Efficient Parallelization of an Unstructured Grid Solver: A Memory-Centric Approach

D. K. Kaushik* D. E. Keyes†

For an unstructured grid computational fluid dynamics computation typical of many large-scale partial differential equations requiring implicit treatment, we describe coding practices that lead to high implementation efficiency for standard computational and communication kernels, in both uniprocessor and parallel senses. Moreover, a family of Newton-like preconditioned Krylov algorithms whose convergence rate degrades only slightly with increasing parallel granularity, relying primarily on sparse Jacobian-vector multiplications, can be expressed in terms of these kernels. A combination of the three (uniprocessor performance, parallel scalability, and algorithmic scalability) is required for overall high performance on the largest scale problems that a given generation of parallel platforms supports.

1. Introduction and Motivation

With teraflops-scale computational modeling expected to be routine by 2003-04, according to the computational "roadmap" of the Accelerated Strategic Computing Initiative (ASCI), there is an increasing need for solvers that are highly scalable in the number of processors. Moreover, since the gap between CPU speed and memory access rate is widening, we need to move towards a memory-centric view of computation in order to obtain reasonable per-processor performance for sparse problems, which include most PDE-based formulations. For that shrinking minority of scientific computations that can be addressed with regular data structures, such as structured sparse or dense problems, automated software tools lead to reasonably good [9] or even excellent [4] performance. Until automated tools like parallel compilers or source-to-source translators can discover enough of the locality (to minimize the vertical memory hierarchy traffic) and concurrency (to minimize the horizontal memory traffic) that is latent in most irregular scientific computations, manual expression of locality and concurrency seems to be the key to achieving high performance. For the purpose of this characterization, we can define a problem as

*Computer Science Department, Old Dominion University, Norfolk, VA 23529-0162 and Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439-4844, kaushik@cs.odu.edu. Supported in part by Wallace Givens Fellowship from Argonne National Laboratory and by a GAANN fellowship from the U. S. Department of Education.

†Department of Mathematics & Statistics, Old Dominion University, Norfolk, VA 23529-0077, Institute for Scientific Computing Research, Lawrence Livermore National Laboratory, Livermore, CA 94551-9899, and ICASE, NASA Langley Research Center, Hampton, VA 23681-2199, keyes@icase.edu. Supported in part by the National Science Foundation under grant ECS-9527169, by NASA under contracts NAS1-19480 and NAS1-97046, and by the U.S. Department of Energy under ANL subcontract 982232402 and LLNL subcontract B347882

irregular when its data access patterns are themselves data-dependent. Fortunately, in the context of PDE-based simulations these irregular patterns are executed many times; for instance, it may be necessary to form the fluxes over an unstructured grid a thousand times in the process of obtaining a highly converged solution. In such cases, the work of partitioning and ordering is easily amortized with good performance as the reward. To illustrate these claims, we present parallelization and performance tuning experiences with a three-dimensional unstructured grid Euler code (compressible and incompressible) from NASA, which we have implemented in the PETSc [3] framework and ported to many machines.

2. Memory-Centric Approach

We view a PDE computation primarily as a mix of loads and stores with embedded floating point operations (FLOPs). Since FLOPs are cheap, we concentrate on minimizing memory references and emphasize strong sequential performance as one of the factors needed for aggregate performance worthy of the theoretical peak of a parallel machine [7]. We use *interlacing* (creating spatial locality for the data items needed successively in time), *structural blocking* for a multicomponent system of PDEs (cutting the number of integer loads significantly, and enhancing reuse of data items in registers), and *vertex and edge reorderings* (increasing the level of temporal and spatial locality in cache).

The basic philosophy of any efficient parallel computation is “owner computes,” with message merging and overlapping communication with computation where possible via split transactions. Each processor “ghosts” its stencil dependences 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. For PDEs, work is performed on the volume and communication is performed on the surface of each subdomain. If a roughly constant number of vertices and edges is maintained on each processor as the number of processors and the size of the PDE grid grow proportionally, the fraction of parallel overhead remains fixed and Amdahl’s Law is defeated [14].

3. Parallel Newton-Krylov-Schwarz Solvers and Software

Our framework for an implicit PDE solution algorithm, with pseudo-timestepping to advance towards an assumed steady state, has the form:

$$\mathbf{F}^{\ell}(\mathbf{u}^{\ell}) \equiv (\frac{1}{\Delta t^{\ell}})[\mathbf{u}^{\ell} - \mathbf{u}^{\ell-1}] + \mathbf{f}(\mathbf{u}^{\ell}) = 0, \tag{1}$$

where $\Delta t^{\ell} \to \infty$ as $\ell \to \infty$, $\mathbf{u}$ represents the fully coupled vector of unknowns, and the steady-state solution satisfies $\mathbf{f}(\mathbf{u}) = 0$.

Each member of the sequence of nonlinear problems, $\mathbf{F}^{\ell}(\mathbf{u}^{\ell}) = 0,\ \ell = 1, 2, \ldots$, is solved with an inexact Newton method. The resulting nonsymmetric linear Jacobian systems for the Newton corrections, namely

$$\frac{\partial \mathbf{F}^{\ell}}{\partial \mathbf{u}}(\mathbf{u}^{\ell})\ \delta\mathbf{u} = -\mathbf{F}^{\ell}(\mathbf{u}^{\ell}), \tag{2}$$

are solved with an appropriate Krylov method, namely GMRES, relying directly only on matrix-free operations. In particular, based on the multivariate Taylor theorem, we replace the Jacobian-vector products required in a Krylov method,

$$\frac{\partial \mathbf{F}^\ell}{\partial \mathbf{u}}(\mathbf{u}^\ell)\ \mathbf{v}$$

for an arbitrary vector $\mathbf{v}$, with

$$\frac{1}{h}[\mathbf{F}^\ell(\mathbf{u}^\ell + h\mathbf{v}) - \mathbf{F}^\ell(\mathbf{u}^\ell)],$$

where h is an appropriately chosen scalar.

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, building an approximate inverse to the global Jacobian out of approximate solutions within each subdomain of a partitioning of the global PDE domain. Schwarz methods are distinguished by the amount of overlap in the partitions governing each subdomain and by the presence or absence of a coarse grid spanning the independent subdomains. A beautiful theory exists [15] quantifying the convergence gains of overlap and the use of multiple levels, which have to be balanced against the cost of these operations on real parallel computer hardware. In our context, the diagonal term in the Jacobian of $\mathbf{F}^\ell$ proportional to $\frac{1}{\Delta t^\ell}$ endows the Jacobian with better conditioning than would be present in a purely elliptic formulation. While our parallel solver has overlap and coarse grid functionality, we do not employ either in the results reported herein. The decision *not* to employ a coarse grid leads to poor conditioning in the final nonlinear iterations, when Δt is built up towards a value which is effectively infinity, for the approach to the steady state [13]. However, most of the linear work occurs for smaller values of Δt.

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. 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) [8]. Combined with pseudo-timestepping, we write "ΨNKS."

We employ the PETSc package [3], which features distributed data structures — index sets, vectors, and matrices — as fundamental objects. Iterative linear and nonlinear solvers, implemented in as data structure-neutral a manner as possible, are combinable modularly, recursively, and extensibly through a uniform application programmer interface. Portability is achieved through MPI, but message-passing detail is not required in user code. We use one of many flavors of MeTiS [10] to partition the unstructured grid.

4. Parallel Port of FUN3D

This section briefly describes the legacy code upon which our port is based, our philosophy of parallelization, a history of the porting process, and illustrative tuning of the pseudo-timestep parameter, compiler flags, and partitioner flavors. Our purpose in the limited available space is not to be comprehensive, but to highlight aspects likely to be of interest to others facing similar objectives.

4.1. Description of FUN3D

The demonstration code, FUN3D, is a tetrahedral vertex-centered unstructured grid code 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 variable-order Roe schemes for approximating the convective fluxes and and a Galerkin discretization for the viscous terms. FUN3D is being used for design optimization of airplanes, automobiles, and submarines, with irregular meshes comprising several million grid points. The optimization loop involves many analysis cycles. Thus, time to reach the steady-state solution in each analysis cycle is crucial to conducting the design optimization in reasonable amount of time. Our effort, from the beginning, has been focused on achieving small time to convergence without compromising scalability, by means of appropriate algorithms and architecturally efficient data structures.

4.2. Methodology of the Parallelization

The parallelization of unstructured mesh codes is complicated by the fact that no two interprocessor data dependency patterns are alike. Further, the user-provided global ordering may be incompatible with the subdomain-contiguous ordering required for high performance and convenient *single program multiple data* (SPMD) coding.

The philosophy of our SPMD implementation is as follows:

- We follow the “owner computes” rule under the dual constraints of minimizing the number of messages and overlapping communication with computation.
- Each processor “ghosts” its stencil dependences on its nearest neighbors, in our case with a one-level halo. Because of the second-order convective scheme, two levels of halo of the primitive variables are needed in some directions, but one two-level halo exchange may be replaced with two one-level halo exchanges — one on the primitive variables followed by one on their gradients. We have experimented with a two-level halo but memory requirements become quite prohibitive at high granularity.
- We enforce a local ordering on the locally-owned nodes; ghost nodes get ordered after contiguous owned nodes. This strategy saves CPU cycles, since it avoids searches while deciding if a node is local or not, and the memory flag that would otherwise be required to distinguish a local or ghost node. Figure 1 shows different orderings that arise as a result of 2-way partitioning for a simple 2D grid.
- Scatter/gather operations are created between local sequential vectors and global distributed vectors, based on runtime connectivity patterns.
- Newton-Krylov-Schwarz matrix-vector and flux evaluation operations are translated into local tasks and communication tasks, nonblocking for overlap where the hardware supports it.

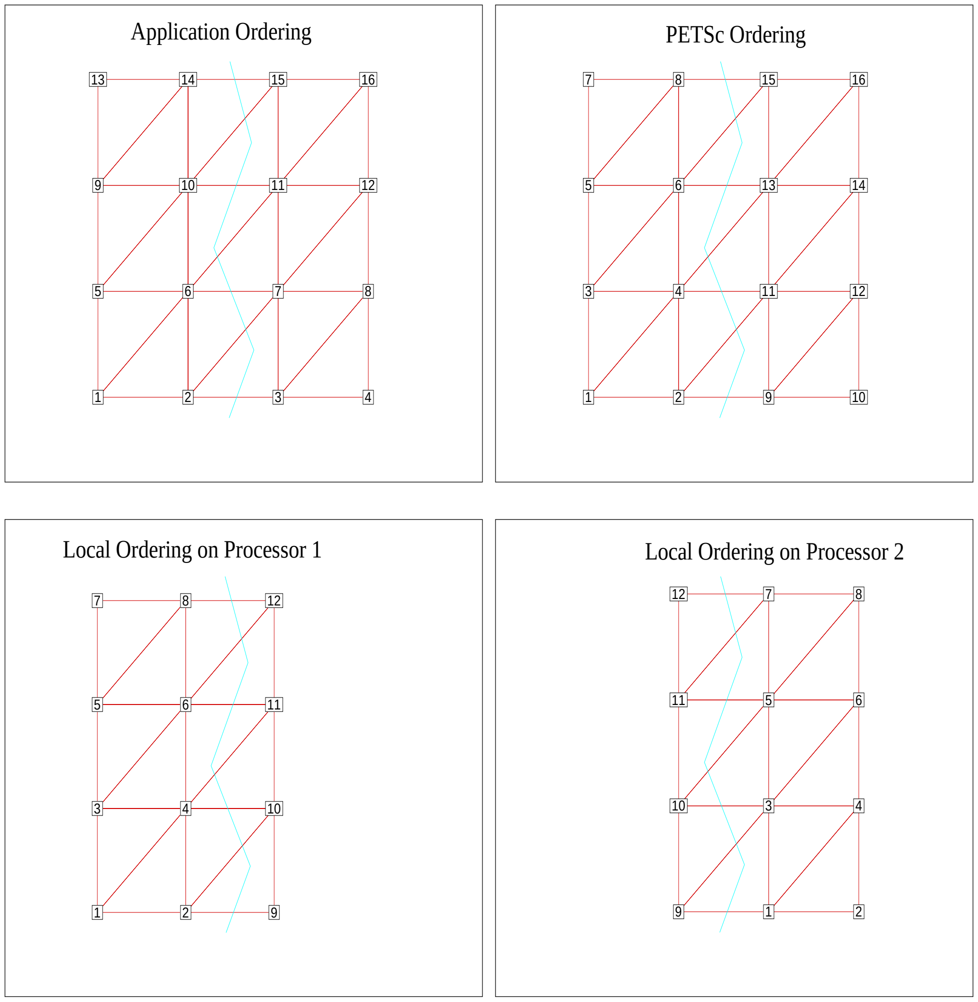

Figure 1. Illustration of three different orderings (user endowed, global library endowed, and local library endowed) for the two-way partitioning of a simple mesh. Note that within each partitioning halo vertices are ordered last and that orderings are contiguous within a partition.

Our parallel experience with FUN3D is with the incompressible/compressible Euler subset thus far, but nothing in the solution algorithms or software changes with additional physical phenomenology. Of course, convergence rate will vary with conditioning, as determined by Mach and Reynolds numbers and the correspondingly induced grid adaptivity. Furthermore, robustness becomes more of an issue in problems admitting shocks or making use of turbulence models. The lack of nonlinear robustness is a fact of life that is largely outside of the domain of parallel scalability. In fact, 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 Euler code, with its smaller number of flops per point per iteration and its aggressive pseudo-transient buildup towards the steady state limit may be a *more*, not less, severe test of scalability.

4.3. History and Software Engineering of the Port

This project was started in October, 1996 and was completed in March, 1997. Since then it has undergone continuous enhancements. The main goal was two-fold: first, to demonstrate the viability of a library-based approach to implicit parallel PDE simulation and second, to create a paradigm for integrating (reusing) existing legacy scientific computing codes with modern (object-oriented) software technology. The library-based approach has several advantages, such as the availability of a large number of parameterizable linear and nonlinear solvers and preconditioners with well optimized distributed data structures and communication. Object-oriented technology promises *encapsulation* to hide the details of implementation from the logical interface, and *extensibility* to easily incorporate the future developments into the code.

The five month (part-time) effort mainly focussed on learning FUN3D (a Fortran77 legacy code), PUNS3D (a mesh preprocessor), and PETSc (a C-based library that is object-oriented at the highest levels). As the parallelization phase proceeded, many new functionalities were added to PETSc to allow an efficient implementation of an unstructured mesh solver (hitherto, PETSc had been oriented towards structured mesh computations). Inasmuch as FUN3D was originally written for vector machines, many data layout transformations [11] had to be carried out to make it efficient on cache based processors. Fortunately, FUN3D had been written without reliance on global `COMMON` arrays. Removal of Fortran's direct memory mapping in the form of `COMMON` arrays is the first (and often most time-consuming) part of the distributed memory parallelization process for most Fortran legacy codes. Approximately 3,300 of 14,400 F77 lines of FUN3D have been retained (primarily as "node code" for flux and Jacobian evaluations); PETSc solver routines replaced the rest. We expect that future unstructured mesh code ports will take significantly less time.

4.4. Performance Tuning of PETSc-FUN3D

A significant amount of effort has been devoted to tuning the performance of the integrated PETSc-FUN3D code. These have occurred on three fronts: algorithmic parameters (of pseudo-transient continuation, nonlinear solver, and linear solver), compiler flags (to invoke the best possible optimizations without going to the assembly level), and data layouts and partitionings (to improve cache performance, and reduce the communication cost and load imbalances). Some representative results are presented to highlight the

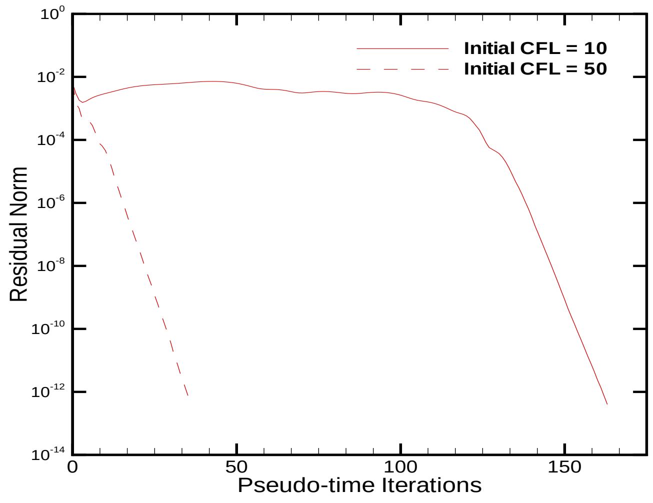

Figure 2. The effect of initial CFL number on convergence rate. The grid has about 2.8 million vertices. The more aggressive approach to the steady state pays off.

importance of the performance tuning we have carried out for PETSc-FUN3D.

Figure 2 shows the effect of initial CFL number on the convergence rate. In general, the best choice of initial CFL number is dependent on the grid size and Mach number. Small CFL adds nonlinear stability far from the solution but retards the approach to the domain of superlinear convergence to the steady state.

The effect of compiler flags is illustrated in Figure 3; the range of performance variation is approximately 40%. This histogram reveals that when we comply with IEEE arithmetic, we get better performance than if we allow the compiler to carry out the transformations that should be faster but less accurate.

Figure 4 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 non-contiguous subdomains and connectivity of the subdomains). k-MeTiS gives better scalability when the number of subdomains is large in our context.

5. Measuring the Parallel Performance

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, etc., 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 ten percent of) the values statistically measured by hardware counters on R10000 processor.

PETSc uses the best timers available in each processing environment. In our rate

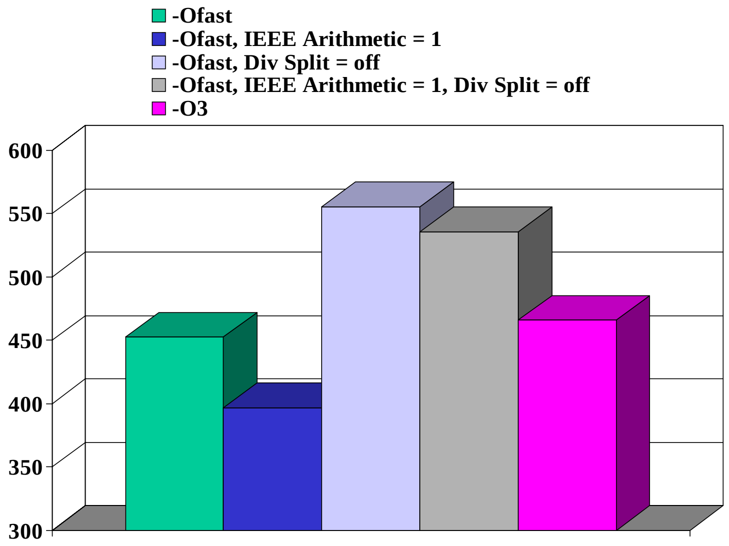

Figure 3. The effect of compiler flags on the execution time (seconds) of a grid with 357,900 vertices run on 32 processors of 250 MHz SGI Origin 2000. Here, "IEEE_Arithmetic = 1" means that the compiler complies with the IEEE arithmetic standard and "Div Split = off" means that the operation $\frac{x}{y}$ is not done as $x \times (\frac{1}{y})$ where reciprocal of y is calculated in some cheap way.

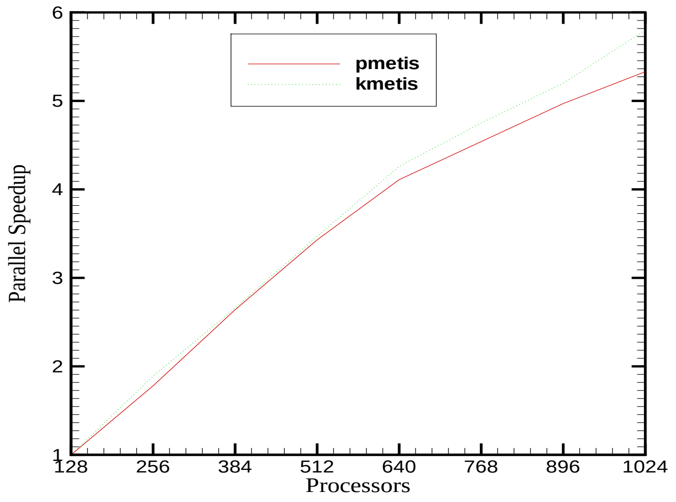

Figure 4. The effect of the two partitioning algorithms, k-MeTiS, and p-MeTiS on parallel scalability of PETSc-FUN3D using up to 1024 600 MHz SGI/Cray T3E processors. The grid has about 2.8 million vertices.

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. We have employed smaller sequential cases to optimize cached data reuse [11,12] to minimize the execution time. In the results below, we decompose the parallel efficiency into two factors: *algorithmic efficiency*, measuring the effect of increased granularity on the number of iterations to convergence, and *implementation efficiency*, measuring the effect of increased granularity on per-iteration performance.

6. Results and Discussion

As emphasized in Section 2, we regard a good uniprocessor implementation as a prerequisite to measuring parallel scalability, since a poor uniprocessor implementation can mask parallel overhead, and return inflated parallel scalability. Therefore, we give examples of both sequential and parallel performance for PETSc-FUN3D in this section. The parallel scalability is based on the best sequential form of the code.

6.1. Sample Sequential Performance

As observed in [11] for the same unstructured flow code, data structure storage patterns for primary and auxiliary fields should adapt to hierarchical memory through: (1) interlacing, (2) blocking of degrees of freedom (DOFs) that are defined at the same point in point-block operations, and (3) reordering of edges for reuse of vertex data. For vertices we have used the Reverse Cuthill McKee (RCM) ordering [6], which is known for reducing cache misses by spatial locality, since a vertex's data is required in the evaluation of its neighbors' stencils.

Table 1 shows the effect of these techniques on one processor the SGI Origin 2000. The combination of the three effects can enhance overall execution time by a factor of 5.7 (a table comparing several architectures is available in [12]). To further understand these results, we carried out hardware counter profiling on R10000 processor. Figure 5 shows that edge reordering reduces the TLB misses by two orders of magnitude while secondary cache misses (which are very expensive) are reduced by a factor of 3.5.

6.2. Sample Parallel Scalability Performance

The parallel scalability of PETSc-FUN3D is shown in Figure 6 for a tetrahedral grid with 2.8 million vertices running on up to 1024 Cray T3E processors. We see that the implementation efficiency of parallelization (i.e., the efficiency on a per-iteration basis) is 82% in going from 128 to 1024 processors. The number of iterations is also fairly flat over the same eight-fold range of processor number (rising from 37 to 42), reflecting reasonable algorithmic scalability. This is much less serious degradation than predicted by the linear elliptic theory, see [15]). The overall efficiency is the product of the implementation efficiency and the algorithmic efficiency. The Mflop/s per processor are also close to flat over this range, even though the relevant working sets in each subdomain vary by nearly

Table 1
Flow over M6 wing; fixed-size grid of 22,677 vertices (90,708 DOFs incompressible; 113,385 DOFs compressible); MIPS R10000, 250MHz, cache: 32KB data / 32KB instr / 4MB L2. Activation of a layout enhancement is indicated by a “×” in the corresponding column. Improvement ratios are averages over the entire code; different subroutines benefit to different degrees.

Enhancements			Results			
			Incompressible		Compressible	
Field Interlacing	Structural Blocking	Edge Reordering	Time/Step	Ratio	Time/Step	Ratio
			83.6s	—	140.0s	—
×			36.1s	2.31	57.5s	2.44
×	×		29.0s	2.88	43.1s	3.25
		×	29.2s	2.86	59.1s	2.37
×		×	23.4s	3.57	35.7s	3.92
×	×	×	16.9s	4.96	24.5s	5.71

a factor of eight.

7. Conclusions and Future Work

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. We [12] and others have solved problems of millions of vertices on hundreds of processors at rates approaching hundreds of gigaflop/s, and we believe such performance is extensible, with further effort, to the teraflop/s regime. In the future, we hope to enhance per-processor performance through improved spatial and temporal locality and the exploitation of “processors in memory” (PIM). We also hope to enhance parallel efficiency through algorithms that synchronize less frequently, and through multiobjective partitioning, which equidistributes communication (surface) work as well as computational (volume) work.

Acknowledgments

The authors thank W. Kyle Anderson of the NASA Langley Research Center for providing FUN3D. Satish Balay, Bill Gropp, Lois McInnes, and Barry Smith of Argonne National Laboratory co-developed the PETSc software employed in this paper. Computer time was supplied by DOE (through Argonne and NERSC) and Silicon Graphics.

REFERENCES

1. W. K. Anderson and D. L. Bonhaus. An implicit upwind algorithm for computing turbulent flows on unstructured grids. *Computers and Fluids*, 23(1):1–21, 1994.
2. W. K. Anderson, R. D. Rausch, and D. L. Bonhaus. Implicit/multigrid algorithms for incompressible turbulent flows on unstructured grids. *Journal of Computational Physics*, 128:391–408, 1996.

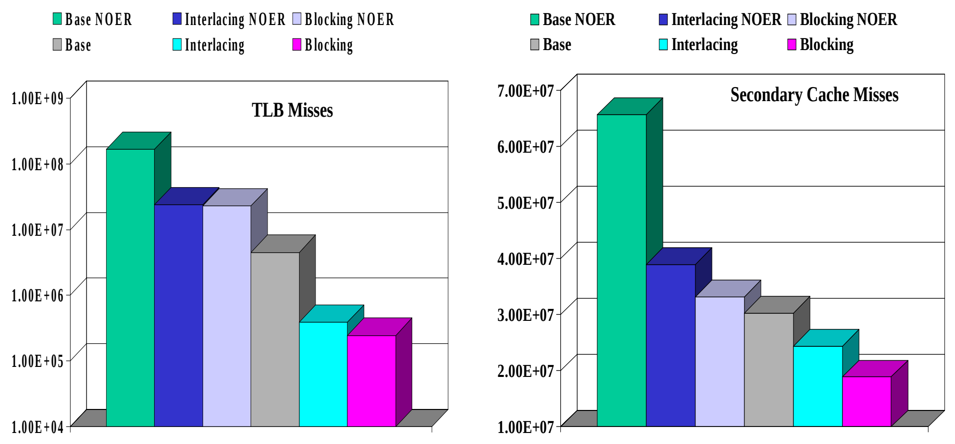

Figure 5. TLB and secondary cache misses corresponding to the data in Table 1. "NOER" denotes no edge ordering, otherwise edges are reordered by default.

3. Satish Balay, William Gropp, Lois Curfman McInnes, and Barry Smith. The Portable, Extensible,Toolkit for Scientific Computing (PETSc) ver. 24. `http://www.mcs.anl.gov/petsc`, 1999.
4. J. Bilmes, K. Asanovic, C.-W. Chin, and Jim Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *Proceedings of the International Conference on Supercomputing*, 1997.
5. X. C. Cai. Some domain decomposition algorithms for nonselfadjoint elliptic and parabolic partial differential equations. Technical Report 461, Courant Institute, NY, New York, 1989.
6. E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *ACM Proceedings of the 24th National Conference*, 1969.
7. W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith. Toward realistic performance bounds for implicit CFD codes. In A. Ecer et al., editor, *Proceedings of Parallel CFD'99*. Elsevier, 1999.
8. W. D. Gropp, L. C. McInnes, M. D. Tidriri, and D. E. Keyes. Parallel implicit PDE computations. In A. Ecer, D. Emerson, J. Periaux, and N. Satofuka, editors, *Proceedings of Parallel CFD'97*, pages 333–344. Elsevier, 1997.
9. M. E. Hayder, C. Ierotheou, and D. E. Keyes. Three parallel programming paradigms: Comparisons on an archetypal PDE computation. Technical report, ICASE, September 1999. To appear.
10. G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal of Scientific Computing*, 20(1):359–392, 1999.
11. D. K. Kaushik, D. E. Keyes, and B. F. Smith. On the interaction of architecture and algorithm in the domain-based parallelization of an unstructured grid incompressible flow code. In J. Mandel *et al.*, editor, *Proceedings of Proceedings of the 10th International Conference on Domain Decomposition Methods*, pages 311–319. Wiley, 1997.
12. D. K. Kaushik, D. E. Keyes, and B. F. Smith. Newton-Krylov-Schwarz methods for aerodynamic problems: Compressible and incompressible flows on unstructured grids. In C.-H. Lai *et al.*, editor, *Proceedings of the 11th International Conference on Domain Decomposition*

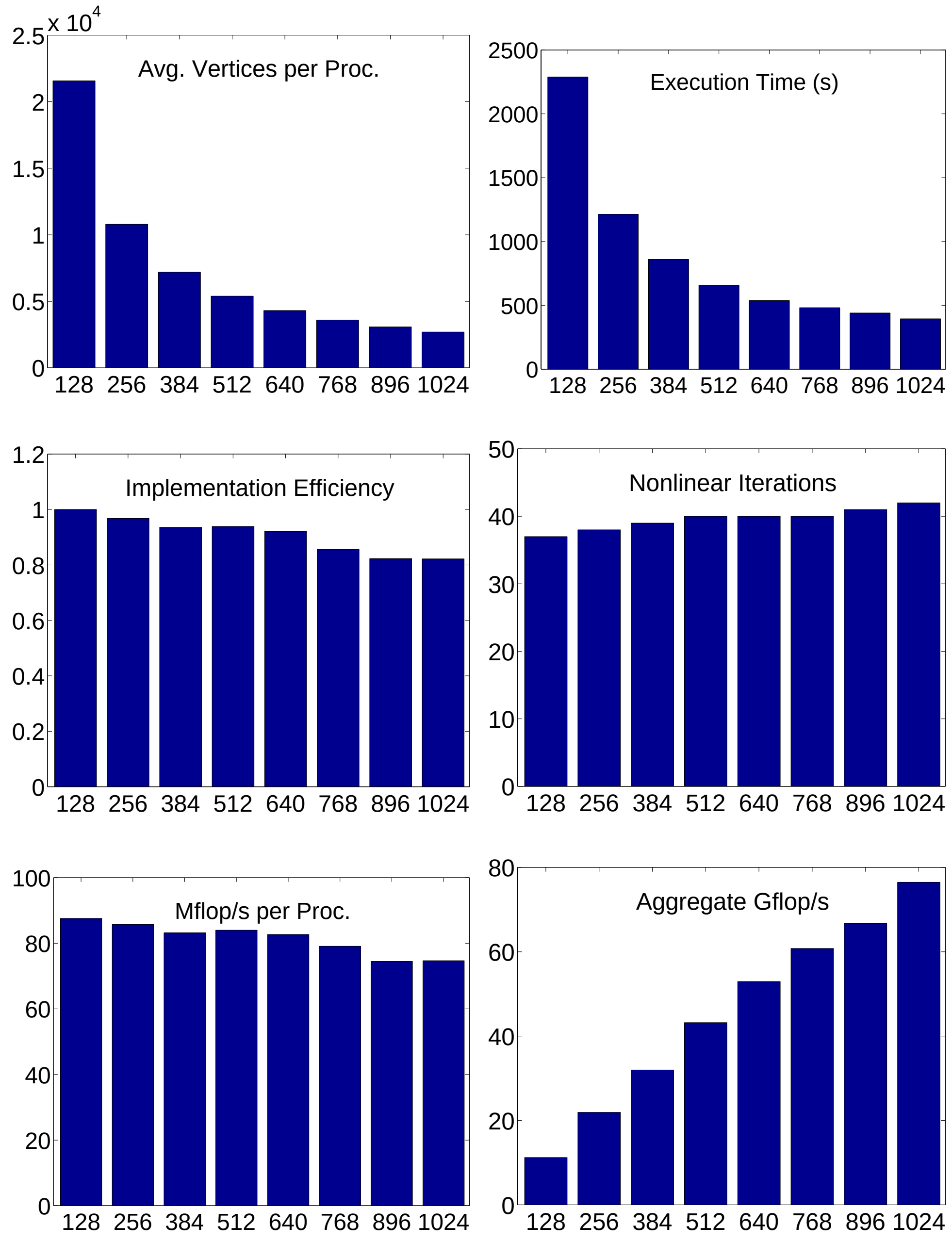

Figure 6. Parallel performance for a fixed size mesh of 2.8 million vertices run on up to 1024 Cray T3E 600 MHz processors

Methods, pages 513–520. Domain Decomposition Press, Bergen, 1999.

13. C. T. Kelley and D. E. Keyes. Convergence analysis of pseudo-transient continuation. *SIAM Journal of Numerical Analysis*, 19:246–265, 1998.
14. D. E. Keyes. How scalable is domain decomposition in practice? In C.-H. Lai, et al., editor, *Proceedings of the 11th International Conference on Domain Decomposition Methods*, pages 286–297. Domain Decomposition Press, Bergen, 1999.
15. B. F. Smith, P. Bjorstad, and W. Gropp. *Domain Decomposition.* Cambridge University Press, 1996.