# A Distributed Memory Unstructured Gauss-Seidel Algorithm for Multigrid Smoothers

Mark F. Adams *

November 19, 2001

**Abstract**

Gauss-Seidel is a popular multigrid smoother as it is provably optimal on structured grids and exhibits superior performance on unstructured grids. Gauss-Seidel is not used to our knowledge on distributed memory machines as it is not obvious how to parallelize it effectively. We, among others, have found that Krylov solvers preconditioned with Jacobi, block Jacobi or overlapped Schwarz are effective on unstructured problems. Gauss-Seidel does however have some attractive properties, namely: fast convergence, no global communication (ie, no dot products) and fewer flops per iteration as one can incorporate an initial guess naturally. This paper discusses an algorithm for parallelizing Gauss-Seidel for distributed memory computers for use as a multigrid smoother and compares its performance with preconditioned conjugate gradients on unstructured linear elasticity problems with up to 76 million degrees of freedom.

Key words: unstructured multigrid, algebraic multigrid, parallel graph algorithms, parallel Gauss-Seidel

## 1 Introduction

The availability of large high performance computers is providing scientists and engineers with the opportunity to simulate a variety of complex physical systems with ever more accuracy and thereby exploit the advantages of computer simulations over laboratory experiments. The finite element method is widely used for these simulations. The finite element method requires that one or several linearized systems of sparse unstructured algebraic equations (the stiffness matrix) be solved for static analyses, or at each time step when implicit time integration is used. These linear system solves are the computational bottleneck (once the simulation has been setup and before the results are interpreted) as the scale of problems increases. Direct solution methods have been, and are still, popular because they are dependable; however the asymptotic complexity of direct methods is high in comparison to optimal iterative methods (ie, multigrid §2).

Once an unstructured multigrid method has been implemented the selection of the smoother and its parameters (eg, number of smoothing steps, size of subdomains, drop tolerances) becomes a primary means of optimizing the solution time. Additionally, a large majority of the time is spent in the smoother. Hence, the selection and efficient implementation of smoothers is of primary interest in optimizing the performance of a multigrid solver.

Conjugate gradients (CG) preconditioned with Jacobi, block Jacobi or overlapping Schwarz has been found to be an effective smoother for problems in 3D linear elasticity [10, 4]. It is well know that for model problems Jacobi smoothers, or more generally additive Schwarz smoothers, require damping. CG in effect provides this damping for unstructured problems. The advantage of additive smoothers is that they are easily parallelizeable; CG is also relatively easy to parallelize effectively as all of the work is performed in standard numerical primitives (ie, matrix-vector products, dot products, etc.) which have presumably been optimized for the machine in use and are easily available. Incomplete factorizations have been found to be effective smoothers [11, 6]. They, however, require the selection of parameters for fill-in and/or shifting the

*Sandia National Laboratories, MS 9417, Livermore CA 94551 (mfadams@ca.sandia.gov).  SC2001 November 2001, Denver 1-58113-293-X/01/0011 $5.00

matrix to maintain positiveness, but incomplete factorizations are also useful smoothers. We do not have access to good incomplete factorization implementations and we do not discuss them further.

These preconditioned CG smoothers, in addition to incomplete factorizations, are the only smoothers used, to our knowledge, on distributed memory machines. Gauss-Seidel is, however, widely used in the multigrid community because

- Better convergence bounds can be proven for Gauss-Seidel than for damped Jacobi on model problems.
- Gauss-Seidel works well on unstructured problems without the need of picking a damping parameter.

Gauss-Seidel has several additional attractive properties:

- Gauss-Seidel has better PRAM complexity as no dot products are required. Future parallel computers will likely have slower networks (relative to processor speed) than today and so Gauss-Seidel may become more advantageous in the future on large problems.
- Krylov methods require that an explicit residual be calculated if an initial guess is provided (as occurs with full multigrid and half of the time with V-cycle multigrid). This cost can be significant as one usually only performs a few iterations in a smoother and so the residual cost can not be amortized well.
- Gauss-Seidel is stationary (unlike CG), this is an attractive property as multigrid theory is not well established for non-stationary smoothers and stationary smoothers are required for GMRES which is a popular Krylov method for unsymmetric problems.

Gauss-Seidel is thus an attractive smoother and is used in serial and shared memory parallel computing [18]. Gauss-Seidel has not, however, been used on distributed memory computers as it is not obvious how to parallelize it well. Note, it is natural to try inexact Gauss-Seidel (eg, processor subdomain block Jacobi with Gauss-Seidel subdomain solver) and these have been found to be adequate for Poisson's problem (ie, 1D elasticity) but require modification to the multigrid interpolation [12], and thus it is not surprising that we have found that inexact Gauss-Seidel does not work for 3D elasticity (at least with our geometric and algebraic multigrid methods). This paper presents a distributed memory unstructured true Gauss-Seidel algorithm that shows promise on 3D unstructured elasticity problems with up to 76 million degrees of freedom.

# 2 Multigrid introduction

This section provides a brief introduction to multigrid, defining terms and providing comments on the structure of multigrid relevant to its implementation on high performance (ie, parallel) computers for unstructured grid problems. Multigrid has been an active area of research for almost 30 years and much literature can found on the subject [15]. Multigrid is motivated by the observation that simple (and inexpensive) iterative methods like Gauss-Seidel, damped Jacobi and block Jacobi, are effective at reducing the high frequency error, but are ineffectual in reducing the low frequency content of the error [7]. 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 material 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 way that the finite element method projects the continuous solution onto a finite dimensional subspace to compute an approximation to the solution. Multigrid is practical because this projection can be prepared and computed reasonably cheaply and has $O(n)$ complexity. The coarse grid correction (the solution projected onto a coarser grid) does not eliminate the low frequency error exactly, but it "deflates" the low frequency error to high frequency error by removing an approximation to the low frequency components from the error.

Multigrid requires three types of operators: 1) the grid transfer operators (ie, the *restriction* and *prolongation* operators, which can be implemented with a rectangular matrix $R$ and $P = R^T$ respectively);

2) the PDE operator, a sparse matrix, for each coarse grid (the fine grid matrix is provided by the finite element application); and 3) cheap (one level) iterative solvers that can effectively eliminate high frequency error in the problem. The coarse grid matrix can be formed in one of two ways, either algebraically to form Galerkin (or variational) coarse grids ($A_{coarse} \leftarrow RA_{fine}P$) or, by creating a new finite element problem on each coarse grid (if an explicit coarse grid is available) thereby letting the finite element implementation construct the matrix.

Figure 1 shows the standard multigrid *V-cycle* and uses a smoother $x \leftarrow S(A, b)$, and restriction operator $R_{i+1}$ that maps residuals from the fine grid space $i$ to the coarse grid space $i+1$ (the rows of $R_{i+1}$ are the discrete representation on the fine grid of the coarse grid function space of grid $i+1$).

**function** $MGV(A_i, r_i)$
  **if there is a coarser grid** $i+1$
    $x_i \leftarrow S(A_i, r_i)$
    $r_i \leftarrow r_i - Ax_i$
    $r_{i+1} \leftarrow R_{i+1}(r_i)$ /* restriction of residual to coarse grid */
    $x_{i+1} \leftarrow MGV(R_{i+1}A_iR_{i+1}^T, r_{i+1})$ /* the recursive application of multigrid */
    $x_i \leftarrow x_i + R_{i+1}^T(x_{i+1})$ /* prolongation of coarse grid correction */
    $r_i \leftarrow r_i - A_ix_i$
    $x_i \leftarrow x_i + S(A_i, r_i)$
  **else**
    $x_i \leftarrow A_i^{-1}r_i$ /* direct solve of coarsest grid */
  **return** $x_i$

Figure 1: Multigrid *V-cycle* Algorithm

Many multigrid algorithms have been developed; the *full* multigrid algorithm is used in our numerical experiments. One full multigrid cycle applies the V-cycle to each grid, by first restricting the residual ($b$) to the coarsest grid and applying a V-cycle (simply a direct solve), interpolating the new solution to the next finer grid as an initial guess, applying the V-cycle to this finer grid, interpolating to the next finer grid and so on until the finest grid is reached. Multigrid is often used as a preconditioner for a (Krylov) iterative method; we use CG preconditioned with one full multigrid iteration in our numerical experience.

The reason for using multigrid is to insure that the convergence rate is independent of the scale of the problem and the cost of each iteration, in floating point operations (flops), asymptotes to a constant as the scale of the problem increases. An additional attractive property of multigrid is that the solver has several distinct parts that are essentially independent: the restriction/prolongation operators, the smoother, the Krylov method accelerator (the actual solver), multigrid algorithms such as V-cycles, F-cycles, W-cycles, and other standard multigrid infrastructure (ie, sparse matrix triple products for algebraic coarse grids). The smoother can have an important impact on this cost, especially on challenging problems, and is the primary parameter in optimizing the solve time for a particular problem.

# 3 Parallel Gauss-Seidel algorithms

Gauss-Seidel is specific type of multiplicative Schwarz method [17]. The algorithms discussed here are general methods for the parallel implementation of multiplicative Schwarz methods. That is, these methods operate on graphs that are derived from the matrix graph and the Schwarz subdomains (blocks). Given a set of, perhaps overlapping, vertex block or lists ($I$,$J$,...) a graph is constructed by coalescing the vertices in each list to one node of the graph. An edge exists between nodes $I$ and $J$ if there is an edge between the finite element nodes $n1 \mid n1 \in I$ and $n2 \mid n2 \in J$. Our numerical experiments use 1) nodal block Gauss-Seidel (these graphs are identical to the graph of the finite element mesh) and 2) non-overlapped Schwarz subdomains. The actual equations only come into play when the weights of the nodes are computed for load (work) balancing (in Equation 1 below) and in the application of Gauss-Seidel in the kernel of the algorithm ("function Gauss-Seidel" in Figure 2). The rest of this paper works exclusively with the graph

that is derived from the subdomains and the matrix graph so that a "node" may be a set of vertices in the finite element mesh.

### 3.1 Algorithm I

Adams presents methods for coloring finite difference stencils so as to parallelize *natural* and *Red/Black* Gauss-Seidel [1] . These methods "pipeline" the computations and are thus not useful when only a few iterations are performed, but preserve the semantics of standard node orderings (which we do not).

A standard method to parallelize Gauss-Seidel on unstructured meshes is to color the nodes, process the nodes of each color, send and receive updated values, and proceeded with the next color. Algorithm I, in Figure 2, is a distributed memory algorithm based on nodal coloring. Given a matrix $A$, vector $x$ with an initial guess, vector $b$ right hand side, a graph of lists of equations and a graph coloring:

**for all colors** $c$
    Send $x$ values needed by other processors to process color $c$
    Receive $x$ values needed for color $c$ /* loose synchronization point */
    **for all nodes (list of equations)** $n$ **with color** $c$
        Gauss-Seidel$(n, A, x, b)$

**function Gauss-Seidel**$(L, A, x, b)$ /* Gauss-Seidel kernel */
    **for all equations** $i \in L$ /* iterate list in reverse on backward pass */
        $t[i] \leftarrow b[i]$ /* buffer vector */
        **for all equations** $j \mid j \notin L, A[i,j] \neq 0$ /* line 4 */
            $t[i] \leftarrow t[i] - A[i,j] \cdot x[j]$
    $x[L] \leftarrow A[L,L]^{-1} \cdot t[L]$

Figure 2: Algorithm I, a simple distributed memory Gauss-Seidel algorithm

There are two main problems with Algorithm I:

- 3D unstructured hexahedral finite element meshes have about 12 or more colors (if vertex blocks are used). This requires many small messages and "loose" synchronization points which are not well suited for common parallel machines.
- This algorithm is perhaps the optimal algorithm for maximizing cache misses and hence minimizing flop rates.

### 3.2 A distributed memory unstructured Gauss-Seidel algorithm

Our algorithm takes advantage of properties of the processor partitions that are common in parallel finite element problems. Namely, nodes are partitioned so as to minimize communication (eg, by minimizing edge cuts) and hence produce highly connected subdomains with many "interior" nodes. Observe that the work on *interior* nodes (that by definition do not have any edges with nodes on other processors) can be used to "hide" the communication required for the boundary nodes. A simple idea is to only color *boundary* nodes (ie, local nodes that are not interior nodes), and process the interior nodes while waiting for results from other processors. But we can do better than that.

Observe that most of the boundary nodes communicate with only one processor, assuming that the processor subdomains are reasonably large. Nodes that communicate with only one processor can be processed all at once (ie, do not have to be restricted to the nodal coloring). This will reduced the number of messages sent and, as many contiguous nodes are processed at once, the order that these nodes are processed in can be optimized to minimize cache misses and hence maximize flop rates.

Processors need to be able to decide who should go first. This can be done by simply comparing processor IDs, or by using a processor coloring. The basic algorithm first colors the processors and orders the colors, this provides an inequality operator for processors. Good coloring (as opposed to the bad coloring provide by processor IDs) is used to reduce the worst case parallel complexity on coarse grids where processor domains

are small. Note, with one node per processor, or nodes randomly partitioned to processors, this algorithm degenerates to Algorithm I. Next, each processor partitions its nodes into interior and boundary nodes as defined above.

The boundary nodes are partitioned into nodes that communicate only with processors that have higher color (call these "Bot" nodes) and nodes that communicate only with processors that have lower color ("Top" nodes) and all of the rest ("Mid" nodes). Note, multigrid requires a symmetric smoother (to be a symmetric preconditioner for a symmetric Krylov method) so a multiplicative smoother must be able to effectively process equations in reverse order as well as in the forward order. To run the algorithm backward the $Top$ nodes are relabeled $Bot$ nodes and visa versa and the node lists are processed in reverse order.

Interior nodes are partitioned into two parts: "Int1" and "Int2" so as to satisfy

$$|Int1| + |Top| = |Int2| + |Bot| \tag{1}$$

where $|L|$ is a measure of the cost (eg, flops) in applying Gauss-Seidel to the equations in the list $L$. "Int1" and "Int2" are further partitioned into two partitions each: "Int1.a", "Int1.b", "Int2.a" and "Int2.b". This partitioning is secondary (our numerical experiments simply put all nodes in the "a" lists). The number of non-zeros in the equations are used as an approximate measure of the cost. Again, to run the algorithm backward, $Int1$ nodes are switched with $Int2$.

$Top$ and $Bot$ nodes are further partitioned into groups that communicate with only one processor so that when an update is received all of the nodes that only depend on that one processor can be processed immediately and care is taken to process these nodes in reverse order on the backward pass.

Figure 3 illustrates these definitions with a diagram of the partitions of a 2D, four processor, problem with the processor colors represented with integers (ie, 1,2,3,4).

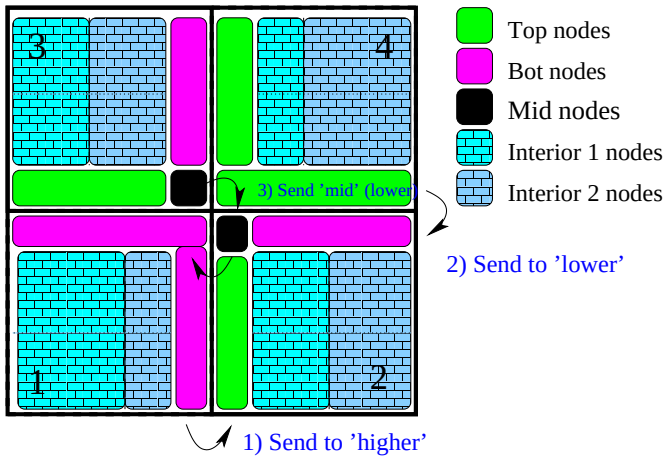


Figure 3: Partitioning diagram for 2D mesh with four processors

Each $Mid$ node ($n$) is equipped with two lists of nodes: 1) all of the neighbor nodes that are *higher* than $n$ and 2) all of the neighbor nodes that are *lower* than $n$ and a pointer to a node list (*dependents*) that is set at the beginning of each iteration to point to the higher node list on a forward pass and the lower node list on a backward pass. Ghost nodes on higher processors are defined as higher nodes, ghost nodes on lower processors are defined as lower nodes and neighbor "Mid" nodes on the same processor are assigned a random number to determine which list they belong to. Nodes are equipped with a flag *done* that is initialized to $false$ at the beginning of each iteration and set to $true$ in the Gauss-Seidel kernel.

With these definitions, and the Gauss-Seidel kernel from Figure 2, one Gauss-Seidel iteration is as follows:

```
Send boundary x values to higher processors        /* Initial send */
Gauss-Seidel(Int1.b, A, x, b)
Receive x values from lower processors /* Initial receive */
Gauss-Seidel(Top, A, x, b) /* one processor nodes can be folded into the previous receive */
Send boundary x values to lower processors
Gauss-Seidel(Int1.a, A, x, b)
Receive x values from higher processors
while ∃n | n ∈ Mid, n.done = false
    Receive boundary x updates and list of ghost nodes
    do
        flag ← false
        for all n ∈ Mid, n.done = false
            if ∀n2 | n2 ∈ n.dependents, n2.done = true
                Gauss-Seidel(n, A, x, b)        sets done flags to true
                Cache updated boundary x values
                flag ← true
    while(flag)
    Send cached boundary x values and list of completed nodes to neighbor processors
Gauss-Seidel(Int2.a, A, x, b) /* one processor nodes are processed after the others */
Receive boundary x updates for any remaining (undone) ghost nodes
Gauss-Seidel(Bot, A, x, b)
Gauss-Seidel(Int2.b, A, x, b)
```

Figure 4: Distributed memory unstructured Gauss-Seidel algorithm

Note, 1) the "Initial" send and receive phase in Figure 4 can be omitted on the second and subsequent iterations, 2) the list in the first line of the "Gauss-Seidel kernel" in Figure 2 must be iterated in reverse when running the algorithm backward, 3) the middle section of this algorithm (the "while" loop that starts on the 8th line) is essentially Algorithm I, and 4) low processors without "Mid" nodes (eg, processor 1 in Figure 5) can postpone the receive on the 7th line of Figure 4 until the data is needed on the second to last line (the processing of "Bot" nodes).

Figure 5 shows a schematic time line for this algorithm for the model problem in Figure 3 with the "Int$x$" node list evenly divided into the "Int$x$.a" and "Int$x$.b" node lists.

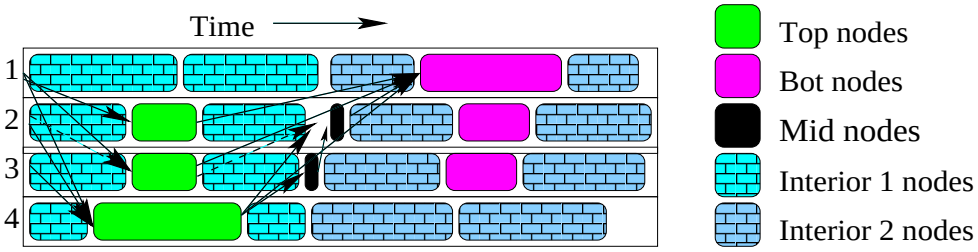


Figure 5: Time line of model 2D problem on a forward pass

Figures 6 and 7 show Vampir outputs of a 2,085,599 degrees of freedom (dof) problem run on 52 Cray T3E and 28 IBM PowerPC processors [13]. The colors on Figures 6 and 7 are similar to those in Figure 3: "Top" work is green, "Bot" work is magenta, the "Int1" work is turquoise, the "Int2" work is blue, "Mid" work is dark purple, and the time waiting in blocking MPI calls is in red. Note, all interior nodes are put in the "a" lists, so that, for instance, each processors work ends with the "Bot" node work (magenta) instead of the "Int2.b" node work (blue) depicted in Figure 5. These figures indicate that we have decent algorithmic efficiency as we do not see too much red areas where processors are waiting for messages. This is promising, but these figures only indicate that the algorithm can work (the numerical results in §5 will quantify this), and are not intended as evidence that the algorithm is effective. Note, the use of Vampir was invaluable in debugging this algorithm (as well as the code), for instance, the form of Equation 1 was refined by finding performance bottlenecks that arise from less optimal versions of this equation (eg, $|Int1| = |Int2|$).

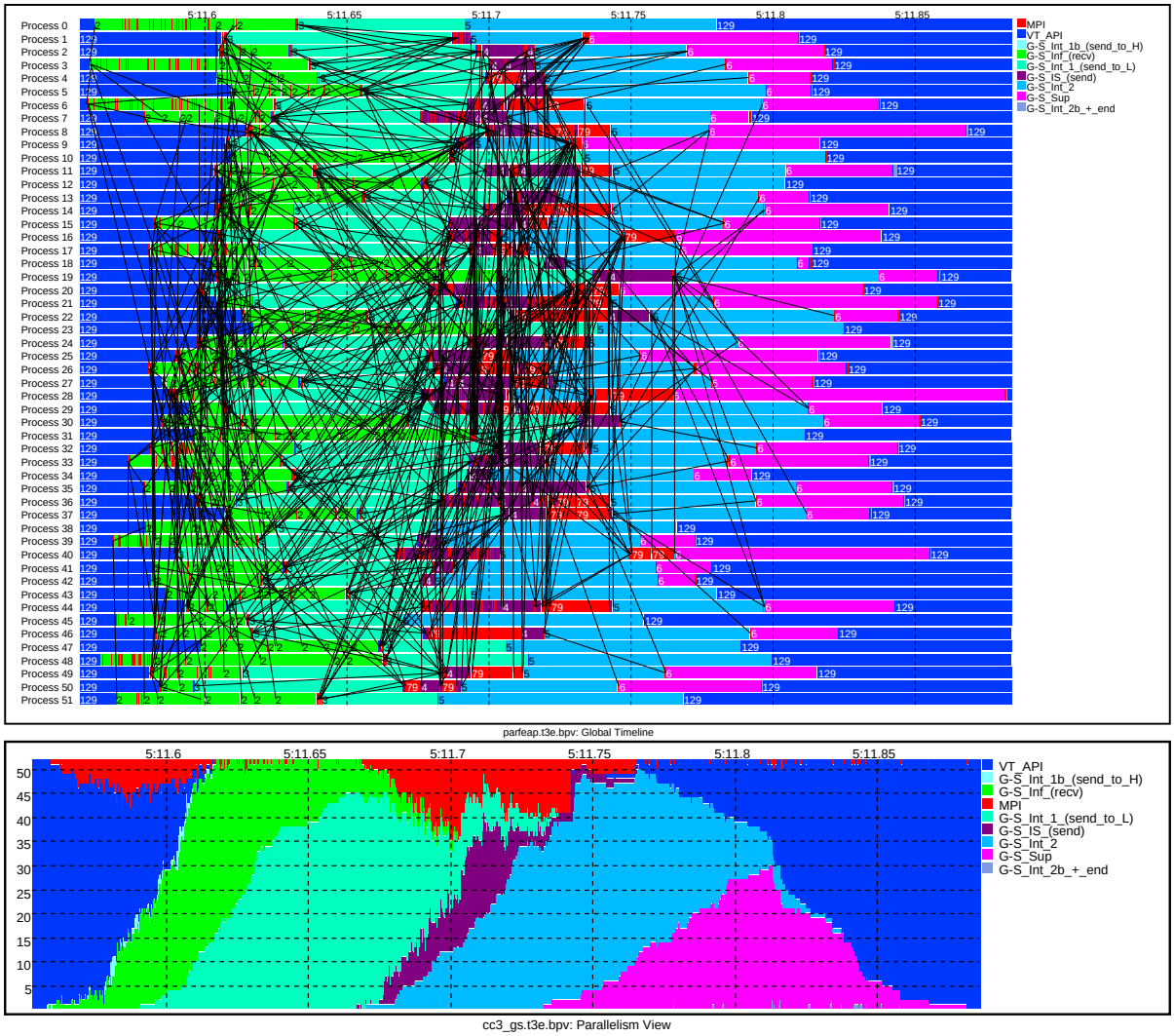


Figure 6: Time line and parallelism for 2M dof problem with 52 Cray T3E processors

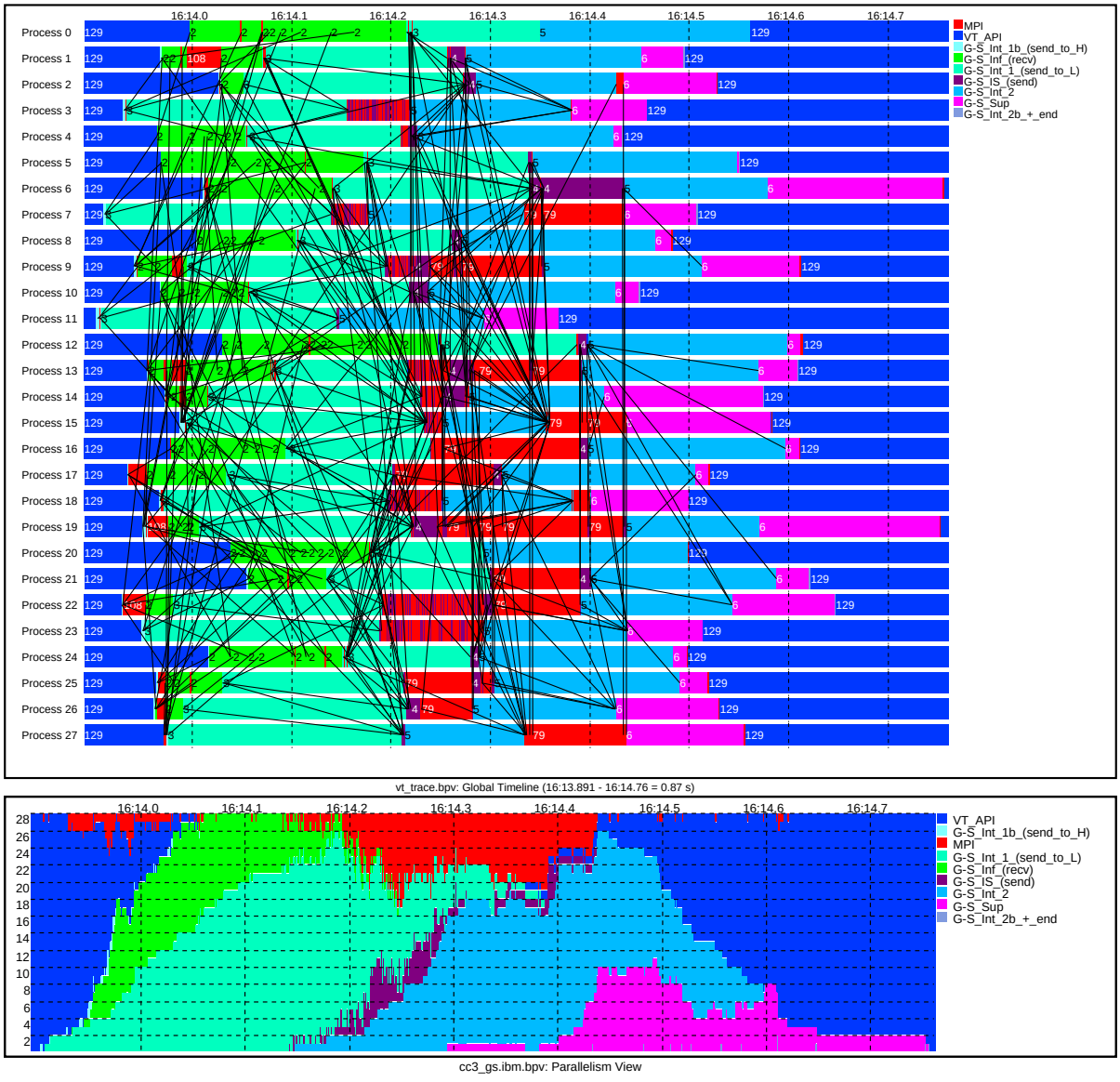


Figure 7: Time line and parallelism for 2M dof problem with 28 IBM PowerPC processors

### 3.3 Algorithm characteristics

Our algorithm utilizes properties of optimal partitioning of 3D finite element problems and is effective because:

- The partitioning that we can expect in practice (ie, from ParMetis) are adequate and we use ParMetis to partition the coarse grids as well as the finest.
- Equation 1 can be satisfied well with respect to flops and reasonably well with respect to time to process these flops (although we have observed some significant variations of flop rates on large problems on some machines).
- Our processor partitions are large enough, and our Schwarz subdomains are small enough, on the finer grids so that only a small percentage of the nodes are processed in the (Algorithm I like) "while" loop in Figure 4.
- The dependency paths for these "Mid" nodes are short (eg, maximum of 3 with ideal partitions, and only for the "corner" nodes of processor partitions, as opposed to 12 or more with Algorithm I) if the processor subdomains are large enough. Additionally these processor subdomains do not have to be very large (eg, about 30 non-overlapped Schwarz subdomains, see Adams for an argument for this property for the parallel maximal independent set problem [2]).

To further characterize the properties of this algorithm we claim that our parallel Gauss-Seidel algorithm is perfectly parallel (ie, has 100% parallel efficiency) for 3D problems under the following assumptions (with comments as to how practical the assumption is):

1. Perfect processor load balancing (this is reasonable).
2. Enough interior nodes to satisfy Equation 1 (easy if the processor domains are large enough, harder with Schwarz domains). This requires at least as much work on the interior nodes as on the boundary nodes and any excess can be used to hide communication as described in assumption 5 below.
3. Optimal flop rates (there can be significant variations in flop rates from processor to processor and the serial flop rates are not optimal as discussed in our numerical results).
4. Zero percent work done on "Mid" nodes (more true as the processor subdomain size increases).
5. Instant communication (obviously not true, but the large messages can be overlapped with computation). This assumption is only needed for messages between "Mid" nodes (eg, the one message between processor 2 and 3 in Figure 5).

These assumptions indicate the potential sources of inefficiency. We can not prove these claims (in fact we know that they are not true) and must rely on numerical experiments to demonstrate the (degree of) effectiveness of this algorithm.

# 4 Parallel architecture

A highly scalable implementation of the algorithms and of a finite element application are used to test the methods. The parallel finite element system *Athena* (Figure 8) is a parallel finite element program built on a serial finite element code (FEAP [9]) and a parallel graph partitioner (ParMetis [14]) and our solver *Prometheus* (Prometheus is freely available in a publicly domain library [16]). Prometheus can be further decomposed into three parts:

- General unstructured multigrid support built on PETSc [5] (*Epimetheus* in Figure 8)
- Non-nested geometric multigrid method (*Prometheus* in Figure 8)
- Aggregation multigrid methods (*Atlas* in Figure 8)

Athena reads a large "flat" finite element mesh input file in parallel, uses ParMetis to partition the finite element graph, and then constructs a complete finite element problem on each processor. These processor sub-problems 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 operator evaluation at the expense of a small amount of redundant work. Thus, these tests use general unstructured software so that, even if the problems are not very complex, the solver is not taking advantage of any of their underlying structure.

Explicit message passing (MPI) is used for performance and portability and all parts of the algorithm have been parallelized for scalability. Clusters of symmetric multi-processors (SMPs) are targeted as this seems to be the architecture of choice for future large machines. Clusters of SMPs are accommodated by first partitioning the problem onto the SMPs and then the local problem is partitioned onto each processor as depicted in Figure 8. This approach implicitly takes advantage of any increase in communication performance within each SMP, though the numerical kernels (in PETSc) are "flat" MPI codes. Prometheus assumes that the provided fine grid is partitioned well but repartitions the (internally constructed) coarse grids with ParMetis to maintain load balance.

The parallel application of multigrid adds a $\log(n)$ term to the parallel complexity as some processors must remain idle on the coarsest grids on very large problems. Given the number of degrees of freedom per processor and the number processors in our numerical experiments the $\log(n)$ term is not significant as most of the flops are performed on grids that are "active" on all processors. But as problems get larger this will become more important. The number of processors is reduced on the coarsest grids when there are few equations per processor particularly on machines with poor communication infrastructure. The reasons for this are two fold: 1) it is difficult to implement the parallel construction of the coarse grid spaces to have the exact serial semantics in the regions between processors and 2) most machines are not modeled accurately with the PRAM complexity model (ie, the coarsest grids on large problems can actually run faster if fewer processors are used as the latencies in the dot products can dominate). Our solver implementation thus, reduces the number of active processors on the coarsest grids to try to keep a minimum of about 200 equations per processor.

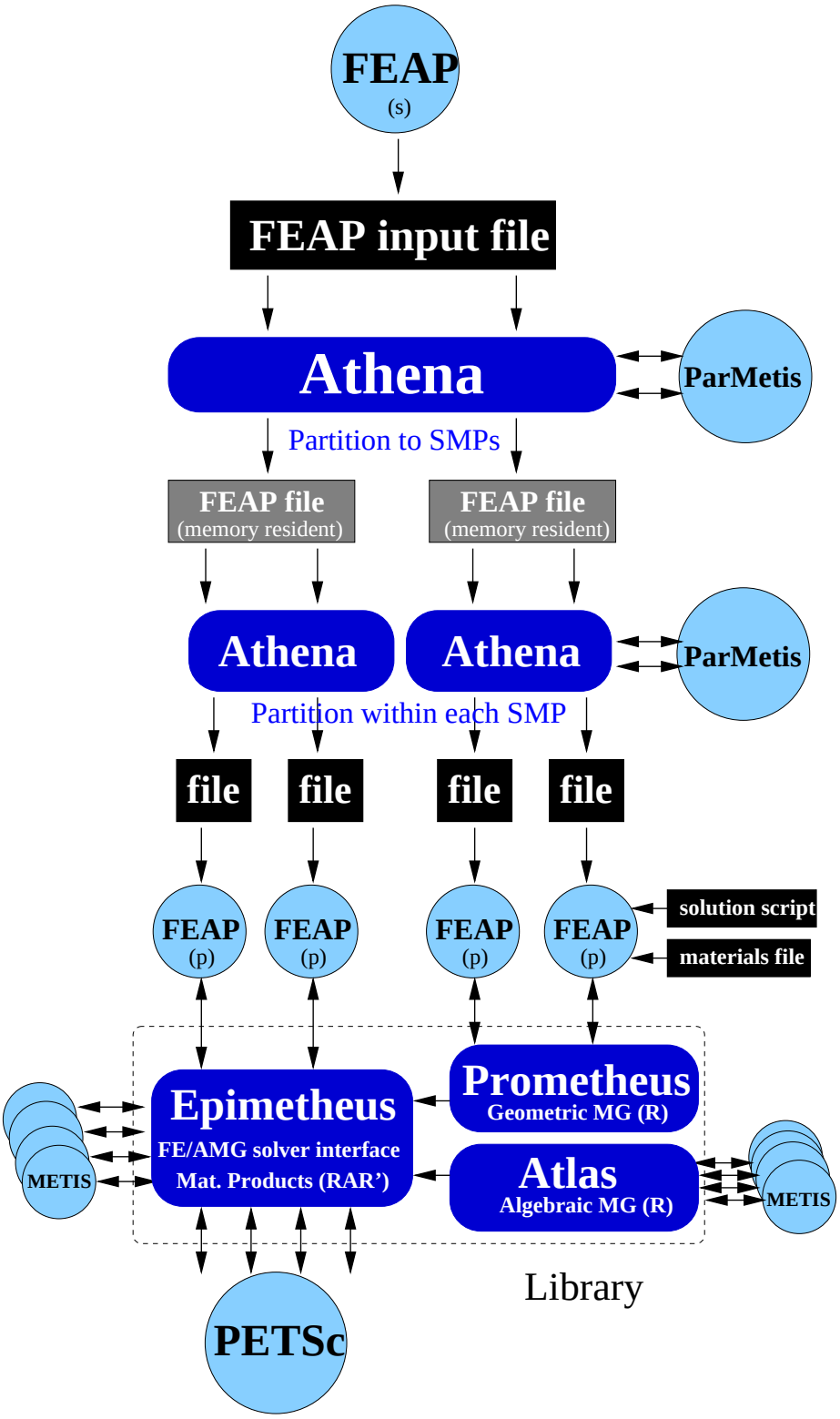


Figure 8: Code Architecture

# 5 Numerical results

To evaluate the performance of parallel Gauss-Seidel one must first look at the serial performance. One disadvantage of Gauss-Seidel is that it is not easy to use standard numerical kernel codes (eg, PETSc's matrix vector product), thus requiring that the Gauss-Seidel kernels be hand coded. The work in the Gauss-Seidel kernel is similar to the work of a matrix-vector product and thus we would hope to be able to achieve similar performance. There are two primary differences, however, between the Gauss-Seidel kernel in our algorithm and a matrix-vector product:

- The Gauss-Seidel kernel must work on a subset of the rows in the matrix, in several stages, and the order of the operations must be done carefully to maintain the "multiplicative" semantics. For instance, PETSc sends messages, then computes with the diagonal processor block of the matrix, then receives messages and then computes with the off-diagonal processor block to optimize performance. The Gauss-Seidel kernel does not have this freedom.
- The Gauss-Seidel kernel must skip the diagonal block of the equations in each "node", thus requiring a test in the inner loop (line 4 in "function Gauss-Seidel" in Figure 2). This test is a simple "if" test in the nodal block case but is more complex in the more general case (ie, Schwarz blocks). This additional complexity is reflected in the "block Gauss-Seidel" flop rates in Table 1.

Table 1 shows the per processor Mflop rates (using one or two processors), on the fine grid of the 79,679 dof problem described in §11 for: 1) theoretical peak, 2) PETSc's matrix-vector product, 3) block Gauss-Seidel (about 42 vertices, 126 equations, per block) and 4) nodal block Gauss-Seidel.

| Machine | Peak | Mat-Vec | block Gauss-Seidel | nodal G-S (% of Mat-Vec) |
|---|---|---|---|---|
| Cray T3E | 950 | 88 | 27 | 46 (52 %) |
| IBM SP PowerPC | 634 | 36 | 21 | 31 (82 %) |
| IBM SP Power3 | 1500 | 151 | 46 | 115 (76 %) |
| Intel (Sandia Red) | 200 | 41 | 22 | 34 (83 %) |
| Compac (DEC alpha) | 880 | 58 | 24 | 36 (62 %) |
| Sun Enterprise 10K | 666 | 30 | 16 | 25 (83 %) |

Table 1: Machine performance (Mflops/sec)

The serial inefficiencies shown in Table 1 are caused by several factors such as there is extra bookkeeping required to implement the algorithm (especially for block Gauss-Seidel as describe above), there are auxiliary data structures required that may cause cache conflicts with the primary data and this algorithm dictates a node ordering that is not the same as the native matrix. That is, this algorithm precludes running straight through the matrix, or vertically partitioning the matrix, as can be done in a matrix vector product. This inefficiency could be reduced by making a separate copy of the matrix - without the diagonal block - in the order of the (forward pass of the) algorithm to get better flop rates, this would require more storage and we have not tested this approach.

## 5.1 Shell problem

This problem is a "wing" with fully clamped boundary condition at the base and a uniform load down on the under side of the wing. The wing is meshed with four node quadrilateral shell elements, with thickness of $\frac{1}{2000}$ times the length of the wing, has 2,248,470 degrees of freedom and four internal stiffener plates. This linearized stiffness matrix has a condition number of about $1.0 \cdot 10^9$. Figure 9 shows the deformed mesh with the first principle stress of a 49,980 dof version of the problem.

Conjugate gradients (CG) is used as the solver, preconditioned with one iteration of full multigrid. Three smoothers are tested: One pre and post smoothing step with 2248 Schwarz subdomains of 1) CG preconditioned with block Jacobi (additive Schwarz), 2) block Gauss-Seidel (multiplicative Schwarz) and 3) one pre and post smoothing steps of nodal Gauss-Seidel. The Schwarz subdomains are constructed with METIS with about 1000 equations per block. The choice of number of smoothing step was made by

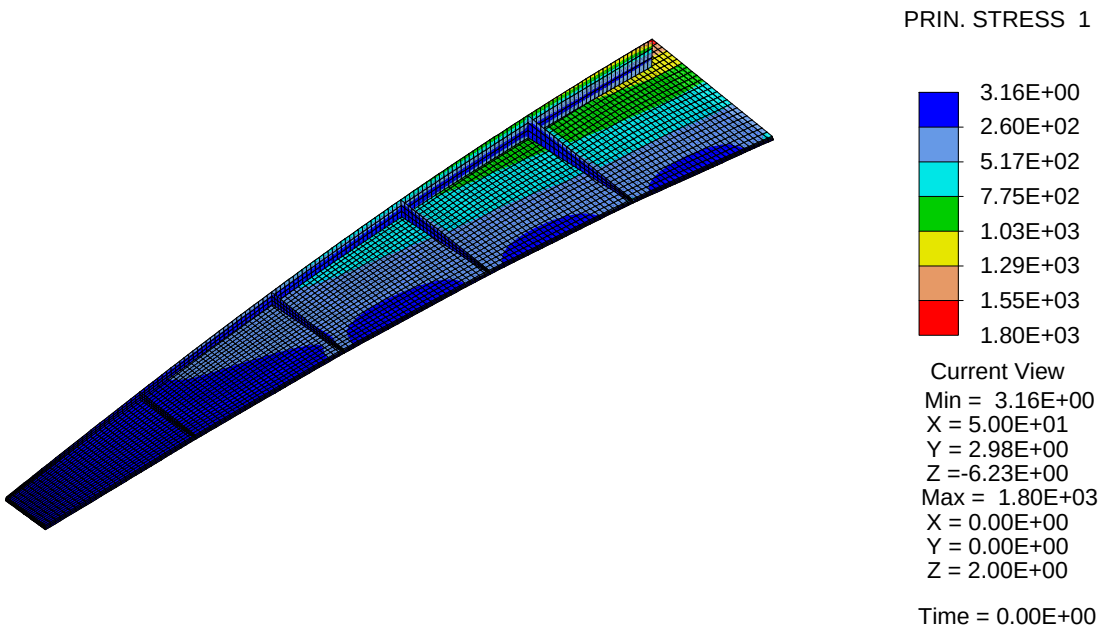


Figure 9: Wing deformed shape

selecting the number, for each smoother, that performed best. Smoothed aggregation algebraic multigrid is used [18, 3], with a relative residual tolerance of $10^{-6}$. Figure 10 (left) shows the residual history of this problem in units of the time for one matrix-vector product on the fine grid. Figure 10 (right) shows the the time for the 1) "mesh setup" (eg, coarse grid construction), 2) the "matrix setup" (eg, subdomain factorizations and coarse grid operator construction) and 3) the "solve" time (ie, time in the CG solver), on 32 PowerPC processors.

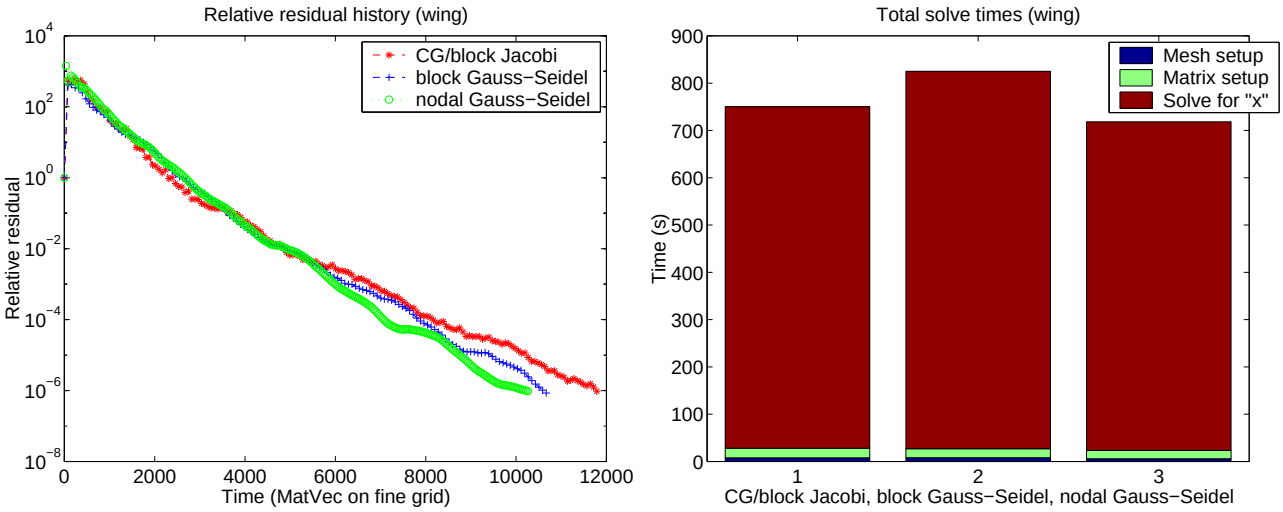


Figure 10: Residual history vs. solve times (left), and sum of mesh setup, matrix setup and solve times (right), on 32 PowerPC processors

This data shows that all three smoothers provide remarkably similar performance.

## 5.2 Scalability studies

This test problem is a series of thin concentric spheres enclosed in a "soft" material (with symmetric boundary conditions so that only one octant need be modeled). The sphere is composed of seventeen alternating layers of *hard* and *soft* materials; Table 2 shows a summary of the constitution of the two material types.

| Material | Elastic modulus | Poisson ratio |
|---|---|---|
| soft | $10^{-4}$ | 0.49 |
| hard | 1 | 0.3 |

Table 2: Materials

The loading and boundary conditions are an imposed uniform displacement (down), on the top surface.

The mesh is parameterized for these scalability studies. Each successive problem has one more layer of elements through each of the seventeen shell layers, with a similar refinement in the other two directions, and in the outer soft domain. The problems range in size from 80K to 76M degrees of freedom. Figure 11 shows the smallest version of the problem with 79,679 dof.

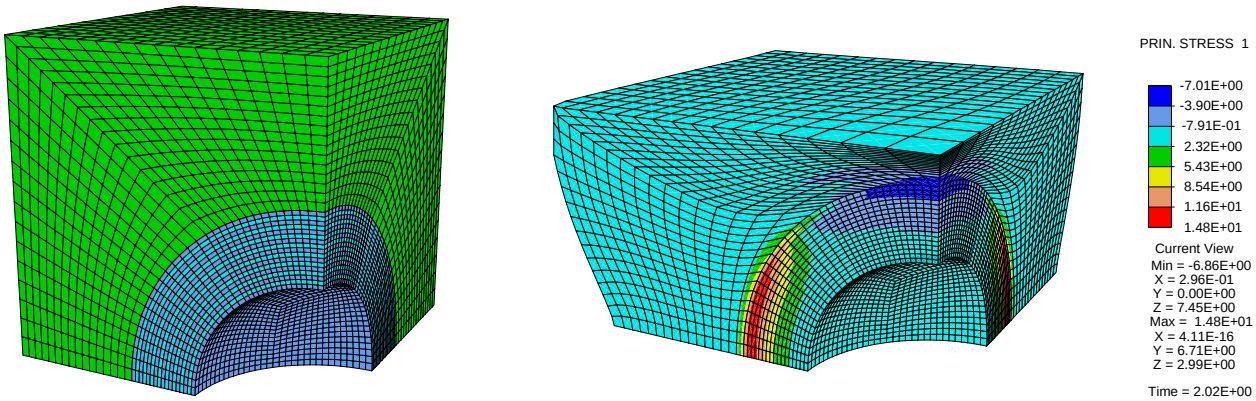


Figure 11: 79,679 dof concentric spheres problem

Conjugate gradients (CG) is used as the solver, preconditioned with one iteration of full multigrid. One pre and post smoothing step is used for 1) CG preconditioned with block Jacobi (additive Schwarz), 2) block Gauss-Seidel (multiplicative Schwarz) and 3) nodal block Gauss-Seidel. The blocks for block Jacobi and block Gauss-Seidel are constructed with METIS with about 125 equations per block. The choice of one smoothing step was made by selecting the number, for each smoother, that performed best on the 640K dof version of this problem. Two unstructured multigrid methods will be used for these studies:

- Non-nested geometric multigrid [4].
- Smoothed aggregation algebraic multigrid [18, 3].

All solves use a relative residual tolerance of $10^{-6}$.

### 5.2.1 IBM SP

The number of processors used is selected to keep about 80K dof per processor, from one to 960 processors on an IBM PowerPC cluster. Due to significant variation in the flop rate on this machine from one run to the next, especially on larger problems, the best results are shown from several runs of each problem - except for the 960 processor case where, due to lack of access to the machine, we were limited in the number of experiments that we could perform and at press time were not able to run one test. Updated versions of this paper will be available on my web page [16]. Figure 12 shows the iteration counts (left), Mflop rate (right), and Figure 13 shows the solve times for the three smoothers and the two multigrid methods.

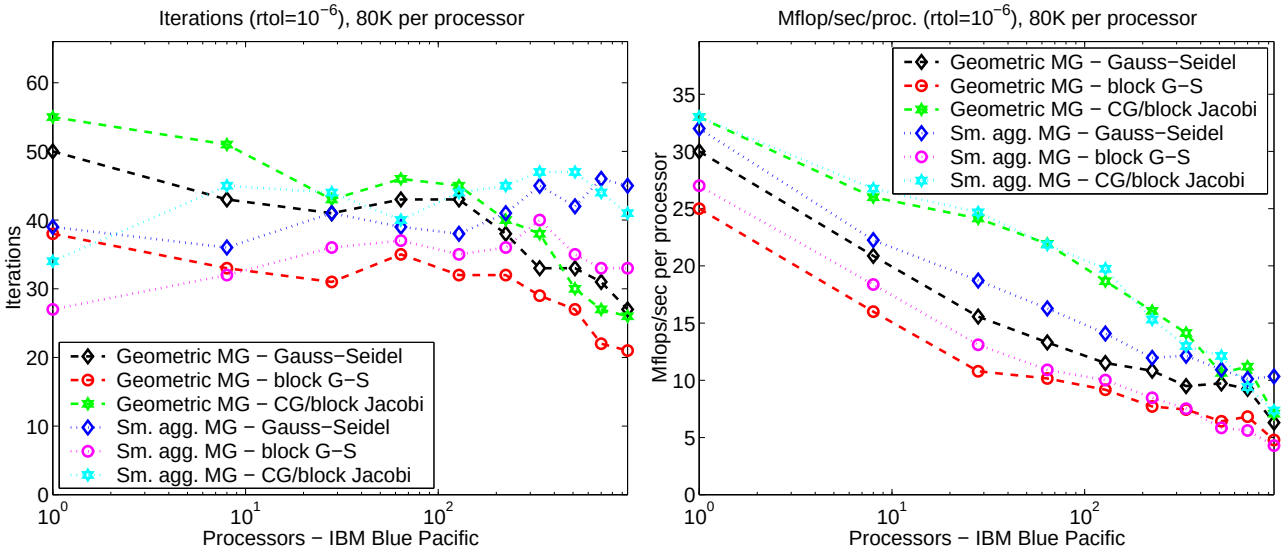


Figure 12: Iteration counts and flop rates on an IBM PowerPC cluster

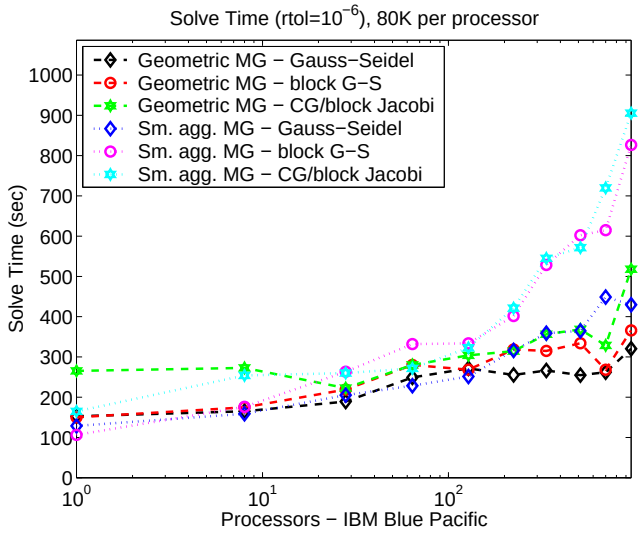


Figure 13: Total solve times on an IBM PowerPC cluster

The convergence rate (ie, the inverse of the number of iterations, Figure 12, left) is best for the block Gauss-Seidel as is expected as Gauss-Seidel has better convergence properties than damped Jacobi on model problems, and the convergence rate is about the same for the nodal Gauss-Seidel and the CG/block Jacobi. We have as high as 31% parallel efficiency for the flop rates (ie, 10 Mflops per processor on 960 processors vs. 32 Mflops on one processor for smoothed aggregation multigrid with nodal Gauss-Seidel smoothing). The ultimate parallel efficiency (solve time on one processor divided by solve time on the largest run) is as high as about 44%, for geometric multigrid with nodal Gauss-Seidel smoothing from one to 960 processors.

There is significant deterioration in the flop rate for smoothed aggregation with the two block smoothers, on the larger problems. We believe that this is due to cache effects; smoothed aggregation uses more memory than geometric multigrid because the interpolation operators are quite large (about one third the size of the stiffness matrix). This combined with the extra memory needed for the block smoothers increases pressure on the cache (the IBM has about 1Gb of usable memory per four processor node and our total program is

using all available memory as reported by the "jr" command at LLNL and is paging. We do not believe that there is paging within the solve, but there may be some at the beginning of the solve ). This deterioration in flop rates is probably exacerbated by a tendency for performance on this machine to degrade after a reboot as the problem is most pronounced on the larger problems. This performance degradation is due to memory management issues (as observed by us and others according to the LLNL support staff); we were not able access a freshly rebooted machine at press time.

### 5.2.2 ASCI Red

The number of processors used is selected to keep about 40K dof per processor (less than the IBM for lack of memory), from two to 1920 processors on the ASCI Red machine at Sandia National Laboratory. Figure 14 shows the iteration counts (left), Mflop rate (right), and Figure 15 shows the solve times for the three smoothers.

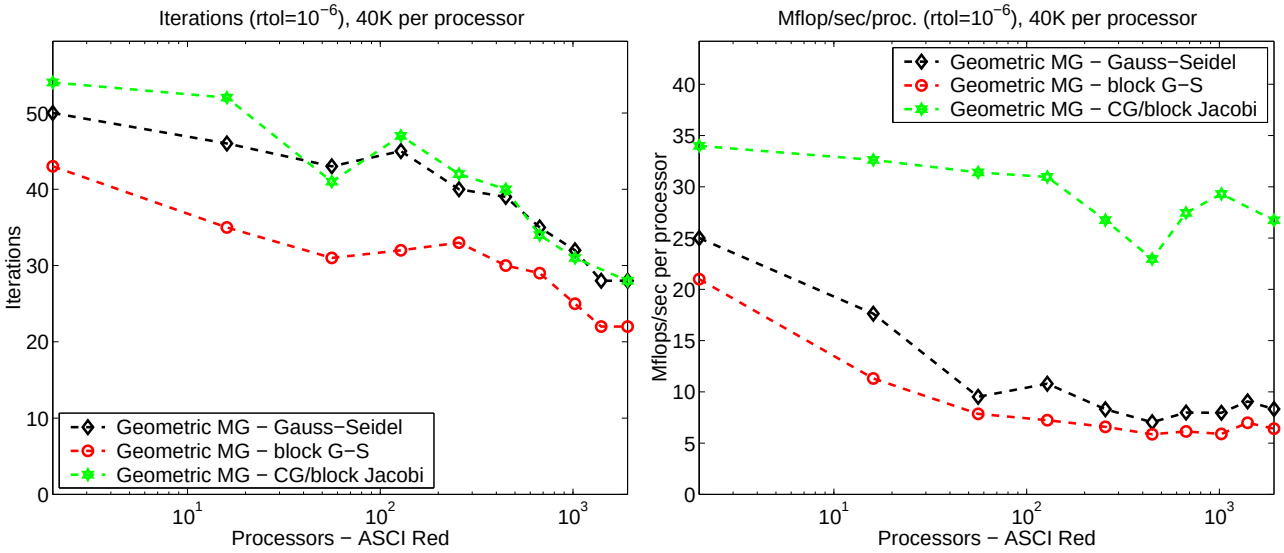


Figure 14: Iteration counts and flop rates on ASCI Red

From this data we notice that convergence rate (ie, the inverse of the number of iterations) is about the same for the nodal Gauss-Seidel and the CG/block Jacobi and best for the block Gauss-Seidel as in the IBM results, as expected. We have about 33% parallel efficiency for the flop rates on the Gauss-Seidel smoothers (eg, 9 Mflops per processor on 1920 processors vs. 27 Mflops per processor on two processors). The ultimate parallel efficiency (solve time on one processor divided by solve time on the largest run) is about 70% for geometric multigrid with Gauss-Seidel smoothing.

# 6 Conclusion

We have shown that Gauss-Seidel can be effectively implemented and used as a multigrid smoother on distributed memory computers and can provide a viable alternative to preconditioned conjugate gradients for unstructured finite element problems, provided that there are enough equations on each processor. A natural optimization, that we have not investigated, is to use Gauss-Seidel on the finest grids, where it performs best in terms of flop rates, and preconditioned conjugate gradients on the coarsest grids with perhaps an increase in smoothing steps or larger Schwarz subdomains to balance the superior convergence properties of Gauss-Seidel. The stationary character of Gauss-Seidel is also very valuable for non-symmetric problems. We have thus added a valuable resource to the available tools for multigrid solvers on distributed memory machines.

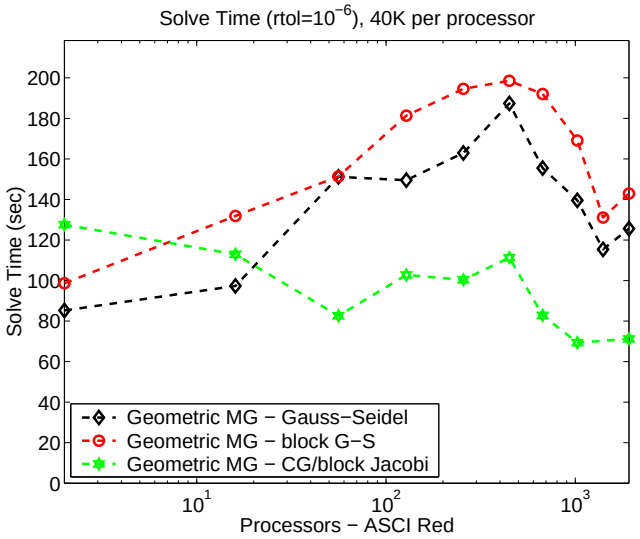


Figure 15: Total solve times on ASCI Red

Future work is to improving the serial performance of the Gauss-Seidel kernel. Some potential areas to investigate are:

- Further minimize the data structures used during the Gauss-Seidel solve.
- Improved node orderings (within node partitions) to optimize cache performance [8].

Additionally, the static partitioning, that uses Equation 1, to balance the work on each side of the $Mid$ node work could be replace with a dynamic partitioning that measures wait time, in the $Mid$ section of the algorithm in Figure 4, and moves nodes between the $Int1.a$ and $Int2.a$ partitions to minimize these wait times and thus accommodate nonuniform communication costs and other forms of load imbalance.

**Acknowledgments.** I would like to thank the reviewers for their many helpful suggestions. I would like to thank the many people that have contributed libraries to this work: R.L. Taylor for providing FEAP, the PETSc team for providing PETSc, George Karypis for providing ParMetis/METIS. I would also like to thank 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 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. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000.

# References

[1] Loyce M. Adams and Harry F. Jordan. Is SOR color-blind? *SIAM J. Sci. Statist. Comput.*, 7(2):490–506, 1986.

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

[3] M. F. Adams. Evaluation of three unstructured multigrid methods on 3D finite element problems in solid mechanics. Technical Report UCB//CSD-00-1103, University of California, Berkeley, 2000.

[4] M. F. Adams. Parallel multigrid solvers for 3D unstructured finite element problems in large deformation elasticity and plasticity. *International Journal for Numerical Methods in Engineering*, 48(8):1241–1262, 2000.

[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] 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*, 38:3529–3544, 1995.

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

[8] C. C. Douglas, J. Hu, M. Iskandarani, M. Kowarschik, U. Rüde, and C. Weiss. Maximizing cache memory usage for multigrid algorithms. In *Multiphase Flows and Transport in Porous Media: State of the Art*, pages 124–137. Springer, Berlin, 2000.

[9] FEAP. www.ce.berkeley.edu/∼rlt.

[10] Y. T. Feng, D Peric, and D. R. J. Owen. A non-nested multi-grid method for solving linear and nonlinear solid mechanics problems. *Compute. Meth. Mech. Engng.*, 144:307–325, 1997.

[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] V.E. Henson and U.M. Yang. BoomerAMG: A parallel algebraic multigrid solver and preconditioner. Technical Report UCRL-JC-139098, Lawrence Livermore National Laboratory, 2000. To appear in Applied Numerical Mathematics.

[13] http://www.pallas.de/pages/vampir.htm. Vampir 2.5 - visualization and analysis of mpi programs.

[14] G. Karypis and V. Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. *ACM/IEEE Proceedings of SC96: High Performance Networking and Computing*, 1996.

[15] MGNet. www.mgnet.org.

[16] Prometheus. www.cs.berkeley.edu/∼madams.

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

[18] 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.