ELSEVIER SCIENCE B.V. [DTD 4.1.0]
JOURNAL PARCO ARTICLE No. 1590
PAGES 01-26 DISPATCH 21 December 2000 PROD. TYPE: FROM DISK
PARCO 1590
A

ELSEVIER
Parallel Computing 000 (2000) 000–000
PARALLEL COMPUTING
www.elsevier.com/locate/parco

High-performance parallel implicit CFD

William D. Gropp [a,1], Dinesh K. Kaushik [a,2], David E. Keyes [b,*,3], Barry F. Smith [a]

[a] *Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, USA*
[b] *Mathematics and Statistics Department, Old Dominion University, Norfolk, VA 23529, USA*

Received 7 August 2000; received in revised form 4 October 2000

Abstract

Fluid dynamical simulations based on finite discretizations on (quasi-)static grids scale well in parallel, but execute at a disappointing percentage of per-processor peak floating point operation rates without special attention to layout and access ordering of data. We document both claims from our experience with an unstructured grid CFD code that is typical of the state of the practice at NASA. These basic performance characteristics of PDE-based codes can be understood with surprisingly simple models, for which we quote earlier work, presenting primarily experimental results. The performance models and experimental results motivate algorithmic and software practices that lead to improvements in both parallel scalability and per node performance. This snapshot of ongoing work updates our 1999 Bell Prize-winning simulation on ASCI computers. © 2000 Elsevier Science B.V. All rights reserved.

MSC: 65H20; 65N55; 65Y05; 76G25

[*] Corresponding author. Tel.: +1-757-683-3906; fax: +1-757-683-3885.
E-mail address: dkeyes@odu.edu (D.E. Keyes).
[1] This work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.
[2] This work was supported by a GAANN Fellowship from the U.S. Department of Education and by Argonne National Laboratory under contract 983572401.
[3] This work was supported by the National Science Foundation under grant ECS-9527169, by NASA under contracts NAS1-97046 and NAS1-19480 (while the author was in residence at ICASE), by Argonne National Laboratory under contract 982232402, and by Lawrence Livermore National Laboratory under subcontract B347882.

Keywords: Parallel implicit solvers; Unstructured grids; Computational fluid dynamics; High-performance computing

1. PDE application overview

Systems modeled by partial differential equations often possess a wide range of time scales – some (or all, in steady-state problems) much faster than the phenomena of interest – suggesting the need for implicit methods. In addition, many applications are geometrically complex, suggesting the convenience of an unstructured mesh for fast automated grid generation. The best algorithms for solving nonlinear implicit problems are often Newton methods, which in turn require the solution of very large, sparse linear systems. The best algorithms for these sparse linear problems, particularly at very large sizes, are often preconditioned iterative methods, of multilevel type if necessary. This nested hierarchy of tunable algorithms has proved effective in solving complex problems in such areas as aerodynamics, combustion, radiation transport, and global circulation.

When well tuned, such codes spend almost all of their time in two phases: flux computations (to evaluate conservation law residuals), where one aims to have such codes spent almost *all* their time, and sparse linear algebraic kernels, which are a fact of life in implicit methods. Altogether, four basic groups of tasks can be identified based on the criteria of arithmetic concurrency, communication patterns, and the ratio of operation complexity to data size within the task. These four distinct phases, present in most implicit codes, are vertex-based loops, edge-based loops, recurrences, and global reductions. Each of these groups of tasks stresses a different subsystem of contemporary high-performance computers. Analysis of our demonstration code shows that, after tuning, the linear algebraic kernels run at close to the aggregate memory-bandwidth limit on performance, the flux computations are bounded either by memory bandwidth or instruction scheduling (depending upon the ratio of load/store units to floating-point units in the CPU), and parallel efficiency is bounded primarily by slight load imbalances at synchronization points.

Our demonstration application code, FUN3D, is a tetrahedral, vertex-centered unstructured mesh code originally developed by W.K. Anderson of the NASA Langley Research Center for compressible and incompressible Euler and Navier–Stokes equations [1,2]. FUN3D uses a control volume discretization with a variable-order Roe scheme for approximating the convective fluxes and a Galerkin discretization for the viscous terms. FUN3D has been used for design optimization of airplanes, automobiles, and submarines, with irregular meshes comprising several million mesh points. The optimization involves many analyses, typically sequential. Thus, reaching the steady-state solution in each analysis cycle in a reasonable amount of time is crucial to conducting the design optimization. Our best achievement to date for multimillion meshpoint simulations is about 15 μs per degree-of-freedom for satisfaction of residuals close to machine precision.

We have ported FUN3D into the PETSc [3] framework using the single program multiple data (SPMD) message-passing programming model, supplemented by multithreading at the physically shared memory level. Thus far, our large-scale parallel experience with PETSc-FUN3D is with compressible or incompressible Euler flows, but nothing in the solution algorithms or software changes when additional physical phenomenology present in the original FUN3D is included. Of course, the convergence rate varies with conditioning, as determined by Mach and Reynolds numbers and the correspondingly induced mesh adaptivity. Robustness becomes an issue in problems that admit shocks or employ turbulence models. When nonlinear robustness is restored in the usual manner, through pseudo-transient continuation, the conditioning of the linear inner iterations is enhanced, and parallel scalability may be improved. In some sense, the subsonic Euler examples on which we concentrate, with their smaller number of flops per point per iteration and their aggressive pseudotransient buildup toward the steady-state limit, may be a *more* severe test of parallel performance than more physically complex cases.

Achieving high sustained performance, in terms of solutions per second, requires attention to three factors. The first is a scalable implementation, in the sense that time per iteration is reduced in inverse proportion to the number of processors, or that time per iteration is constant as problem size and processor number are scaled proportionally. The second is good per processor performance on contemporary cache-based microprocessors. The third is algorithmic scalability, in the sense that the number of iterations to convergence does not grow with increased numbers of processors. The third factor arises because the requirement of a scalable implementation generally forces parameterized changes in the algorithm as the number of processors grows. If the convergence is allowed to degrade, however, the overall execution is not scalable, and this must be countered algorithmically. These factors in the overall performance are considered in Sections 3–5, respectively, which are the heart of this paper. Section 2 first expands on the algorithmics. Section 6 details our highest performing runs to date, and Section 7 summarizes our work and looks ahead.

2. ΨNKS: a family of parallel implicit solution algorithms

Our implicit algorithmic framework for advancing toward an assumed steady state for the system of conservation equations, $\mathbf{f}(\mathbf{u}) = 0$, has the form

$$\left(\frac{1}{\Delta t^\ell}\right)\mathbf{u}^\ell + \mathbf{f}(\mathbf{u}^\ell) = \left(\frac{1}{\Delta t^\ell}\right)\mathbf{u}^{\ell-1},$$

where $\Delta t^\ell \to \infty$ as $\ell \to \infty$, $\mathbf{u}$ represents the fully coupled vector of unknowns, and $\mathbf{f}(\mathbf{u})$ is the vector of nonlinear conservation laws.

Each member of the sequence of nonlinear problems, $\ell = 1, 2, \ldots$, is solved with an inexact Newton method. The resulting Jacobian systems for the Newton corrections are solved with a Krylov method, relying directly only on matrix-free Jacobian-vector product operations. The Krylov method needs to be preconditioned for acceptable inner iteration convergence rates, and the preconditioning can be the "make-or-break" feature of an implicit code. A good preconditioner saves time and space by permitting fewer iterations in the Krylov loop and smaller storage for the Krylov subspace. An additive Schwarz preconditioner [5] accomplishes this in a concurrent, localized manner, with an approximate solve in each subdomain of a partitioning of the global PDE domain. The coefficients for the preconditioning operator are derived from a lower-order, sparser and more diffusive discretization than that used for $\mathbf{f}(\mathbf{u})$, itself. Applying any preconditioner in an additive Schwarz manner tends to increase flop rates over the same preconditioner applied globally, since the smaller subdomain blocks maintain better cache residency, even apart from concurrency considerations [28]. Combining a Schwarz preconditioner with a Krylov iteration method inside an inexact Newton method leads to a synergistic, parallelizable nonlinear boundary value problem solver with a classical name: Newton–Krylov–Schwarz (NKS) [12]. We combine NKS with pseudo-timestepping [17] and use the shorthand ΨNKS to describe the algorithm.

To implement ΨNKS in FUN3D, we employ the PETSc package [3], which features distributed data structures – index sets, vectors, and matrices – as fundamental objects. Iterative linear and nonlinear solvers are implemented within PETSc in a data structure-neutral manner, providing a uniform application programmer interface. Portability is achieved through MPI, but message-passing detail is not required in the application. We use MeTiS [16] to partition the unstructured mesh.

The basic philosophy of any efficient parallel computation is "owner computes," with message merging and overlap of communication with computation where possible via split transactions. Each processor "ghosts" its stencil dependencies on its neighbors' data. Grid functions are mapped from a global (user) ordering into contiguous local orderings (which, in unstructured cases, are designed to maximize spatial locality for cache line reuse). Scatter/gather operations are created between local sequential vectors and global distributed vectors, based on runtime-deduced connectivity patterns.

As mentioned above, there are four groups of tasks in a typical PDE solver, each with a distinct proportion of work to datasize to communication requirements. In the language of a vertex-centered code, in which the data are stored at cell vertices, these tasks are as follows:

- Vertex-based loops
  - state vector and auxiliary vector updates
- Edge-based "stencil op" loops
  - residual evaluation, Jacobian evaluation
  - Jacobian-vector product (often replaced with matrix-free form, involving residual evaluation)
  - interpolation between grid levels
- Sparse, narrow-band recurrences
  - (approximate) factorization, back substitution, relaxation/smoothing
- Vector inner products and norms
  - orthogonalization/conjugation
  - convergence progress checks and stability heuristics.

Vertex-based loops are characterized by work closely proportional to datasize, pointwise concurrency, and no communication.

Edge-based "stencil op" loops have a large ratio of work to datasize, since each vertex is used in many discrete stencil operations, and each degree of freedom at a point (momenta, energy, density, species concentration) generally interacts with all others in the conservation laws – through constitutive and state relationships or directly. There is concurrency at the level of the number of edges between vertices (or, at worst, the number of edges of a given "color" when write consistency needs to be protected through mesh coloring). There is local communication between processors sharing ownership of the vertices in a stencil.

Sparse, narrow-band recurrences involve work closely proportional to data size, the matrix being the largest data object and each of its elements typically being used once. Concurrency is at the level of the number of fronts in the recurrence, which may vary with the level of exactness of the recurrence. In a preconditioned iterative method, the recurrences are typically broken to deliver a prescribed process concurrency; only the quality of the preconditioning is thereby affected, not the final result. Depending upon whether one uses a pure decomposed Schwarz-type preconditioner, a truncated incomplete solve, or an exact solve, there may be no, local only, or global communication in this task.

Vector inner products and norms involve work closely proportional to data size, mostly pointwise concurrency, and global communication. Unfortunately, inner products and norms occur rather frequently in stable, robust linear and nonlinear methods.

Based on these characteristics, one anticipates that vertex-based loops, recurrences, and inner products will be *memory bandwidth limited*, whereas edge-based loops are likely to be only *load/store limited*. However, edge-based loops are vulnerable to *internode bandwidth* if the latter does not scale. Inner products are vulnerable to *internode latency* and *network diameter*. Recurrences can resemble some combination of edge-based loops and inner products in their communication characteristics if preconditioning fancier than simple Schwarz is employed. For instance, if incomplete factorization is employed globally or a coarse grid is used in a multilevel preconditioner, global recurrences ensue.

3. Implementation scalability

Domain-decomposed parallelism for PDEs is a natural means of overcoming Amdahl's law in the limit of fixed problem size per processor. Computational work on each evaluation of the conservation residuals scales as the volume of the (equal-sized) subdomains, whereas communication overhead scales only as the surface. This ratio is fixed when problem size and processors are scaled in proportion, leaving only global reduction operations over all processors as an impediment to perfect performance scaling.

In [18], it is shown that on contemporary tightly coupled parallel architectures in which the number of connections between processors grows in proportion to the number of processors, such as meshes and tori, aggregate internode bandwidth is more than sufficient, and limits to scalability may be determined by a balance of work per node to synchronization frequency. On the other hand, if there is nearest-neighbor communication contention, in which a fixed resource like an internet switch is divided among all processors, the number of processors is allowed to grow only as the one-fourth power of the problem size (in three dimensions). This is a curse of typical Beowulf-type clusters with inexpensive networks; we do not discuss the problem here, although it is an important practical limitation in many CFD groups.

When the load is perfectly balanced (which is easy to achieve for static meshes) and local communication is not an issue because the network is scalable, the optimal number of processors is related to the network diameter. For logarithmic networks, like a hypercube, the optimal number of processors, P, grows directly in proportion to the problem size, N. For a d-dimensional torus network, $P \propto N^{d/d+1}$. The proportionality constant is a ratio of work per subdomain to the product of synchronization frequency and internode communication latency.

3.1. Scalability bottlenecks

In Table 1, we present a closer look at the relative cost of computation for PETSc-FUN3D for a fixed-size problem of 2.8 million vertices on the ASCI Red machine, from 128 to 3072 nodes. The intent here is to identify the factors that retard the scalability.

From Table 1, we observe that the buffer-to-buffer time for global reductions for these runs is relatively small and does not grow on this excellent network. The primary factors responsible for the increased overhead of communication are the implicit synchronizations and the ghost point updates (interprocessor data scatters).

Interestingly, the increase in the percentage of time (3–10%) for the scatters results more from algorithmic issues than from hardware/software limitations. With an increase in the number of subdomains, the percentage of grid point data that must be communicated also rises. For example, the total amount of nearest neighbor data that must be communicated per iteration for 128 subdomains is 2 gigabytes, while for 3072 subdomains it is 8 gigabytes. Although more network wires are available when more processors are employed, scatter time increases. If problem size and

Table 1
Scalability bottlenecks on ASCI Red for a fixed-size 2.8 M-vertex case[a]

Number of processors	Percentage of time		
	Global reductions	Implicit synchronizations	Ghost point scatters
128	5	4	3
512	3	7	5
3072	5	14	10

[a] The preconditioner used in these results is block Jacobi with ILU(1) in each subdomain. We observe that the principal nonscaling factor is the implicit synchronization.

processor count are scaled together, we would expect scatter times to occupy a fixed percentage of the total and load imbalance to be reduced at high granularity.

3.2. Effect of partitioning strategy

Mesh partitioning has a dominant effect on parallel scalability for problems characterized by (almost) constant work per point. As shown above, poor load balance causes idleness at synchronization points, which are frequent in implicit methods (e.g., at every conjugation step in a Krylov solver). With NKS methods, then, it is natural to strive for a very well balanced load. The p-MeTiS algorithm in the MeTiS package [16], for example, provides almost perfect balancing of the number of mesh points per processor. However, balancing work alone is not sufficient. Communication must be balanced as well, and these objectives are not entirely compatible. Fig. 1 shows the effect of data partitioning using p-MeTiS, which tries to balance the number of nodes and edges on each partition, and k-MeTiS, which tries to reduce the number of noncontiguous subdomains and connectivity of the subdomains. Better overall scalability is observed with k-MeTiS, despite the better load balance for the p-MeTiS partitions. This is due to the slightly poorer numerical convergence rate of the iterative NKS algorithm with the p-MeTiS partitions. The poorer convergence rate can be explained by the fact that the p-MeTiS partitioner generates disconnected pieces within a single “subdomain,” effectively increasing the number of blocks in the block Jacobi or additive Schwarz algorithm and increasing the size of the interface. The convergence rates for block iterative methods degrade with increasing number of blocks, as discussed in Section 5.

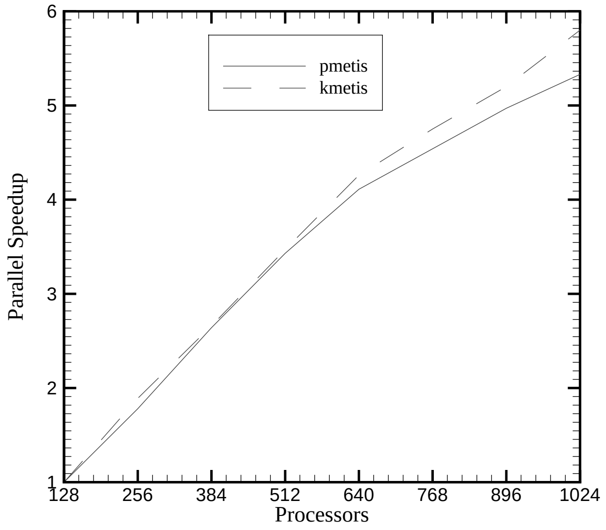

Fig. 1. Parallel speedup relative to 128 processors on a 600 MHz Cray T3E for a 2.8 M-vertex case, showing the effect of partitioning algorithms *k*-MeTiS, and *p*-MeTiS.

3.3. Domain-based and/or instruction-level parallelism

The performance results above are based on subdomain parallelism using the message passing interface (MPI) [13]. With the availability of large scale SMP clusters, different software models for parallel programming require a fresh assessment. For machines with physically distributed memory, MPI has been a natural and successful software model. For machines with distributed shared memory and nonuniform memory access, both MPI and OpenMP have been used with respectable parallel scalability. For clusters with two or more SMPs on a single node, the mixed software model of threads within a node (OpenMP being a special case of threads because of the potential for highly efficient handling of the threads and memory by the compiler) and MPI between the nodes appears natural. Several researchers (e.g., [4,20]) have used this mixed model with reasonable success.

We investigate the mixed model by employing OpenMP only in the flux calculation phase. This phase takes over 60% of the execution time on ASCI Red and is an ideal candidate for shared-memory parallelism because it does not suffer from the memory bandwidth bottleneck (see Section 4). In Table 2, we compare the performance of this phase when the work is divided by using two OpenMP threads per node with the performance when the work is divided using two independent MPI processes per node. There is no communication in this phase. Both processors work with the same amount of memory available on a node; in the OpenMP case, it is shared between the two threads, while in the case of MPI it is divided into two address spaces.

The hybrid MPI/OpenMP programming model appears to be a more efficient way to employ shared memory than are the heavyweight subdomain-based processes (MPI alone), especially when the number of nodes is large. The MPI model works with larger number of subdomains (equal to the number of MPI processors), resulting in slower rate of convergence. The hybrid model works with fewer chunkier subdomains (equal to the number of nodes) that result in faster convergence rate and shorter execution time, despite the fact that there is some redundant work when the data from the two threads are combined due to the lack of a vector-reduce operation in the OpenMP standard (version 1) itself. Specifically, some redundant work arrays must be allocated that are not present in the MPI code. The subsequent gather

Table 2
Execution time on the 333 MHz Pentium Pro ASCI Red machine for function evaluations only for a 2.8 M-vertex case, showing differences in exploiting the second processor sharing the same memory with either OpenMP instruction-level parallelism (number of subdomains equals the number of nodes) or MPI domain-level parallelism (number of subdomains is equal to the number of processes per node)

Nodes	MPI/OpenMP threads per node (s)		MPI processes per node (s)	
	1	2	1	2
256	483	261	456	258
2560	76	39	72	45
3072	66	33	62	40

operations (which tend to be memory bandwidth bound) can easily offset the advantages accruing from the low-latency shared-memory communication. One way to get around this problem is to use coloring strategies to create the disjoint work sets, but this takes away the ease and simplicity of the parallelization step promised by the OpenMP model.

4. Single-processor performance modeling and tuning

In this section, we describe the details of per processor performance and tuning. Since the gap between memory and CPU speeds is ever widening [14] and algorithmically optimized PDE codes do relatively little work per data item, it is crucial to efficiently utilize the data brought into the levels of memory hierarchy that are close to the CPU. To achieve this goal, the data structure storage patterns for primary (e.g., momenta and pressure) and auxiliary (e.g., geometry and constitutive parameter) fields should adapt to hierarchical memory. Three simple techniques have proved very useful in improving the performance of the FUN3D code, which was originally tuned for vector machines. These techniques are interlacing, blocking, and edge reordering. They are within the scope of automated compiler transformations in structured grid codes but, so far must be implemented manually in unstructured codes.

4.1. Interlacing, blocking, and edge reordering

Table 3 shows the effectiveness of interlacing, block, and edge reordering (described below) on one processor of the SGI Origin2000. The combination of the three effects can enhance overall execution time by a factor of 5.7. To further understand the dramatic effect of reordering the edges, we carried out hardware counter profiling on the R10000 processor. Fig. 2 shows that edge reordering reduces the

Table 3
Execution times for Euler flow over M6 wing for a fixed-size grid of 22,677 vertices (90,708 DOFs incompressible; 113,385 DOFs compressible)[a]

Enhancements			Results			
Field interlacing	Structural blocking	Edge reordering	Incompressible		Compressible	
			Time/step (s)	Ratio	Time/step (s)	Ratio
			83.6	–	140.0	–
×			36.1	2.31	57.5	2.44
×	×		29.0	2.88	43.1	3.25
		×	29.2	2.86	59.1	2.37
×		×	23.4	3.57	35.7	3.92
×	×	×	16.9	4.96	24.5	5.71

[a] The processor is a 250 MHz MIPS R10000. Activation of a layout enhancement is indicated by "×" in the corresponding column.

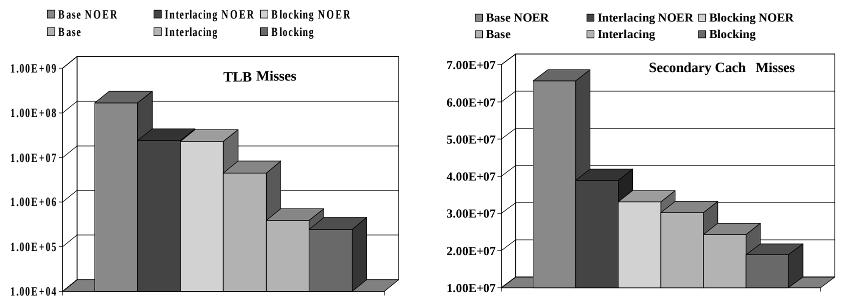

Fig. 2. TLB misses (log scale) and secondary cache misses (linear scale) for a 22,677-vertex case on a 250 MHz R10000 processor, showing dramatic improvements in data locality due to data ordering and blocking techniques. ("NOER" denotes no edge ordering; otherwise edges are reordered by default.)

Table 4
Comparison of optimized to original performance, absolute and as percentage of peak, for PETSc-FUN3D on many processor families

Processor	Clock	Peak	Opt. MF/s	Orig. MF/s	Opt.% peak	Orig.% peak	Ratio
R10000	250	500	127	26	25.4	5.2	4.9
RS6000/ P3	200	800	163	32	20.3	4.0	5.1
RS6000/ P2	120	480	117	15	24.3	3.1	7.8
RS6000/ 604e	333	666	66	15	9.9	2.3	4.4
Pentium Pro	333	333	60	21	18.8	6.3	3.0
Alpha 21164	600	1200	91	16	7.6	1.3	5.7
Ultra II	400	800	71	20	8.9	2.5	3.6

misses in the translation lookaside buffer (TLB) cache by two orders of magnitude, while secondary cache misses (which are very expensive) are reduced by a factor of 3.5. (The TLB cache is used in virtual memory address translation.)

Table 4 compares the original and optimized per processor performance for several other architectures. The ratio of improvement in the last column varies from 2.6 to 7.8. Improvement ratios are averages over the entire code; different subroutines benefit to different degrees.

4.1.1. Field interlacing

Field interlacing creates the spatial locality for the data items needed successively in time. This is achieved by choosing

$$u1, v1, w1, p1, u2, v2, w2, p2, \ldots$$

in place of

$$u1, u2, \ldots, v1, v2, \ldots, w1, w2, \ldots, p1, p2, \ldots$$

for a calculation that uses u, v, w, p together. We denote the first ordering "interlaced" and the second "noninterlaced." The noninterlaced storage pattern is good for vector machines. For cache-based architectures, the interlaced storage pattern has many advantages: (1) it provides high reuse of data brought into the cache, (2) it makes the memory references closely spaced, which in turn reduces the TLB misses, and (3) it decreases the size of the working set of the data cache(s), which reduces the number of conflict misses.

4.1.2. Structural blocking

Once the field data are interlaced, it is natural to use a block storage format for the Jacobian matrix of a multicomponent system of PDEs. The block size is the number of components (unknowns) per mesh point. As shown for the sparse matrix–vector case in [10], this structural blocking significantly reduces the number of integer loads and enhances the reuse of the data items in registers. It also reduces the memory bandwidth required for optimal performance.

4.1.3. Edge and node reorderings

In the original FUN3D code, the edges are colored for good vector performance. No pair of nodes in the same discretization stencil share a color. This strategy results in a very low cache line reuse. In addition, since consecutive memory references may be far apart, the TLB misses are a grave concern. About 70% of the execution time in the original vector code is spent serving TLB misses. As shown in Fig. 2, this problem is effectively addressed by reordering the edges.

The edge reordering we have used sorts the edges in increasing order by the node number at the one end of each edge. In effect, this converts an edge-based loop into a vertex-based loop that reuses vertex-based data items in most or all of the stencils that reference them several times before discarding it. Since a loop over edges goes over a node's neighbors first, edge reordering (in conjunction with a bandwidth reducing ordering for nodes) results in memory references that are closely spaced. Hence, the number of TLB misses is reduced significantly. For vertex ordering, we have used the Reverse Cuthill McKee (RCM) [7], which is known in the linear algebra literature to reduce cache misses by enhancing spatial locality.

4.2. Performance analysis of the sparse matrix–vector product

The sparse matrix–vector product (or "matvec") is an important part of many iterative solvers in its own right, and also representative of the data access patterns of explicit grid-based stencil operations and recurrences. While detailed performance modeling of this operation can be complex, particularly when data reference patterns are included [26,27,29], a simplified analysis can still yield upper bounds on the achievable performance of this operation.

In [10], we estimate the memory bandwidth required by sparse matvecs in unstructured grid codes, after making some simplifying assumptions that idealize the rest of the memory system. We assume that there are no conflict misses, meaning that each matrix and vector element is loaded into cache only once until flushed by capacity misses. We also assume that the processor never waits on a memory reference; that is, that any number of loads and stores are satisfied in a single cycle.

The matrix is stored in compressed rows (equivalent to PETSc's AIJ format) or block AIJ (BAIJ format) [3]. For each nonzero in the matrix, we transfer one integer (giving the column incidence) and two doubles (the matrix element and the corresponding row vector element), and we do one floating-point multiply-add (fmadd) operation (which is two flops). Finally, we store the output vector element. Including loop control and addressing overheads, this leads (see [10]) to a data volume estimate of 12.36 bytes per fmadd operation for a sample PETSc-FUN3D sparse Jacobian. This gives us an estimate of the bandwidth required in order for the processor to do all $2 * n_{\mathrm{nz}}$ flops at its peak speed, where n_{nz} is the number of nonzeros in the Jacobian. Unfortunately, bandwidth as measured by the STREAM [21] benchmark is typically an order of magnitude less. Alternatively, given a measured memory bandwidth rating, we can predict the maximum achievable rate of floating-point operations. Finally, we can measure the achieved floating-point operations. The last four columns of Table 5 summarize the results of this combined theoretical/experimental study for a matrix with 90,708 rows and 5,047,120 nonzero entries from a PETSc-FUN3D application (incompressible) with four unknowns per vertex. For this matrix, with a block size of four, the column incidence array is smaller by a factor of the block size. We observe that the blocking helps significantly by reducing the memory bandwidth requirement. In [10], we also describe how multiplying more than one vector at a time requires less memory bandwidth per matvec because of reuse of matrix elements. We can multiply four vectors in about 1.5 times the time needed to multiply a single vector. If the three additional vectors can be employed in a block Krylov method, they are almost free, so algorithmic work on block-Krylov methods is highly recommended.

To further incriminate memory bandwidth as the bottleneck to the execution time of sparse linear solvers, we have performed an experiment that effectively doubles the available memory bandwidth. The linear solver execution time is dominated by the

Table 5
Effect of memory bandwidth on the performance of sparse matrix–vector products on a 250 MHz R10000 processor[a]

Format	Bytes/fmadd	Bandwidth (MB/s)		Mflop/s	
		Required	Achieved	Ideal	Achieved
AIJ	12.36	3090	276	58	45
BAIJ	9.31	2327	280	84	55

[a] The STREAM benchmark memory bandwidth [21] is 358 MB/s; this value of memory bandwidth is used to calculate the ideal Mflop/s. The achieved values of memory bandwidth and Mflop/s are measured using hardware counters.

cost of preconditioning when (as in our production PETSc-FUN3D code) the Jacobian-vector products required in the Krylov methods are performed in a matrix-free manner by finite-differencing a pair of flux evaluations. Since the preconditioning is already very approximate, we have implemented the data structures storing PETSc's preconditioners in single precision while preserving double-precision in all other parts of the code. Once an element of the preconditioner is in the CPU, it is padded to 64 bits with trailing zeros, and all arithmetic is done with this (consistent but inaccurate) double precision value. The consistency is required to suppress the contamination of the Krylov space with roundoff errors. The loss of accuracy in the preconditioner is irrelevant to the final result, which satisfies the true linearized Newton correction equation to required precision, and it is nearly irrelevant to the convergence rate of the preconditioned iteration. However, it is very relevant to the execution time, as shown in Table 6. Asymptotically, as the preconditioner matrix becomes the dominant noncacheable object in the workingset, the running time of the linear solution is halved, as evidenced by a comparison of columns 2 and 3 in Table 6.

The importance of memory bandwidth to the overall performance is suggested by the single-processor performance of PETSc-FUN3D shown in Fig. 3. The performance of PETSc-FUN3D is compared with the peak performance and the result of the STREAM benchmark [21], which measures achievable performance for memory bandwidth limited computations. These comparisons show that the STREAM results are much better indicators of realized performance than the peak numbers. The parts of the code that are memory bandwidth-limited (like the sparse triangular preconditioner solution phase, which is responsible for about 25% of the overall execution time) are bound to show poor performance, as compared with dense matrix–matrix operations, which often achieve 80–90% of peak.

The importance of reducing the memory bandwidth requirements of algorithms is emphasized by reference to the hardware profiles of the ASCI machines, which are scheduled to reach a peak of 100 Tflop/s by 2004. Table 7 shows the peak processing and memory bandwidth capacities of the first four of these machines. The "white" machine is being delivered to the US Department of Energy at the time of this writing. The "blue" and "red" machines rank in the top three spots of the Top 500 installed computers as of June 2000 [9]. The last column shows that memory

Table 6
Execution times on a 250 MHz R10000 processor for the linear algebra phase of a 357,900-vertex case with single- or double-precision storage of the preconditioner matrix

Number of processors	Computational phase			
	Linear solve (s)		Overall (s)	
	Double	Single	Double	Single
16	223	136	746	657
64	60	34	205	181
120	31	16	122	106

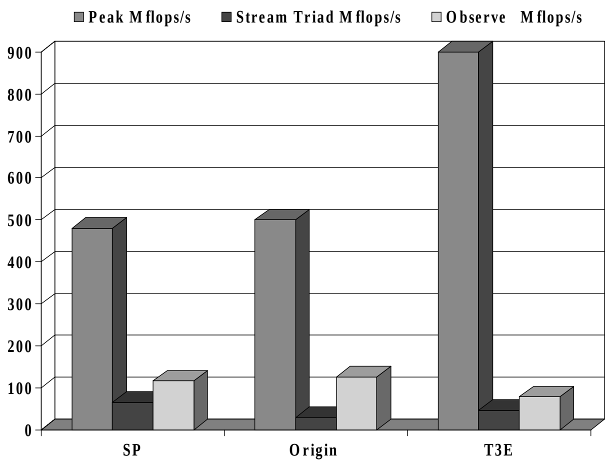

Fig. 3. Sequential performance of PETSc-FUN3D for a 22,677-vertex case.

Table 7
Peak processing and memory bandwidth profiles of the ASCI machines

Platform	Number procs.	Sys. peak (TF/s)	Proc. peak (MF/s)	BW/proc. (MB/s)	BW/proc. (MW/s)	(MF/s)/(MW/s)
White	8192	12.3	1500	1000	125.0	12.0
BlueMtn	6144	3.1	500	390	48.8	10.2
BluePac	5808	3.9	666	360	45.0	14.8
Red	9632	3.2	333	266	33.3	10.0

bandwidth, in double precision words per second, is off by an order of magnitude from what is required if each cached word is used only once. As the raw speed of the machines is increased, this ratio does not improve. Therefore, algorithms must improve to emphasize locality. Several proposals for discretization and solution methods that improve spatial or temporary locality are made in [19]. Many of these require special features in memory control hardware and software that exist today but are not commonly exploited by computational modelers in high-level scientific languages.

4.3. Performance analysis of the flux calculation

Even parts of the code that are not memory intensive often achieve much less than peak performance because of the limits on the number of basic operations that can be performed in a single clock cycle [10]. This is true for the flux calculation routine in PETSc-FUN3D, which consumes approximately 60% of the overall execution time.

While looping over each edge, the flow variables from the vertex-based arrays are read, many floating-point operations are done, and residual values at each node of the edge are updated. Because of the large number of floating-point operations in this phase, memory bandwidth is not (yet) a limiting factor on machines at the high end. Measurements on our Origin2000 support this; only 57 MB/s are needed to keep the flux calculation phase at full throttle [10]. However, the measured floating-point performance is still just 209 Mflop/s out of a theoretical peak of 500 Mflop/s. This is substantially less than the performance that can be achieved with dense matrix–matrix operations.

To understand where the limit on the performance of this part of the code comes from, we take a close look at the assembly code for the flux calculation function. This examination yields the following workload mix for the average iteration of the loop over edges: 519 total instructions, 111 integer operations, 250 floating-point instructions of which there are 55 are `fmadd` instructions (for $195 + 2 \times 55 = 305$ flops), and 155 memory references. Most contemporary processors can issue only one load or store in one cycle. Since the number of floating-point instructions is less than the number of memory references, the code is bound to take at least as many cycles as the number of loads and stores.

If all operations could be scheduled optimally for this hardware – say, one floating-point instruction, one integer instruction, and one memory reference per cycle – this code would take 250 instructions and achieve 305 Mflop/s. However, dependencies between these instructions, as well as complexities in scheduling the instructions [22,24], make it very difficult for the programmer to determine the number of cycles that this code would take to execute. Fortunately, many compilers provide this information as comments in the assembly code. For example, on the Origin2000, when the code is compiled with cache optimizations turned off (consistent with our assumption that data items are in primary cache for the purpose of estimating this bound), the compiler estimates that the above work can be completed in about 325 cycles. This leads to a theoretical performance bound of 235 Mflop/s (47% of the peak on the 250 MHz dual-issue processor). We actually measure 209 Mflop/s using hardware counters. This shows that the performance in this phase of the computation is restricted by the instruction scheduling limitation. A detailed analytical model for this phase of computation is under way.

4.4. Performance comparison

In Fig. 4, we compare three performance bounds: the peak performance (based on the clock frequency and the maximum number of floating-point operations per cycle), the performance predicted from the memory bandwidth limitation, and the performance based on operation issue limitation. For the sparse matrix–vector multiply, it is clear that the memory-bandwidth limit on performance is a good approximation. The greatest differences between the performance observed and predicted by memory bandwidth are on the systems with the smallest caches (IBM SP and T3E), where our assumption that there are no conflict misses is least likely to be valid.

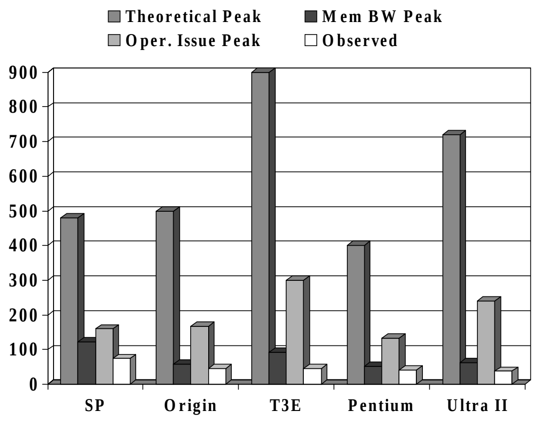

Fig. 4. Three performance bounds for sparse matrix–vector product; the bounds based on memory bandwidth and instruction scheduling are much more closer to the observed performance than the theoretical peak of the processor. Memory bandwidth values are taken from the STREAM benchmark Website.

5. Convergence scalability

The convergence rates and, therefore, the overall parallel efficiencies of additive Schwarz methods are often dependent on subdomain granularity. Except when effective coarse-grid operators and intergrid transfer operators are known, so that optimal multilevel preconditioners can be constructed, the number of iterations to convergence tends to increase with granularity for elliptically controlled problems, for either fixed or memory-scaled problem sizes. In practical large-scale applications, however, the convergence rate degradation of single-level additive Schwarz is sometimes not as serious as the scalar, linear elliptic theory would suggest. Its effects are mitigated by several factors, including pseudo-transient nonlinear continuation and dominant intercomponent coupling. The former parabolizes the operator, endowing diagonal dominance. The latter renders the off-diagonal coupling less critical and, therefore, less painful to sever by domain decomposition. The block diagonal coupling can be captured fully in a point-block ILU preconditioner.

5.1. Convergence of Schwarz methods

For a general exposition of Schwarz methods for linear problems, see [25]. Assume a d-dimensional isotropic problem. Consider a unit aspect ratio domain with quasi-uniform mesh parameter h and quasi-uniform subdomain diameter H. Then problem size $N = h^{-d}$, and, under the one-subdomain-per-processor assumption, processor number $P = H^{-d}$. Consider four preconditioners: point Jacobi, subdomain Jacobi, 1-level additive Schwarz (subdomain Jacobi with overlapped subdomains), and 2-level additive Schwarz (overlapped subdomains with a global coarse problem with approximately one degree-of-freedom per subdomain). The first two can be thought of as degenerate Schwarz methods (with zero overlap, and possibly point-sized subdomains). Consider acceleration of the Schwarz method by a Krylov method such as conjugate gradients or one of its many generalizations to nonsymmetric problems (e.g., GMRES). Krylov–Schwarz iterative methods typically converge in a number of iterations that scales as the square-root of the condition number of the Schwarz-preconditioned system. Table 8 lists the expected number of iterations to achieve a given reduction ratio in the residual norm. The first line of this table pertains to diagonally scaled CG, a common default parallel implicit method, but one that is *not* very algorithmically scalable, since the iteration count degrades with a power of N. The results in this table were first derived for symmetric definite operators with exact solves on each subdomain, but they have been extended to operators with nonsymmetric and indefinite components and inexact solves on each subdomain.

The intuition behind this table is the following: errors propagate from the interior to the boundary in steps that are proportional to the largest implicit aggregate in the preconditioner, whether pointwise (in N) or subdomainwise (in P). The use of overlap avoids the introduction of high-energy-norm solution near discontinuities at subdomain boundaries. The 2-level method projects out low-wave number errors at the price of solving a global problem.

Only the 2-level method scales perfectly in convergence rate (constant, independent of N and P), like a traditional multilevel iterative method. However, the 2-level method shares with multilevel methods a nonscalable cost-per-iteration from the necessity of solving a coarse-grid system of size $\mathrm{O}(P)$. Unlike recursive multilevel methods, a 2-level Schwarz method may have a rather fine coarse grid, for example, $H = \mathrm{O}(h^{1/2})$, which makes it less scalable overall. Parallelizing the coarse grid solve is necessary. Neither extreme of a fully distributed or a fully redundant coarse solve is optimal, but rather something in between.

5.2. Algorithmic tuning for ΨNKS solver

The following is an incomplete list of parameters that need to be tuned in various phases of a pseudo-transient Newton–Krylov–Schwarz algorithm.

- Nonlinear robustness continuation parameters: discretization order, initial timestep, exponent of timestep evolution law.

Table 8
Iteration count scaling of Schwarz-preconditioned Krylov methods, translated from the theory into problem size N and processor number P, assuming quasi-uniform grid, quasi-unit aspect ratio grid and decomposition, and quasi-isotropic operator

Preconditioning	Iteration count	
	In 2D	In 3D
Point Jacobi	$\mathrm{O}(N^{1/2})$	$\mathrm{O}(N^{1/3})$
Subdomain Jacobi	$\mathrm{O}((NP)^{1/4})$	$\mathrm{O}((NP)^{1/6})$
1-level Additive Schwarz	$\mathrm{O}(P^{1/2})$	$\mathrm{O}(P^{1/3})$
2-level Additive Schwarz	$\mathrm{O}(1)$	$\mathrm{O}(1)$

- Newton parameters: convergence tolerance on each timestep, globalization strategy (line search or trust region parameters), refresh frequency for Jacobian preconditioner.
- Krylov parameters: convergence tolerance for each Newton correction, restart dimension of Krylov subspace, overall Krylov iteration limit, orthogonalization mechanism.
- Schwarz parameters: subdomain number, quality of subdomain solver (fill level, number of sweeps), amount of subdomain overlap, coarse grid usage.
- Subproblem parameters: fill level, number of sweeps.

5.2.1. Parameters for pseudo-transient continuation

Although asymptotically superlinear, solution strategies based on Newton's method must often be approached through pseudo-timestepping. For robustness, pseudo-timestepping is often initiated with very small timesteps and accelerated subsequently. However, this conventional approach can lead to long "induction" periods that may be bypassed by a more aggressive strategy, especially for the smooth flow fields.

The timestep is advanced toward infinity by a power-law variation of the switched evolution/relaxation (SER) heuristic of Van Leer and Mulder [23]. To be specific, within each residual reduction phase of computation, we adjust the timestep according to

$$N^{\ell}_{\mathrm{CFL}} = N^{0}_{\mathrm{CFL}} \left(\frac{\|f(u^0)\|}{\|f(u^{\ell-1})\|} \right)^{p},$$

where p is a tunable exponent close to unity. Fig. 5 shows the effect of initial CFL number (the Courant–Friedrich–Levy number, a dimensionless measure of the timestep size), N^0_{CFL}, on the convergence rate. In general, the best choice of initial CFL number is dependent on the grid size and Mach number. A small CFL adds nonlinear stability far from the solution but retards the approach to the domain of superlinear convergence of the steady state. For flows with near discontinuities, it is safer to start with small CFL numbers.

In flows with shocks, high-order (second or higher) discretization for the convection terms should be activated only after the shock position has settled down. We begin such simulations with a first-order upwind scheme and switch to second-order after a certain residual reduction. The exponent (p) in the power law above is damped to 0.75 for robustness when shocks are expected to appear in second-order discretizations. For first-order discretizations, this exponent may be as large as 1.5. A reasonable switchover point of the residual norm between first-order and second-order discretization phases has been determined empirically. In shock-free simulations we use second-order accuracy throughout. Otherwise, we normally reduce the first two to four orders of residual norm with the first-order discretization, then switch to second. This order of accuracy applies to the flux calculation. The preconditioner matrix is always built out of a first-order analytical Jacobian matrix.

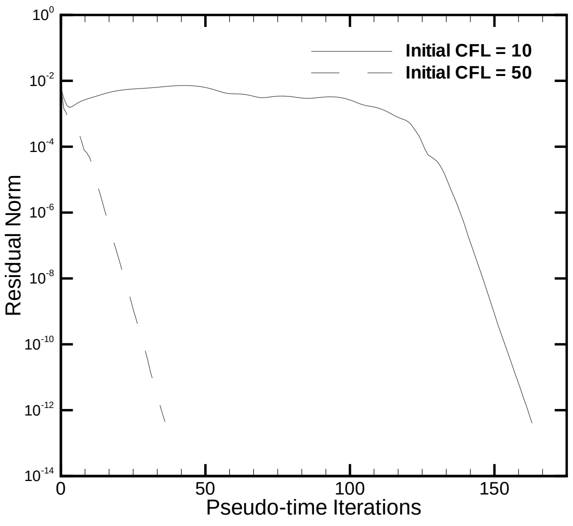

Fig. 5. Residual norm versus iteration count for a 2.8 M-vertex case, showing the effect of initial CFL number on convergence rate. The convergence tuning of nonlinear problems is notoriously case specific.

5.2.2. Parameters for Krylov solver

We use an inexact Newton method on each timestep [8]; that is, the linear system within each Newton iteration is solved only approximately. Especially in the beginning of the solution process, this saves a significant amount of execution time. We have considered the following three parameters in this phase of computation: convergence tolerance, the number of simultaneously storable Krylov vectors, and the total number of Krylov iterations. The typical range of variation for the inner convergence tolerance is 0.001–0.01. We have experimented with progressively tighter tolerances near convergence, and saved Newton iterations thereby, but did not save time relative to cases with loose and constant tolerance. The Krylov subspace dimension depends largely on the problem size and the available memory. We have used values in the range of 10–30 for most of the problems. The total number of linear iterations (within each nonlinear solve) has been varied from 10 for the smallest problem to 80 for the largest one. A typical number of fine-grid flux evaluations for achieving 10^{-10} residual reduction on a million-vertex Euler problem is a couple of thousand.

5.2.3. Additive Schwarz preconditioner

Table 9 explores two quality parameters for the additive Schwarz preconditioner: subdomain overlap and quality of the subdomain solve using incomplete factorization. We exhibit execution time and iteration count data from runs of PETSc-FUN3D on the ASCI Red machine for a fixed-size problem with 357,900 grid points and 1,789,500 degrees-of-freedom. These calculations were performed using GMRES(20), one subdomain per processor (without overlap for block Jacobi and

Table 9
Execution times and linear iteration counts on the 333 MHz Pentium Pro ASCI Red machine for a 357,900-vertex case, showing the effect of subdomain overlap and incomplete factorization fill level in the additive Schwarz preconditioner[a]

Number of processors	*ILU(0) in each subdomain*					
	Overlap					
	0		1		2	
	Time (s)	Linear Its	Time (s)	Linear Its	Time (s)	Linear Its
32	688	930	**661**	816	696	813
64	**371**	993	374	876	418	887
128	**210**	1052	230	988	222	872
	ILU(1) in each subdomain					
32	598	674	**564**	549	617	532
64	**334**	746	335	617	359	551
128	**177**	807	178	630	200	555
	ILU(2) in each subdomain					
32	**688**	527	786	441	–	–
64	**386**	608	441	488	531	448
128	**193**	631	272	540	313	472

[a] The best execution times for each ILU fill level and number of processors are in boldface in each row.

with overlap for ASM), and ILU(k) where k varies from 0 to 2, and with the natural ordering in each subdomain block. The use of ILU(0) with natural ordering on the first-order Jacobian, while applying a second-order operator, allows the factorization to be done in place, with or without overlap. However, the overlap case does require forming an additional data structure on each processor to store matrix elements corresponding to the overlapped regions.

From Table 9 we see that the larger overlap and more fill help in reducing the total number of linear iterations as the number of processors increases, as theory and intuition predict. However, both increases consume more memory, and both result in more work per iteration, ultimately driving up execution times in spite of faster convergence. Best execution times are obtained for any given number of processors for ILU(1), as the number of processors becomes large (subdomain size small), for zero overlap.

The additional computation/communication costs for additive Schwarz (as compared with block Jacobi) are the following.

1. Calculation of the matrix couplings among processors. For block Jacobi, these need not be calculated.
2. Communication of the "overlapped" matrix elements to the relevant processors.
3. Factorization of the larger local submatrices.
4. Communication of the ghost points in the application of the ASM preconditioner. We use restricted additive Schwarz method (RASM) [6], which communicates only when setting up the overlapped subdomain problems and ignores the updates

coming from the overlapped regions. This saves a factor of two in local communication relative to standard ASM.
5. Inversion of larger triangular factors in each iteration.

The execution times reported in Table 9 are highly dependent on the machine used, since each of the additional computation/communication costs listed above may shift the computation past a knee in the performance curve for memory bandwidth, communication network, and so on.

5.2.4. Other algorithmic tuning parameters

In [11] we highlight some additional tunings that have yielded good results in our context. Some subsets of these parameters are not orthogonal but interact strongly with each other. In addition, optimal values of some of these parameters depend on the grid resolution. We are currently using derivative-free asynchronous parallel direct search algorithms [15] to more systematically explore this large parameter space.

We emphasize that the discussion in this section does not pertain to discretization parameters, which constitute another area of investigation – one that ultimately impacts performance at a higher level. The algorithmic parameters discussed in this section do not affect the accuracy of the discrete solution, but only the rate at which the solution is attained. In all of our experiments, the goal has been to minimize the overall execution time, not to maximize the floating-point operations per second. There are many tradeoffs that enhance Mflop/s rates but retard execution completion.

6. Large-scale demonstration runs

We use PETSc's profiling and logging features to measure the parallel performance. PETSc logs many different types of events and provides valuable information about time spent, communications, load balance, and so forth for each logged event. PETSc uses manual counting of flops, which are afterwards aggregated over all the processors for parallel performance statistics. We have observed that the flops reported by PETSc are close to (within 10% of) the values statistically measured by hardware counters on the R10000 processor.

PETSc uses the best timers available at the user level in each processing environment. In our rate computations, we exclude the initialization time devoted to I/O and data partitioning. To suppress timing variations caused by paging in the executable from disk, we preload the code into memory with one nonlinear iteration, then flush, reload the initial iterate, and begin performance measurements.

Since we are solving large fixed-size problems on distributed-memory machines, it is not reasonable to base parallel scalability on a uniprocessor run, which would thrash the paging system. Our base processor number is such that the problem has just fit into the local memory.

The same fixed-size problem is run on large ASCI Red configurations with sample scaling results shown in Fig. 6. The implementation efficiency is 91% in going from

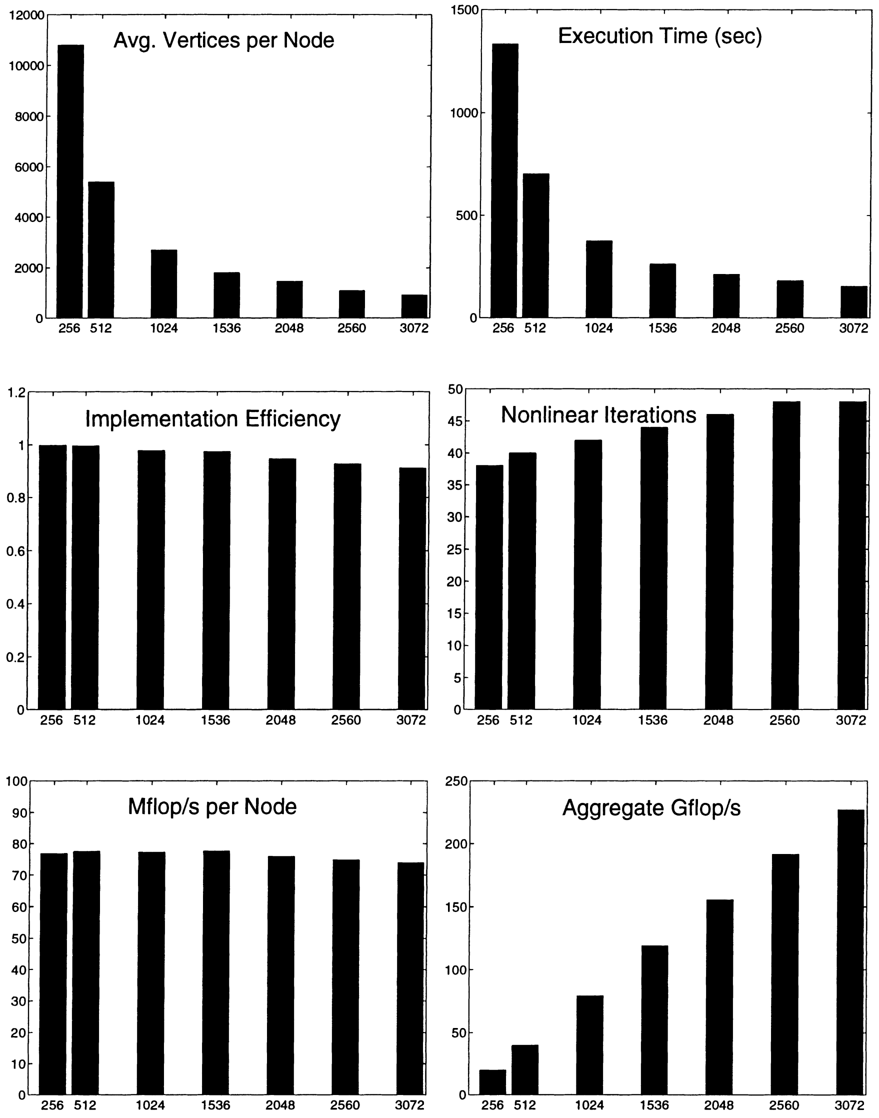

Fig. 6. Parallel performance for a fixed size mesh of 2.8 M vertices run on up to 3072 ASCI Red 333 MHz Pentium Pro processors.

256 to 3072 nodes. For the data in Fig. 6, we employed the `-procs 2` runtime option on ASCI Red. This option enables 2-processor-per-node multithreading during threadsafe, communication-free portions of the code. We have activated this feature for the floating-point-intensive flux computation subroutine alone. On 3072 nodes, the largest run we have been able to make on the unclassified side of the

machine to date, the resulting Gflop/s rate is 227 (when the preconditioner is stored in double precision). Undoubtedly, further improvements to the algebraic solver portion of the code are also possible through multithreading, but the additional coding work does not seem justified at present.

Fig. 7 shows aggregate flop/s performance and a log–log plot showing execution time for our largest case on the three most capable machines to which we have thus far had access. In both plots of this figure, the dashed lines indicate ideal behavior. Note that although the ASCI Red flop/s rate scales nearly linearly, a higher fraction of the work is redundant at higher parallel granularities, so the execution time does not drop in exact proportion to the increase in flop/s. The number of vertices per processor ranges from about 22,000 to fewer than 1000 over the range shown. We point out that for just 1000 vertices in a three-dimensional domain, about half are on the interface (e.g., 488 interface vertices on a $10 \times 10 \times 10$ cube).

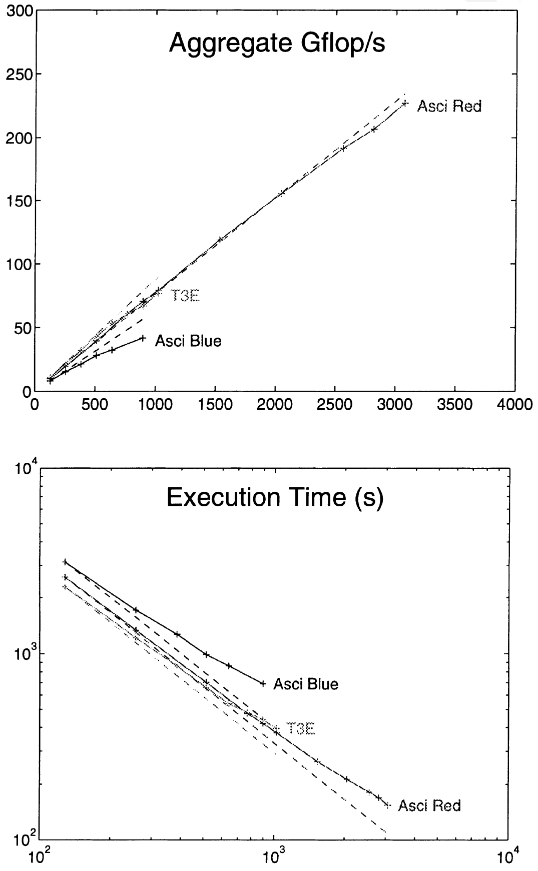

Fig. 7. Gigaflop/s ratings and execution times on ASCI Red (up to 3072 dual processor nodes), ASCI Pacific Blue (up to 768 processors), and a Cray T3E (up to 1024 processors) for a 2.8 M-vertex case, along with dashed lines indicating “perfect” scalings.

7. Conclusions

Large-scale implicit computations have matured to a point of practical use on distributed/shared memory architectures for static-grid problems. More sophisticated algorithms, including solution adaptivity, inherit the same features *within* static-grid phases, of course, but require extensive additional infrastructure for dynamic parallel adaptivity, rebalancing, and maintenance of efficient, consistent distributed data structures.

Unstructured implicit CFD solvers are amenable to scalable implementation, but careful tuning is needed to obtain the best product of per-processor efficiency and parallel efficiency. The number of cache misses and the achievable memory bandwidth are two important parameters that should be considered in determining an optimal data storage pattern. The impact of data reorganizing strategies (interlacing, blocking, and edge/vertex reorderings) is demonstrated through the sparse matrix–vector product model and hardware counter profiling.

Given contemporary high-end architecture, critical research directions for solution algorithms for systems modeled by PDEs are: (1) multivector algorithms and less synchronous algorithms, and (2) hybrid programming models. To influence future architectures while adapting to current ones, we recommend adoption of new benchmarks featuring implicit methods on unstructured grids, such as the application featured herein.

Acknowledgements

We are indebted to Lois C. McInnes and Satish Balay of Argonne National Laboratory, to W. Kyle Anderson, formerly of the NASA Langley Research Center, and to Dimitri Mavriplis of ICASE for collaborations leading up to the work presented here. Debbie Swider of Argonne National Laboratory was of considerable assistance in performing ASCI platform runs. Computer time was supplied by Argonne National Laboratory, Lawrence Livermore National Laboratory, NERSC, Sandia National Laboratories, and SGI-Cray.

References

[1] W.K. Anderson, D.L. Bonhaus, An implicit upwind algorithm for computing turbulent flows on unstructured grids, Comput. Fluids 23 (1994) 1–21.

[2] W.K. Anderson, R.D. Rausch, D.L. Bonhaus, Implicit/multigrid algorithms for incompressible turbulent flows on unstructured grids, J. Comput. Phys. 128 (1996) 391–408.

[3] S. Balay, W.D. Gropp, L.C. McInnes, B.F. Smith, The Portable Extensible Toolkit for Scientific Computing (PETSc) version 28, http://www.mcs.anl.gov/petsc/petsc.html, 2000.

[4] S.W. Bova, C.P. Breshears, C.E. Cuicchi, Z. Demirbilek, H.A. Gabb, Dual-level parallel analysis of harbor wave response using MPI and OpenMP, Int. J. High Performance Comput. Appl. 14 (2000) 49–64.

[5] X.-C. Cai, Some domain decomposition algorithms for nonselfadjoint elliptic and parabolic partial differentia l equations, Technical Report 461, Courant Institute, New York, 1989.
[6] X.-C. Cai, M. Sarkis, A restricted additive Schwarz preconditioner for general sparse linear systems, SIAM J. Scientific Comput. 21 (1999) 792–797.
[7] E. Cuthill, J. McKee, Reducing the bandwidth of sparse symmetric matrices, in: Proceedings of the 24th National Conference of the ACM, 1969.
[8] R.S. Dembo, S.C. Eisenstat, T. Steihaug, Inexact Newton methods, SIAM J. Numer. Anal. 19 (1982) 400–408.
[9] J. Dongarra, H.-W. Meuer, E. Strohmaier, The TOP 500 List, http://www.netlib.org/benchmark/top500.html, 2000.
[10] W.D. Gropp, D.K. Kaushik, D.E. Keyes, B.F. Smith, Toward realistic performance bounds for implicit CFD codes, in: D. Keyes, A. Ecer, J. Periaux, N. Satofuka, P. Fox (Eds.), Proceedings of the Parallel CFD'99, Elsevier, Berlin, 1999, pp. 233–240.
[11] W.D. Gropp, D.K. Kaushik D.E. Keyes, B.F. Smith, Performance modeling and tuning of an unstructured mesh CFD application, in: Proceedings of the SC2000, IEEE Computer Society, 2000.
[12] W.D. Gropp, L.C. McInnes, M.D. Tidriri, D.E. Keyes, Globalized Newton–Krylov–Schwarz algorithms and software for parallel implicit CFD, Int. J. High Performance Comput. Appl. 14 (2000) 102–136.
[13] W. Gropp, E. Lusk, A. Skjellum, Using MPI: Portable Parallel Programming with the Message Passing Interface, second ed., MIT Press, Cambridge, MA, 1999.
[14] J.L. Hennessy, D.A. Patterson, Computer Architecture: A Quantitative Approach, Morgan Kaufmann, Los Altos, CA, 1996.
[15] P.D. Hough, T.G. Kolda, V.J. Torczon, Asynchronous parallel pattern search for nonlinear optimization, Technical Report SAND2000-8213, Sandia National Laboratories, Livermore, January 2000.
[16] G. Karypis, V. Kumar, A fast and high quality scheme for partitioning irregular graphs, SIAM J. Scientific Comput. 20 (1999) 359–392.
[17] C.T. Kelley, D.E. Keyes, Convergence analysis of pseudo-transient continuation, SIAM J. Numer. Anal. 35 (1998) 508–523.
[18] D.E. Keyes, How scalable is domain decomposition in practice? in: C.-H. Lai et al. (Eds.), Proceedings of the 11th International Conference on Domain Decomposition Methods, Domain Decomposition Press, Bergen, 1999.
[19] D.E. Keyes, Four horizons for enhancing the performance of parallel simulations based on partial differential equations, in: Proceedings of the Europar 2000, Lecture Notes in Computer Science, Springer, Berlin, 2000.
[20] D.J. Mavriplis, Parallel unstructured mesh analysis of high-lift configurations, Technical Report 2000-0923, AIAA, 2000.
[21] J.D. McCalpin, STREAM: Sustainable memory bandwidth in high performance computers, Technical report, University of Virginia, 1995, http://www.cs.virginia.edu/stream.
[22] MIPS Technologies, Inc., http://techpubs.sgi.com/library/manuals/ 2000/007-2490-001/pdf/007-2490-001.pdf. MIPS R10000 Microprocessor User's Manual, January 1997.
[23] W. Mulder, B. Van Leer, Experiments with implicit upwind methods for the Euler equations, J. Comput. Phys. 59 (1985) 232–246.
[24] Silicon Graphics, Inc, http://techpubs.sgi.com/library/manuals/ 3000/007-3430-002/pdf/007-3430-002.pdf. Origin 2000 and Onyx2 Performance and Tuning Optimization Guide, 1998, Document Number 007-3430-002.
[25] B.F. Smith, P. Bjørstad, W. Gropp, Domain Decomposition: Parallel Multilevel Algorithms for Elliptic Partial Differential Equations, Cambridge University Press, Cambridge, 1996.
[26] O. Temam, W. Jalby, Characterizing the behavior of sparse algorithms on caches, in: Proceedings of the Supercomputing 92, IEEE Computer Society, 1992, pp. 578–587.
[27] S. Toledo, Improving the memory-system performance of sparse-matrix vector multiplication, IBM J. Res. Dev. 41 (1997) 711–725.

[28] G. Wang, D.K. Tafti, Performance enhancements on microprocessors with hierarchical memory systems for solving large sparse linear systems, Int. J. High Performance Comput. Appl. 13 (1999) 63–79.

[29] J. White, P. Sadayappan, On improving the performance of sparse matrix–vector multiplication, in: Proceedings of the Fourth International Conference on High Performance Computing (HiPC'97), IEEE Computer Society, 1997, pp. 578–587.