work at the end of the run, the difference between the "matrix setup" that we use and the actual time as discussed above, etc.).

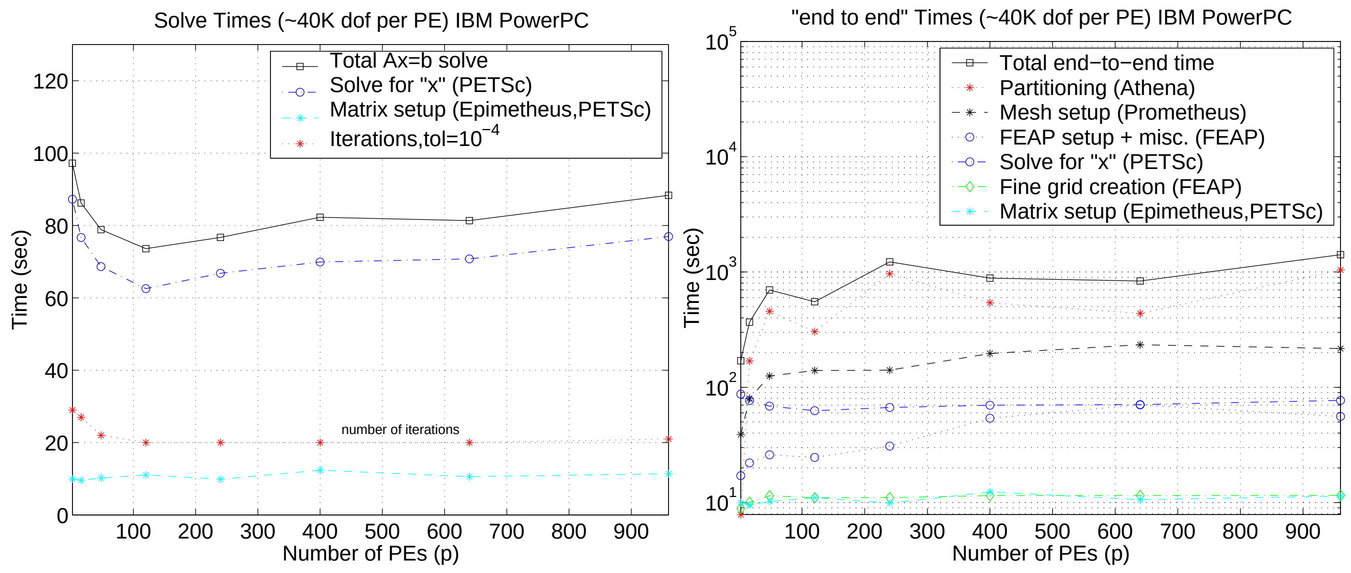


Figure 10: 40,000 dof per processor solve times (left) and total times (right) for one linear solve

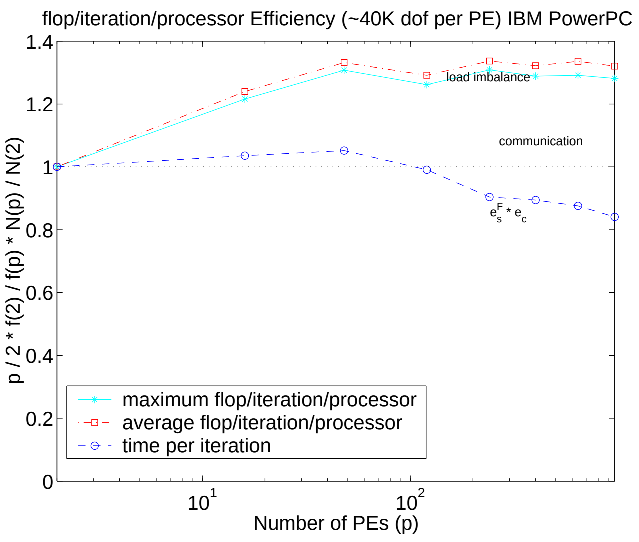
All phases of the computation are scaling reasonably well. The FEAP interface is not optimal in the current implementation as we actually write a file to disk and FEAP (immediately) reads this file - this could be done in memory by having FEAP read from a memory stream instead of a file stream (we have implemented this optimization in the recursive application of Athena, see section §5). Also, Athena is running slowly and erratically, as can be seen in the times of Figure 10 (right). We believe that this is due to bottlenecks in the communication system as we see speedups of about six with the use of twice as many processors. Additionally, we have not concentrated our code optimization efforts towards the initial setup phase of our parallel finite element application for lack of software development resources and as these costs are amortized in transient problems as can be seen in section §7.2. Thus, we use highly scalable algorithms for the set up phase (as it is scaling well) but have not fully optimized them to reduce the constants in the run time. Note, Athena is not in the library release of Prometheus.

We see super-linear efficiency in the solve times (eg, the solve times are decreasing as the problem size increases) in Figures 10, for two reasons. First, we have super-linear convergence rates (ie, $e_s^I > 1.0$), as shown in Table 2. Second, the vertices added in each successive scale problem have a higher percentage of *interior* vertices than the base 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 $O(n^{\frac{2}{3}})$ whereas the interior increases by $O(n)$. Thus, the ratio of interior vertices to surface vertices increases as the scale of discretization decreases (n increases). Our coarse grid heuristics in section §4 articulate the surfaces well (boundary and material interfaces), resulting in a higher ratio of surface vertices promoted to the coarse grid. Thus, the *rate* of vertex reduction is higher on the larger problems as they have proportionally more interior vertices, leading to less work per fine grid vertex (ie, $e_s^F > 1.0$), as can be seen in Figure 11 (left).

Equations	79,679	622,815	2,085,599	4,924,223	9,594,879	16,553,759	26,257,055	39,160,959
Processors	2	15	50	120	240	400	640	960
MG preconditioned PCG Iterations in 1^{st} linear solve	29	27	22	20	20	20	20	21
Total PCG iterations in nonlinear solve	3108	4121	3117	3355	3060	3008	2978	3215
Total Newton iterations	62	63	62	65	68	69	68	70
Ave. PCG iterations per linear solve	50	65	50	52	45	44	44	46
Total Mflop/sec in MG iterations	63	421	1194	2901	5112	8524	13218	19253

Table 2: Number of iterations for first linear solve and total nonlinear solve

Figure 11 (left) shows the scaled efficiency of the number of flops per iteration per unknown in the solver e_s^F, as well as the scaled efficiency of the time for each iteration. The efficiency of the time per iteration, the lower line in Figure 11 (left), shows the combined effect of the super-linear flop efficiency e_s^F and the parallel (in)efficiency of the flop rate (flop/sec/processor) e_c. Communication efficiency e_c is shown in Figure 11 (right). The total "solve for x" efficiency e (ie, the solve efficiency without the matrix setup or the grid setup phases) is shown in Figure 12 and is approximately $e = e_s^I \cdot e_s^F \cdot e_c$.



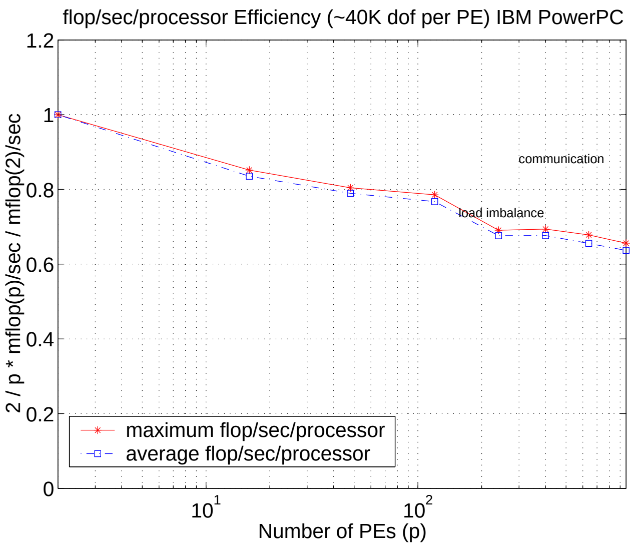


Figure 11: Flop/iteration/proc. (flop scale efficiency e_s^F); flop rate (communication efficiency e_c)

Figure 11 (right), the scaled efficiency of the maximum and average processor flop rate in the solve phase, shows that we have 62% parallel efficiency in the flop rate (from the two processor case). We do not have a one processor version of this problem but, using the flop rate of 34 Mflop/sec in the solve from a similar 40 K degree of freedom problem on one processor, we have 59% parallel efficiency in the solve phase. Figure 11 also shows two views of two sources of inefficiency: communication e_c and load imbalance l as discussed in section §6.

Note, Figure 11 (left) is scaled by a factor $\frac{2}{P} * \frac{N(P)}{N(2)}$ to account for the non-constant number of unknowns per processor ($N(P)/P$); this factor is shown in Figure 12 (left) and the efficiencies of all of the major steps the linear solve in Figure 12 (right).

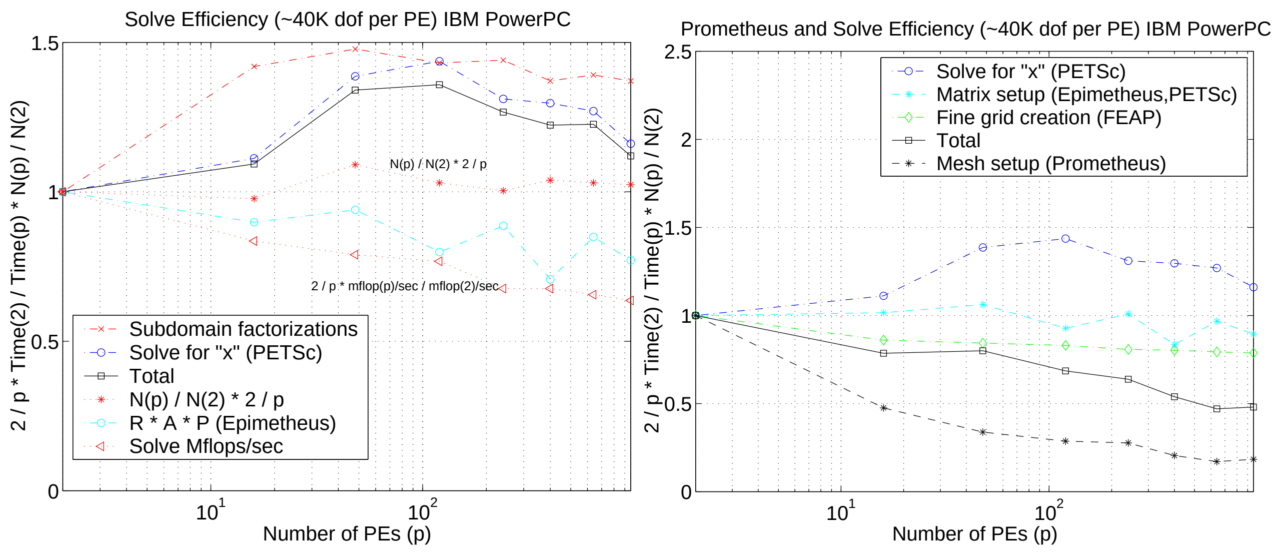


Figure 12: efficiency e for all major components of one linear solve with about 40,000 dof per processor

7.2 Nonlinear scalability study

We use a full Newton nonlinear solution method; convergence is declared when the energy norm of the correction is 10^{-20} times that of the first correction. This means that in Newton iteration m convergence is declared when

$$\left|x_m^T \cdot (b - Ax_m)\right| < 10^{-20} \cdot \left|x_0^T \cdot (b - Ax_0)\right|$$

. The linear solver, within each Newton iteration, is preconditioned conjugate gradient (PCG), preconditioned with one "full" multigrid cycle. We use one pre-smoothing and one post-smoothing step within multigrid, preconditioned with block Jacobi with 6 blocks for every 1,000 unknowns (these block Jacobi sub-domains are constructed with METIS).

The slightly modified version of $FEAP$ ($FEAP_p$ [9]) calls our linear solver in 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 and $rtol_m = min(10^{-3}, \frac{\|r_m\|}{\|r_{m-1}\|} \cdot 10^{-1})$ on all subsequent iterations ($m > 1$). This heuristic is intended to minimize the number of total iterations required in the Newton solve by only solving each set of linear equations 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 is not likely to be economical.

The full nonlinear problem is run with ten "time" steps with a particular displacement increment in each step that results in a total vertical displacement of 3.6 inches down (the octant is 12.5 inches on a side and the top "soft" section is 5 inches thick at the central vertical axis). Figure 13 (left) shows the percentage of the integration points, in the "hard" shells, whose stress state have reached the yield surface [22]; this gives an indication of the damage to the system and results in material constitution that is very similar to that of highly incompressible materials. Over 24% of the integration points, in the hard shells, are in the yield state at the final configuration.

Figure 13 (right), and Table 2, show the number of multigrid iterations in each linear solve of each of the ten Newton solves, stacked on top of each other and color coded for each Newton iteration. From this data we can see that the total number of iterations is staying about constant as the scale of the problem increases. This data suggests that the nonlinear problem is getting *harder* to solve as the discretization is refined, because the number of iterations in the first linear solve

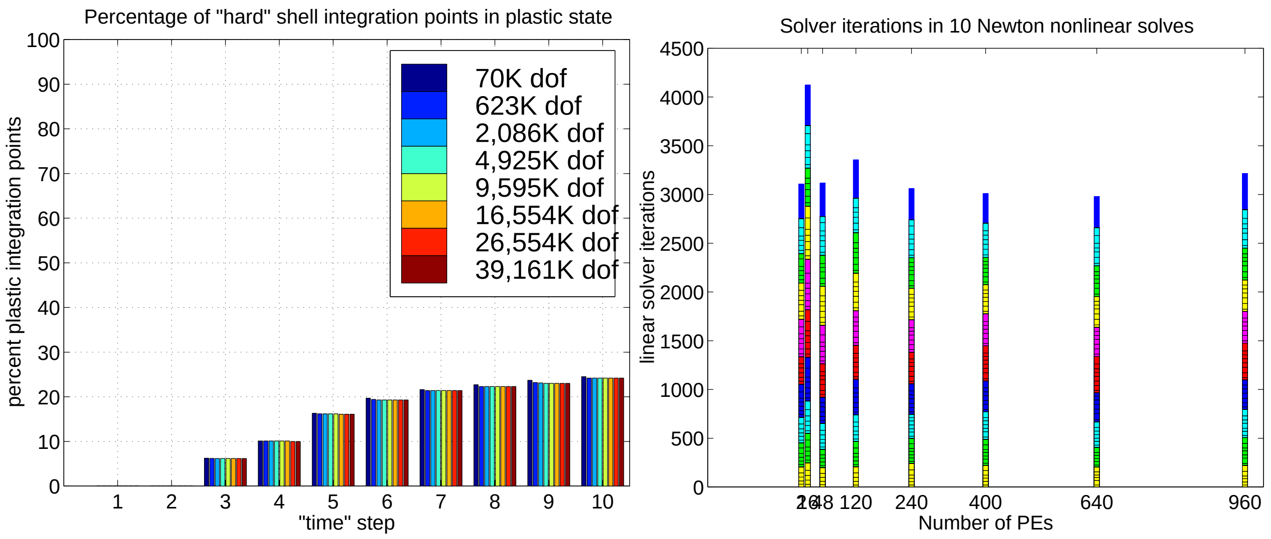


Figure 13: Percent of integration points in "hard" material that are in plastic state in each "time" step; Number of iterations for all solves in nonlinear problem (see Table 2)

of the first step decreases as the problem size increases, as is shown in Table 2. That is, we are seeing a slight growth in the number of Newton iterations required, and the average number of iterations in the linear solver is not decreasing as dramatically as in the first linear solve. Table 2 shows the detailed iteration count data from these experiments.

8 Conclusion

We have developed a promising method for solving the linear set of equations arising from implicit finite element applications. Our approach, a 3D parallel extension to a serial 2D algorithm, is to our knowledge unique in that it is a fully automatic (ie, 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 solved 40 million degree of freedom problems on an IBM PowerPC cluster with 240 4-way SMP nodes with about 60% parallel efficiency. Prometheus is the only fully parallelized scalable public domain solver that we are aware of and can be obtained from the Prometheus home page [19].

We have also developed a highly parallel finite element implementation, built on an existing state-of-the-art serial research finite element implementation. The implementation of our system (Athena/Prometheus) 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 (20,000 lines of C), and geometric predicates (4,000 lines of C) [21].

Future work will include the continuing hardening of the algorithms and implementation with the application of Prometheus to a wider variety of problems, as well as expanding the domain of applications which we support (eg, higher order elements, non-continuum elements). As we also plan to explore alternative (effective) unstructured multigrid algorithms such as smoothed aggregation [25], to evaluate (and make publicly available) competitive algorithms.

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. We would like to thank the reviewers for their helpful comments, Lawrence Livermore National Laboratory for providing access to its IBM cluster 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 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] M. F. 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.

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

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

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

[5] 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.

[6] T. F. Chan and B. 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.

[7] J. Demmel. *Applied Numerical Linear Algebra*. SIAM, 1997.

[8] 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.

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

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

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

[12] 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.

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

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

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

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

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

[18] 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.

[19] Prometheus. http://www.cs.berkeley.edu/∼madams/Prometheus-1.1.

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

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

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

[23] B. Smith, P. Bjorstad, and W. Gropp. *Domain Decomposition*. Cambridge University Press, 1996.

[24] S. 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.

[25] 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.

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