Winter 2002

Performance Modeling and Prediction for the Scalable Solution of Partial Differential Equations on Unstructured Grids

Dinesh Kumar Kaushik
Old Dominion University

Performance Modeling and Prediction for the Scalable Solution of Partial Differential Equations on Unstructured Grids

by

Dinesh Kumar Kaushik
M.S. May 1991, Indian Institute of Technology, Kanpur, India

A Dissertation Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirement for the Degree of

DOCTOR OF PHILOSOPHY

COMPUTER SCIENCE

OLD DOMINION UNIVERSITY
December 2002

Approved by:

David Keyes (Director)

William Gropp (Member)

Alex Pothen (Member)

Chester Grosch (Member)

Mohammad Zubair (Member)

ABSTRACT

PERFORMANCE MODELING AND PREDICTION FOR THE SCALABLE SOLUTION OF PARTIAL DIFFERENTIAL EQUATIONS ON UNSTRUCTURED GRIDS

Dinesh Kumar Kaushik
Old Dominion University, 2002
Advisor: Dr. David E. Keyes

This dissertation studies the sources of poor performance in scientific computing codes based on partial differential equations (PDEs), which typically perform at a computational rate well below other scientific simulations (e.g., those with dense linear algebra or N-body kernels) on modern architectures with deep memory hierarchies. We identify that the primary factors responsible for this relatively poor performance are: insufficient available memory bandwidth, low ratio of work to data size (good algorithmic efficiency), and nonscaling cost of synchronization and gather/scatter operations (for a fixed problem size scaling). This dissertation also illustrates how to reuse the legacy scientific and engineering software within a library framework.

Specifically, a three-dimensional unstructured grid incompressible Euler code from NASA has been parallelized with the Portable Extensible Toolkit for Scientific Computing (PETSc) library for distributed memory architectures. Using this newly instrumented code (called PETSc-FUN3D) as an example of a typical PDE solver, we demonstrate some strategies that are effective in tolerating the latencies arising from the hierarchical memory system and the network. Even on a single processor from each of the major contemporary architectural families, the PETSc-FUN3D code runs from 2.5 to 7.5 times faster than the legacy code on a medium-sized data set (with approximately 10^5 degrees of freedom). The major source of performance improvement is the increased locality in data reference patterns achieved through blocking, interlacing, and edge reordering.

To explain these performance gains, we provide simple performance models based on memory bandwidth and instruction issue rates.

Experimental evidence, in terms of translation lookaside buffer (TLB) and data cache miss rates, achieved memory bandwidth, and graduated floating point instructions per memory reference, is provided through accurate measurements with hardware counters. The performance models and experimental results motivate algorithmic and software practices that lead to improvements in both parallel scalability and per-node performance. We identify the bottlenecks to scalability (algorithmic as well as implementation) for a fixed-size problem when the number of processors grows to several thousands (the expected level of concurrency on terascale architectures). We also evaluate the hybrid programming model (mixed distributed/shared) from a performance standpoint.

To my parents

ACKNOWLEDGMENTS

I express my gratitude to all my colleagues, friends, and family members, whose continuous support has helped in completing this dissertation.

I express deep appreciation and gratitude to my advisor, Dr. David Keyes, for providing invaluable guidance, encouragement, and support during the entire period of my doctoral study. He has been my favorite teacher, friend, and mentor. He is an epitome of knowledge and success to which I aspire.

I am also grateful to Dr. William Gropp of Argonne National Laboratory and a dissertation committee member. His continuous support and encouragement have enabled me and motivated me to investigate many challenging issues that have become part of this dissertation. I extend my gratitude to Dr. Barry Smith of Argonne National Laboratory for providing technical and moral support for this work from its inception. It was his confidence about extending PETSc to unstructured problems, expressed at an Argonne workshop in 1996, that launched the code collaboration from which this thesis sprang.

I recognize the work of Dr. W. Kyle Anderson, whose state of the art code provided a solid foundation and important application for this research. It was humbling to share in a 1999 Gordon Bell Prize with Drs. Anderson, Gropp, Keyes, and Smith, which was made possible by our strong interdisciplinary collaboration, to which each made an indispensable contribution.

Special thanks go to my other dissertation committee members Dr. Alex Pothen, Dr. Chester Grosch, and Dr. Mohammad Zubair. I appreciate their time and effort in reading my dissertation and providing useful feedback on many technical issues relevant to this dissertation. I am grateful to Dr. Steven Zeil and Dr. Piyush Mehrotra for serving on my candidacy exam committee and dedicating time to familiarize themselves with my work.

This work has benefited significantly from the technical help provided by PETSc team members Mr. Satish Balay, Dr. Lois Curfman McInnes, Dr. Matthew Knepley, and Mr. Kristopher Buschelman.

I want to thank my wife, Urmila, whose unending love and understanding have kept me going. Special mention goes to the help and support provided by my friends Arunkumar Satyanarayana, Mohagna Pandya, Srikanth Pidugu, Satish Boregowda, and Dmitry Karpeev.

I acknowledge the support provided by various U.S. Government Agencies. This research was supported by a **GAANN** fellowship from the Department of Education through Old Dominion University, a **Givens** fellowship from the Department of Energy at Argonne National Laboratory, the Accelerated Strategic Computing Initiative (ASCI) of the Department of Energy through a Level-2 subcontract to Old Dominion University, and a **PACI** grant from National Science Foundation. Computer resources for this work were supplied by Argonne National Laboratory, the Institute for Computer Applications in Science and Engineering (ICASE) at the NASA Langley Research Center, Lawrence Livermore National Laboratory, the National Energy Research Scientific Computing Center (NERSC), Sandia National Laboratories, and SGI-Cray.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

The rapidly evolving field of parallel computing has come a long way, passing through several overlapping yet distinct phases [36]. In the past, the emphasis was on solving grand size problems faster. However, today we want to solve these problems not only fast computationally, but with rapid turnaround time (including the time taken to develop the new application on a new architecture) as well. On one hand, we want scalable and portable algorithms while on the other, we want tunable, general purpose software. We want to harness the advantages of object oriented technology without compromising performance. In addition, multiprocessing is becoming increasingly attractive, thanks to the evolution of highly integrated microprocessors and memory chips. Within the next decade, it should be feasible to integrate a billion transistors on a reasonably sized silicon chip [44]. At this level of integration, there is a growing need for finding a very high degree of concurrency to utilize the offered processing power. Though architectural concurrency is easy to achieve, algorithmic concurrency to match is less so in scientific codes. Intuitively, this is due to global domains of influence in many problems presented to the computer as implicitly discretized operator equations — implicitness being all but legislated for the multiscale systems of global climate, transonic airliners, petroleum reservoirs etc., the simulation of which justifies expenditures for the highest-end machines.

These architectural trends, characteristic of the modern computing technology in general, are of great significance for the field of scientific computing [21], which tries to simulate complex physical phenomena that are difficult to study experimentally. The study of computational science is not a proper subset of *numerical analysis* [49], where emphasis is laid on the optimal complexity for a specified level of accuracy implicitly assuming flat memory [36]. With future

The model journal used for this dissertation is *BIT*.

machines having deep memory hierarchies, there is bound to be much stronger algorithmic and architectural interaction, rendering the arithmetic complexity estimates increasingly irrelevant at the implementation level.

The architecture of the terascale systems available these days, built around hierarchical distributed memory, is hostile to conventional sequential optimal PDE algorithms in some respects. The distributed aspects must be overcome with judicious combinations of message-passing and shared memory programming models. The hierarchical aspects must be overcome with register blocking, cache blocking, and prefetching. Algorithms for PDE-based simulations must be highly concurrent and straightforward to load balance, latency tolerant, cache friendly (with strong temporal and spatial locality of reference), and highly scalable (in the sense of convergence rate) as problem size and processor number are increased in proportion. The goal for algorithmic scalability is to fill up the memory of arbitrarily large machines while preserving constant (or at most logarithmically growing) running times with respect to a proportionally smaller problem on one processor. Domain-decomposed multilevel methods are natural for all of these requirements. We argue that domain decomposition is natural for the software engineering of simulation codes. Valuable legacy code designed for a sequential PDE analysis can often be reused subdomain-by-subdomain in the solver framework of a parallel library.

I.1 PRESENT STUDY — SCOPE AND OUTLINE

Scope

The motivation behind this work is to demonstrate the feasibility of a highly scalable parallel solver for PDEs within an object oriented library framework. In order to achieve this goal, we identify the following relevant issues that this work attempts to address.

- Available memory bandwidth on most contemporary processors is insufficient to match the

available processing power. Therefore, the data structure design needs to optimize the use of this scant resource.

- The reasons behind the wide gap between peak performance and observed performance of PDE-based scientific computing codes need to be investigated. This, in turn, requires the development of performance models that are realistic and consider the effect of architectural parameters (like memory bandwidth, instruction issue rate, etc.).

- Until automated tools like parallel compilers or source-to-source translators can discover enough of the concurrency that is latent in most scientific computations, manual expression of that concurrency is the only alternative for achieving high performance. Within a well-defined class of applications, therefore, parallel libraries are a natural solution.

- The legacy scientific and engineering software (which holds a great investment of effort) needs to be reused within the library framework, which promises to provide several nice features such as a large variety of high performance linear and nonlinear solvers, a high level view of communication, interfaces to multiple programming languages, etc.

- If the integrated library based solver is going to be accepted in the scientific computing community, it must be superior to the legacy code in execution time and memory consumption. This requires case-specific performance tuning at the implementation and algorithmic level.

Outline

The rest of this document is organized as follows. We provide a review of the algorithms employed in this work in Chapter II. We illustrate the intelligent reuse of existing software in Chapter III by presenting the details of porting a legacy sequential (FUN3D from NASA Langley) code into an object oriented library (PETSc from Argonne National Laboratory). In the subsequent

chapters, we discuss three factors in parallel performance: per-processor performance, scalable parallel implementation, and scalable algorithms. In Chapter IV, we demonstrate significant performance gains arising from some data layout and reordering techniques (like interlacing of field variables, cache and register blocking etc.). We provide simple performance models to gain quantitative understanding of the effect of these techniques on the architectural resource requirements (like memory bandwidth, instruction scheduling). We also document the experimental evidence of their effectiveness in terms of TLB and data cache miss rates, using hardware counters. In Chapter V, we discuss the bottlenecks to implementation scalability in terms of architectural parameters. Specifically we identify the computational phases that tend to take an increasingly large fraction of wall clock time as we grow the number of processors (to several thousands). In the same chapter, we also describe the performance gains coming from the use of a hybrid (mixed distributed/shared) programming model, which is natural on clusters with more than one processor per node. We point out that this performance gain comes primarily from better algorithmic convergence rates as compared to the pure message passing case, since the former can work with smaller number of subdomains.

CHAPTER II
ALGORITHMIC CONTEXT

To understand the algorithms used in this dissertation, let us consider the following system of partial differential equations (PDEs)

$$V\frac{\partial \mathbf{u}}{\partial t} + \mathcal{F}(\mathbf{u}) = 0, \tag{1}$$

where $\mathbf{u}$ is a vector of functions depending upon spatial variables $\mathbf{x}$ and time t and $\mathcal{F}$ is a vector of spatial differential operators acting on $\mathbf{u}$. In finite element methods, V is called *mass matrix* and has diagonal and off-diagonal entries. In finite volume methods (such as the one employed in this work), it has only nonnegative diagonal entries. The advantage of this formulation is that it allows an arbitrary combination of PDEs with or without a temporal term (with zero diagonal entries in V for some equations).

Semidiscretizing in space to approximate $\mathcal{F}(\mathbf{u})$ with $\mathbf{f}(\mathbf{u})$, and in time with implicit Euler, we get the algebraic system:

$$(\frac{V}{\Delta t^{\ell}})\mathbf{u}^{\ell} + \mathbf{f}(\mathbf{u}^{\ell}) = (\frac{V}{\Delta t^{\ell}})\mathbf{u}^{\ell-1}. \tag{2}$$

Here we have used only first order discretization for the temporal term for simplicity. For steady state problems, $\Delta t^{\ell} \rightarrow \infty$ as $\ell \rightarrow \infty$. This discretization is termed as implicit since $\mathbf{f}(\mathbf{u})$ is evaluated at the current time step (in contrast to an explicit scheme where it is evaluated using $\mathbf{u}$ at the previous time step). We prefer to use implicit schemes since they offer numerical stability for stiff time-dependent PDEs and allow much larger time steps (as compared to explicit schemes) for steady-state PDE formulations solved in a pseudo-transient manner.

II.1 THE NEWTON-KRYLOV-SCHWARZ FAMILY OF ALGORITHMS

The implicit PDE solution algorithm employed in this work is based on Jacobian-free Newton-Krylov methodology, using overlapping domain decomposition (Schwarz) method as precondi-

tioner. The primary benefits of this class of algorithms are ease of parallelization with good scalability and their optimal convergence rate (when used with a coarse grid). In this section, we discuss different components of the Newton-Krylov-Schwarz (NKS) algorithm without the temporal term of Equation 1. To improve the robustness of NKS, we employ a pseudo-transient continuation technique, which is discussed in Section II.2.

Newton Methods

In the case of a nonlinear PDE, a linearization step is first performed and then the resulting linear system is solved. For example, to solve $\mathbf{F}(\mathbf{u}) = 0$, where,

- $\mathbf{F}: \Re^n \longrightarrow \Re^n$,
- There exists $\mathbf{u}^* \in \Re^n$ such that $\mathbf{F}(\mathbf{u}^*) = 0$,
- $\mathbf{F}$ is continuously differentiable within a neighborhood of $\mathbf{u}^*$,
- $\mathbf{F}'(\mathbf{u}^*)$ is nonsingular,

the Newton iterative method is as follows [21].

1. $k \longleftarrow 1$.
2. Solve $\mathbf{F}'(\mathbf{u}^{k-1})\delta\mathbf{u}^k = -\mathbf{F}(\mathbf{u}^{k-1})$.
3. Set $\mathbf{u}^k = \mathbf{u}^{k-1} + \lambda^k \delta\mathbf{u}^k$, where λ^k is a trust region based step length control [6] and $\mathbf{u}^0$ is the initial guess.
4. If converged, stop.

   Otherwise, $k \longleftarrow k + 1$ and goto step (2).

When the linear system in step (2) of the Newton process is solved only approximately or $\mathbf{F}'(\mathbf{u}^{k-1})$ is not evaluated as the true Jacobian of the right hand side vector $\mathbf{F}(\mathbf{u}^{k-1})$, this algorithm becomes an "inexact Newton Method". The primary reason for using an inexact Newton method is to save execution time though it may take more iterations (than true Newton method) to converge. There can be many choices for the inexact Jacobian [38]:

1. use an implicitly defined operator whose action on a vector can be approximated with multivariate Taylor series expansions;

2. use automatic differentiation software such as ADIFOR [8] or ADIC [9];

3. use a matrix that is constructed from finite differences of $\mathbf{F}$;

4. use a matrix that is derived from a discretization different from but related to that used for $\mathbf{F}$; or

5. use a matrix that was evaluated during some previous iteration (based on some previous state vector; this matrix is also called "lagged" Jacobian).

The first method is useful for Krylov methods where matrix elements themselves are never needed but the action of the matrix on any arbitrary Krylov vector (v) is represented as $\mathbf{F}'(\mathbf{u})\mathbf{v} \approx \frac{1}{h}[\mathbf{F}(\mathbf{u}+h\mathbf{v})-\mathbf{F}(\mathbf{u})]$. This method requires one extra function evaluation (more for higher orders) but saves large amount of memory (since Jacobian is not explicitly formed). The parameter h is chosen based on approximation error and floating point cancellation error (encountered while subtracting two close floating point numbers) [28]. This approach is termed as "matrix-free" or "Jacobian-free" Newton-Krylov method. The second method (using automatic differentiation) can also be employed to carry out the Jacobian matrix vector product simultaneously with $\mathbf{F}$ at a cost roughly 2.5 times the cost of $\mathbf{F}$ itself. In the method (3) above, the Jacobian matrix is

constructed explicitly element-by-element through a sequence of finite differences. For example, $[\mathbf{F}'(\mathbf{u})]_{ij} = \frac{\partial \mathbf{F}_i}{\partial \mathbf{u}_j}(\mathbf{u}) \approx \frac{1}{h}[\mathbf{F}_i(\mathbf{u} + h\mathbf{e}_j) - \mathbf{F}_i(\mathbf{u})]$, where h is a differencing parameter and $\mathbf{e}_j$ is the jth unit vector. The last two methods are used to derive useful preconditioners for the iterative linear solvers.

Newton-Krylov Methods

In step (2) of the Newton process, a Krylov method can be used to solve the linear equation. This combination is called a Newton-Krylov (NK) method. *Krylov subspace methods* make up a wide class of iterative methods with members that may be specialized to various classes of linear systems. A popular method for nonsymmetric systems is the *generalized minimum residual* (**GMRES** [45]) method, which has been used in this work as well. GMRES has the property of minimizing the norm of residual vector over a Krylov subspace at every step.

To solve $Ax = b$, GMRES forms a well conditioned basis $V_m = \{v_1, v_2, \ldots, v_m\}$ $(n \ll m)$ from the result of a Gram-Schmidt process on the Krylov space $\{r_0, Ar_0, \ldots, A^{m-1}r_0\}$. It then chooses the solution from the column space of V_m that minimizes the residual $\| b - Ax \|$. GMRES uses the matrix A to do the matrix-vector multiplications only. Therefore, the individual elements of A are never needed if we have some means (such as the matrix-free method described above) of doing the matrix-vector products. When the matrix A represents the Jacobian of a discretized system, each of these matrix-products is equivalent to one stencil update of an explicit scheme in terms of computational and communication cost.

Full GMRES has a nonincreasing residual and cannot break down short of delivering the exact solution or running out of memory. For a $n \times n$ problem GMRES terminates in at most n steps. Usually, due to memory limitations we set the size of Krylov space to n_r and GMRES is restarted if convergence is not attained after n_r steps.

Newton-Krylov-Schwarz Methods

When a Newton-Krylov method uses a Krylov-Schwarz KS method (such as restricted additive Schwarz [14]) to solve the linear system, the resulting algorithm is termed "Newton-Krylov-Schwarz" [27]. Schwarz methods [13, 20, 48] are divide-and-conquer methods for solving PDE problem $\mathcal{L}u = f$ in Ω, through solving a sequence of problems $\mathcal{L}'_i u'_i = f'_i$ in subdomains Ω_i covering Ω, and iteratively combining the partial solutions u'_i to form u. Thus, Schwarz methods partition a solution space into multiple subspaces (possibly overlapping) and form an approximate inverse of the operator in each subspace.

In KS methods, the linear system is preconditioned with domain-decomposed additive Schwarz method [48]. The original system $Ax = b$ is converted to an equivalent form $B^{-1}Ax = B^{-1}b$ through the action of a preconditioner B. It is presumed that B^{-1} is cheaper to calculate than A^{-1}. In KS methods, B^{-1} is constructed on a subdomain-by-subdomain basis through suitable approximations to a local Jacobian.

In this work, we have primarily used a block Jacobi preconditioner, which is a degenerate kind of Schwarz-style domain decomposition method. The block Jacobi preconditioner can be expressed as $B^{-1} = R_0^T J_{0,u^l}^{-1} R_0 + \sum_{i=1}^{K} R_i^T J_{i,u^l}^{-1} R_i$ where J_{i,u^l} is the local Jacobian for subdomain i, subscript "0" corresponding to a possible coarse grid, u^l is the vector of local grid values, R_i is a restriction operator for subdomain i, and R_i^T is an extension operator. All subdomains can be computed simultaneously with this method due to the lack of coupling among them. When an overlapping Schwarz scheme is used instead of a non-overlapping, there exists a certain degree of coupling among subdomains, but the computations on each are still potentially concurrent. This scheme will increase communication costs but improve convergence (see Section V.5).

A Schwarz preconditioner can be used with a matrix-free method provided some local representation of the operator exists for preconditioning. Left preconditioning of the Jacobian J with

an operator B^{-1} can be obtained through

$$B^{-1}J(\mathbf{u}^l)\mathbf{u} \simeq \frac{1}{h_l}[B^{-1}\mathbf{F}(\mathbf{u}^l + h_l\mathbf{u}) - B^{-1}\mathbf{F}(\mathbf{u}^l)],$$

and right preconditioning via

$$J(\mathbf{u}^l)B^{-1}\mathbf{u} \simeq \frac{1}{h_l}[\mathbf{F}(\mathbf{u}^l + h_l B^{-1}\mathbf{u}) - \mathbf{F}(\mathbf{u}^l)],$$

where h_l is a suitably chosen small scalar.

It can be shown that the two-level Schwarz method (with good overlap of subdomains) has a convergence rate that is independent of the number of subdomains and fineness of the discretization, like a traditional multilevel iterative method [11, 12, 29, 52]. However, the two-level method shares with multilevel methods a nonscalable cost-per-iteration from the necessity of solving a coarse-grid system of size $\mathcal{O}(P)$, where P is the number of subdomains. Unlike recursive multilevel methods, a two-level Schwarz method may have a rather fine coarse grid, for example, $H = \mathcal{O}(h^{1/2})$, 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. When reuse is possible, storing a parallel inverse can be cost-effective [53].

The convergence rates and overall parallel efficiencies (see Section V.1) of single-level additive Schwarz methods are dependent upon the number and shape of the subdomains. The number of iterations to convergence and the communication overhead per iteration generally increase with increasing numbers of subdomains, problem size being held constant.

In practical large-scale applications, however, the convergence rate degradation of fine-grain single-level additive Schwarz is often not as serious as the scalar, linear elliptic theory would suggest [33]. Its effects are mitigated by several factors, including an outer context of nonlinearity and pseudo-transient continuation, and strong intercomponent coupling that can be captured exactly in a point-block ILU preconditioner. Another "forgiveness factor" for additive Schwarz, in practice,

is the convenience with which Schwarz-based preconditioners can be made to play to the cache in modern microprocessors. This is discussed further in Section V.5

II.2 PSEUDO-TRANSIENT NEWTON-KRYLOV-SCHWARZ ALGORITHM

For time-accurate solutions of Equation 2, Newton's method normally converges rapidly since the initial guess is not too far from the actual solution on each timestep. However, for steady-state nonlinear problems ($\mathbf{f}(\mathbf{u}) = 0$), Newton's method may not converge for an arbitrary initial guess. To improve the robustness of Newton's method for steady-state problems, an unsteady form of the original boundary value problem (BVP) is solved like an initial value problem (IVP). This is called pseudo-transient continuation [35]. Initially, when time step Δt is small, we get a better conditioned linear system (as compared to the one obtained just with the steady-state form). As the solution evolves (and approaches the ball of convergence for the Newton method), Δt grows to very large values and we obtain the steady-state solution asymptotically. When pseudo-transient continuation is employed in conjunction with a NKS method, we obtain the pseudo-transient Newton-Krylov-Schwarz (ΨNKS) algorithm [34, 37] as shown in Figure 1.

A time-stepping scheme is required to complete the algorithm. One choice is successive evolution-relaxation (SER) [43], which lets the time step grow in inverse proportion to residual norm progress:

$$\Delta t^{\ell} = \Delta t^{0} \frac{\|f(u^{0})\|}{\|f(u^{\ell-1})\|}. \tag{3}$$

where ℓ is the current time step number and $\|f(u^{\ell-1})\|$ is the 2-norm of the residual vector at the previous time step ($\ell - 1$). This is the scheme employed in this work (see Section V.5 for further details).

Alternatively, a temporal truncation error strategy bounds the maximum temporal truncation error in each individual component, based on a local estimate for the leading term of the error.

```
do l = 1, n_time
   SELECT TIME-STEP
   do k = 1, n_Newton
      compute nonlinear residual and Jacobian
      do j = 1, n_Krylov
         do i = 1, n_Precon
            solve subdomain problems concurrently
         enddo
         perform Jacobian-vector product
         ENFORCE KRYLOV BASIS CONDITIONS
         update optimal coefficients
         CHECK LINEAR CONVERGENCE
      enddo
      perform vector update
      CHECK NONLINEAR CONVERGENCE
   enddo
enddo
```

Figure 1: Pseudo-transient Newton-Krylov-Schwarz algorithm. The operations written in upper-case customarily involve global synchronizations, about which we comment in Chapter V.

Here step size is controlled through its relation to the truncation error. Another way is to set an upper limit for change in each component of the state vector and adjust the time step so as to bring the measured change to the target. Typically, the time step is not allowed to more than double in a favorably converging situation, or to be reduced by more than a factor of ten in an unfavorable one, unless feasibility is at stake, in which case the time step may be drastically cut [35, 38].

To use ΨNKS in PDEs effectively, we may need to tune several parameters, such as the initial time step size, scaling of the differencing parameter in the matrix-free application of the Jacobian, the convergence tolerance of the inner Krylov iterations, etc. We explain the parameter tuning process in detail for ΨNKS algorithm as employed in this work (in the context of a large scale CFD problem) in Section V.5.

CHAPTER III

PETSc-FUN3D CODE

This chapter discusses the details of the PETSc-FUN3D code, which has been used to carry out the performance studies described in Chapters IV and V. First some features of the original (sequential) NASA FUN3D code are discussed. Next we highlight our approach to parallelize an unstructured implicit solver using the "Portable, Extensible Toolkit for Scientific Computing" (PETSc) [5, 7, 6] library. We also characterize the different computational phases based on the architectural parameters they stress.

III.1 NASA CODE FUN3D

The legacy 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, 3]. FUN3D uses a control volume discretization with variable-order Roe schemes for approximating the convective fluxes 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 mesh points. The optimization loop involves many analysis cycles. Thus, reaching the steady-state solution in each analysis cycle in a reasonable amount of time is crucial to conducting the design optimization. From the beginning, our effort has been focused on minimizing the time to convergence without compromising scalability, by means of appropriate algorithms and architecturally efficient data structures.

Thus far, our large-scale parallel experience with PETSc-FUN3D is with the compressible or incompressible Euler subset, but nothing in the solution algorithms or software changes with additional physical phenomenology. Of course, the convergence rate will vary with conditioning,

as determined by Mach and Reynolds numbers and the correspondingly induced mesh adaptivity. Furthermore, robustness becomes more of an issue in problems admitting shocks or using 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 toward the steady-state limit, may be a *more*, not less, severe test of parallel performance.

III.2 PARALLELIZATION METHODOLOGY

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.

In addition, loss of *regularity* in unstructured grid solvers makes them more *memory* and *integer operation intensive*; nevertheless, a library-based solver should be competitive *in serial* with a legacy solver in terms of memory and execution time. These were our challenges in porting FUN3D into PETSc.

The key points of our SPMD implementation are 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 dependencies 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 sequential 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 prohibitive at high granularity.

- We enforce a local ordering on the locally-owned nodes; ghost nodes are 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 2 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.

III.3 PARALLEL IMPLEMENTATION USING PETSC

To implement ΨNKS methods on distributed memory parallel computers, we employ the PETSc library that attempts to handle, through a uniform interface and in a highly efficient way, the low-level details of the distributed memory hierarchy. Examples of such details include striking the right balance between buffering messages and minimizing buffer copies, overlapping communication and computation, organizing node code for strong cache locality, preallocating memory in sizable chunks rather than incrementally, and separating tasks into one-time and every-time subtasks using the inspector/executor paradigm. The benefits to be gained from these and from other numerically neutral but architecturally sensitive techniques are so significant that it is efficient in

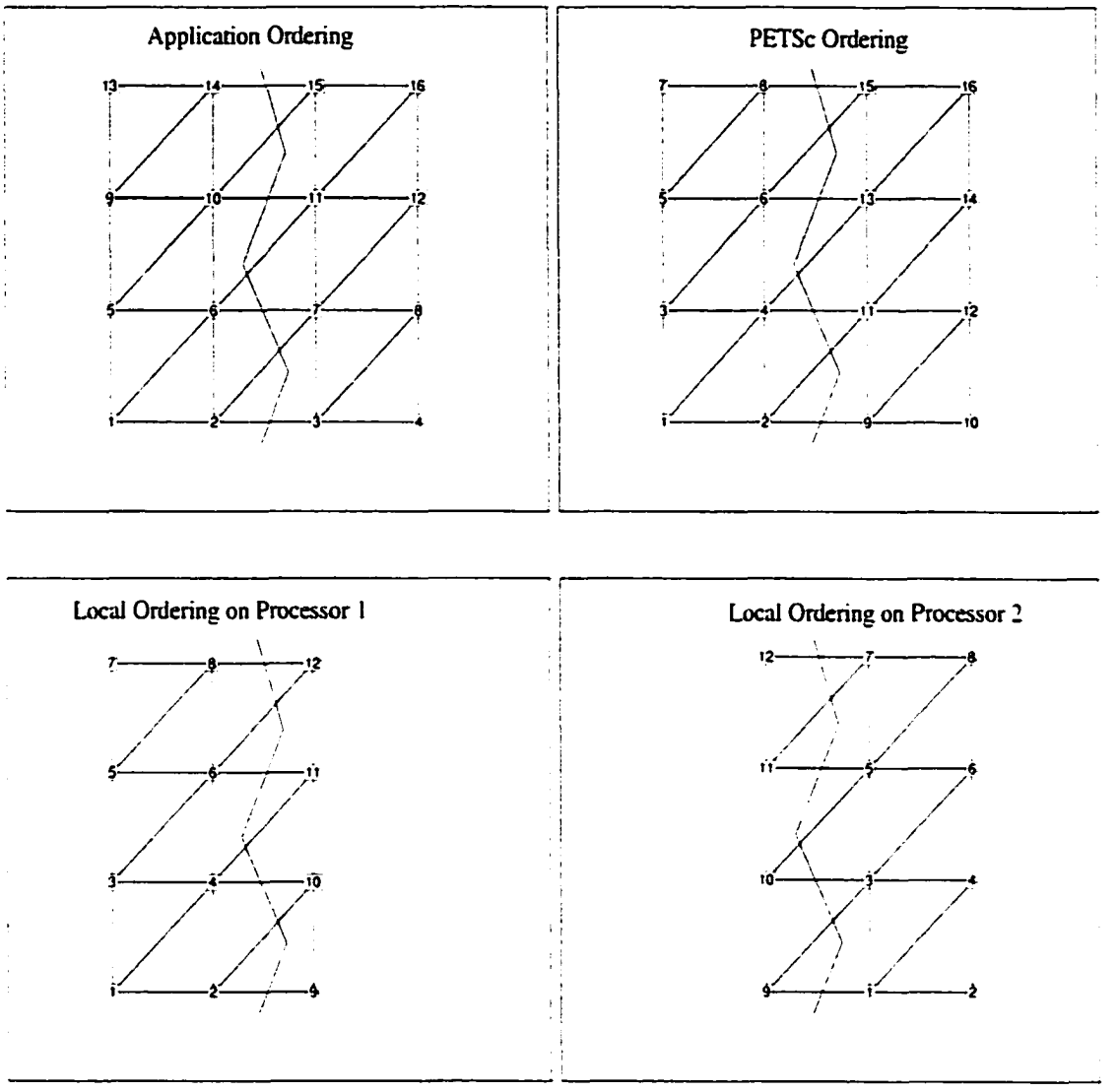


Figure 2: Illustration of three different orderings (user endowed, global library endowed, and local library endowed) for the two-way partitioning of a simple mesh. Note that within each partitioning halo vertices are ordered last and that orderings are contiguous within a partition.

both the programmer-time and execution-time senses to express them in general purpose code.

PETSc is a large and versatile package integrating distributed vectors, distributed matrices in several sparse storage formats, Krylov subspace methods, preconditioners, and Newton-like nonlinear methods with built-in trust region or line search strategies and continuation for robustness. It has been designed to provide the numerical infrastructure for application codes involving the implicit numerical solution of PDEs, and it sits atop MPI for portability to most parallel machines. The PETSc library is written in C, but may be accessed from user codes written in C, FORTRAN, and C++. PETSc has many features relevant to PDE analysis, including matrix-free Krylov methods, blocked forms of parallel preconditioners, and various types of time-stepping.

A diagram of the calling tree of a typical NKS application appears in Figure 3. The arrows represent calls that cross the boundary between application-specific code and PETSc library code; all internal details of both are suppressed. The top-level user routine performs I/O related to initialization, restart, and post-processing and calls PETSc subroutines to create data structures for vectors and matrices and to initiate the nonlinear solver. PETSc calls user routines for function evaluations $\mathbf{f}(\mathbf{u})$ and (approximate) Jacobian evaluations $\mathbf{f}'(\mathbf{u})$ at given vectors $\mathbf{u}$ representing the discrete state of the flow. Auxiliary information required for the evaluation of $\mathbf{f}$ and $\mathbf{f}'(\mathbf{u})$ that is not carried as part of $\mathbf{u}$ is communicated through PETSc via a user-defined "context" that encapsulates application-specific data. (Such information typically includes dimensioning data, grid data, physical parameters, and quantities that could be derived from the state $\mathbf{u}$, but are most conveniently stored instead of recalculated, such as constitutive quantities.)

When well tuned, large-scale PDE codes spend almost all of their time in two phases: flux computations to evaluate conservation law residuals, called "function evaluations" in Figure 3, 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

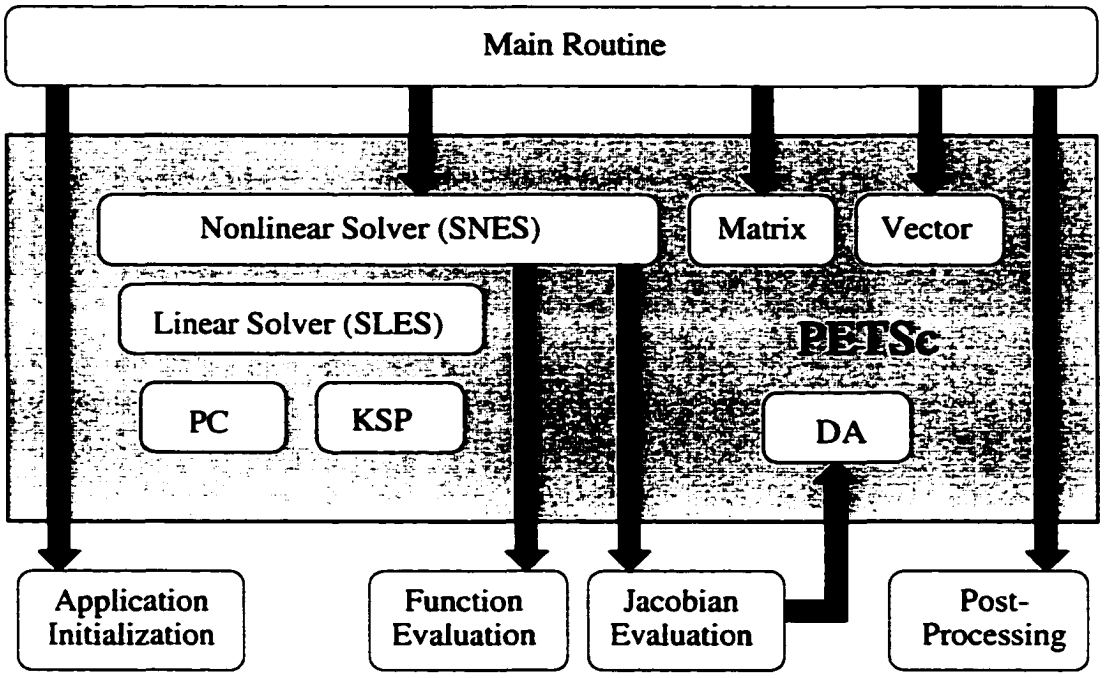


Figure 3: Coarsened calling tree of the PETSc-FUN3D code, showing the user-supplied main program and callback routines for providing the initial nonlinear iterate, computing the nonlinear residual vector at a PETSc-requested state, and evaluating the Jacobian (preconditioner) matrix.

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.

III.4 COMPLEXITY ANALYSIS OF PETSC-FUN3D

As mentioned above, there are four groups of tasks in a typical PDE solver (like PETSc-FUN3D), each with a distinct proportion of work to datasize to communication requirements. In the language of a vertex-centered code, in which the data is 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.

III.5 TEST PROBLEMS

We test the code on the ONERA M6 wing, a standard three-dimensional test case, for which extensive experimental data is given in [46]. Figure 4 shows the surface mesh of computational domain around ONERA M6 wing. We have used the following three tetrahedral meshes in this work.

- Very large mesh (called GridA) with about 2.8 million vertices and 19 million edges
- Large mesh (called GridB) with about 357,900 vertices and 2.4 million edges
- Medium size mesh (called GridC) with 22,700 vertices and 146,384 edges

The sequential performance studies (Chapter IV) have used the GridC mesh while the parallel cases (Chapter V) use primarily GridA (and sometimes GridB).

A frequently studied parameter combination combines a freestream Mach number of 0.84 with an angle of attack of 3.06^o. This transonic case gives rise to a characteristic λ-shock (Figure 5). We have used this case with the compressible version of the code.

Algorithmic setting

We employ backward Euler to do the time integration while advancing the time step using the SER heuristic of Van Leer and Mulder [43] (see Chapter V). Within each time step, the nonlinear problem is solved using an inexact Newton method (by only doing one Newton iteration per step). The linear problem is solved using restarted GMRES. We primarily use block Jacobi preconditioner (zero overlap) where a subdomain is mapped to a single processor (or process in the

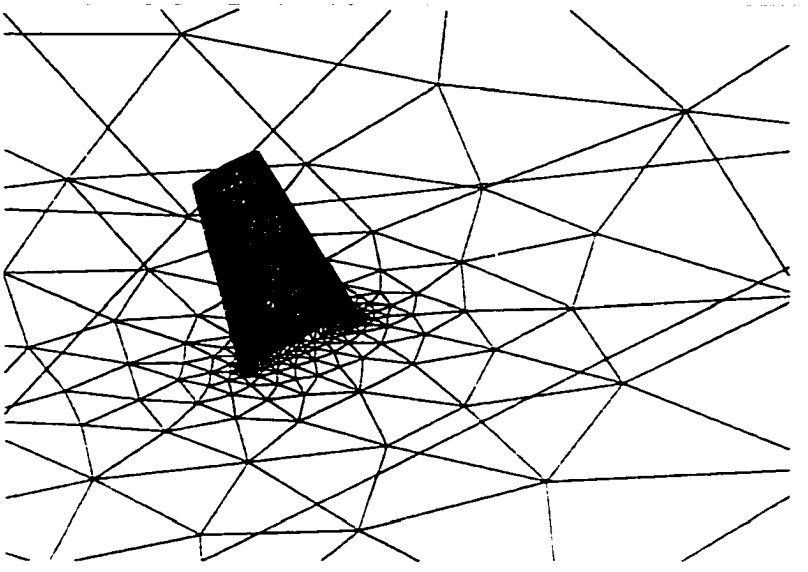


Figure 4: Surface visualization of the ONERA M6 Wing.

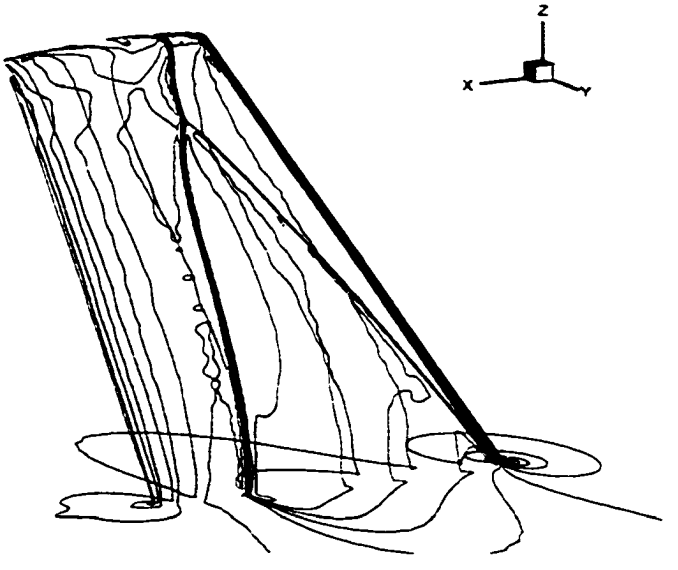


Figure 5: Mach contours on the ONERA M6 Wing at freestream Mach number = 0.839.

hybrid programming case). We use incomplete factorization (ILU) with in each domain. These algorithms require tuning of several parameters for optimal performance. The tuning process is described in detail in Chapter V.

III.6 HISTORY OF THE PETSC-FUN3D PROJECT

This project was started in October, 1996 in collaboration with NASA and Argonne National Laboratory. 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.

Inasmuch as FUN3D was originally written for vector machines, many data layout transformations [33] 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 association 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.

CHAPTER IV

SINGLE PROCESSOR PERFORMANCE STUDIES

Traditionally, numerical analysts have evaluated the performance of algorithms by counting the number of floating-point operations. It is well known that this is not a good estimate of performance on modern computers; for example, the performance advantage of the level-2 and level-3 BLAS over the level-1 BLAS for operations that involve the same number of floating-point operations is due to better use of memory, particularly the reuse of fast memory [18, 19]. Paradoxically, however, the success of the level-2 and level-3 BLAS at reaching near-peak levels of performance has obscured the difficulties faced by many other numerical algorithms. On the algorithmic side, tremendous strides have been made; many algorithms now require only a handful of floating-point operations per mesh point. On the hardware side, however, memory system performance is improving at a rate that is much slower than that of processor performance [30, 41]. The result is a mismatch in capabilities: algorithm design has minimized the work per data item, but hardware design predicated upon on executing an increasingly large number of operations per data item.

The importance of memory bandwidth to the overall performance is suggested by the performance results shown in Figure 6. These show the single-processor performance for our code, PETSc-FUN3D [23, 33]. The performance of PETSc-FUN3D is compared with the peak performance and the results of the STREAM benchmark [41], which measures achievable performance for memory-bandwidth limited computations. The figure shows that the STREAM results are a much better indicator of performance than are the peak numbers. To illustrate the performance limitations caused by insufficient available memory bandwidth, we discuss in Section IV.1 a sparse matrix-vector multiply algorithm, a critical operation in many iterative methods used in implicit CFD codes.

Even for computations that are not memory intensive, computational rates often fall far short

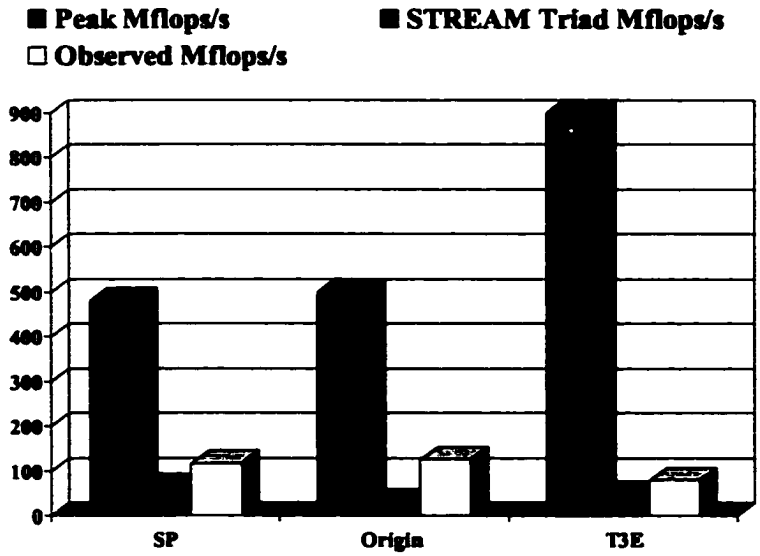


Figure 6: Sequential performance of PETSc-FUN3D for a small grid of 22,677 vertices (with 4 unknowns per vertex) run on a 120 MHz IBM SP, a 250 MHz SGI Origin2000, and a 450 MHz Cray T3E.

of peak performance. This is true for the flux computation in our code, even when the code has been well tuned for cache-based architectures [34]. We show in Section IV.3 that instruction scheduling is a major source of the performance shortfall in the flux computation step.

This chapter focuses on the per-processor performance of compute nodes used in parallel computers. Our experiments have shown that PETSc-FUN3D has good scalability [23, 2, 39] However, since good per-processor performance reduces the fraction of time spent in computation as opposed to communication, achieving the best per-processor performance is a critical prerequisite to demonstrating *uninflated* parallel performance [4].

IV.1 PERFORMANCE ANALYSIS OF SPARSE MATRIX-VECTOR PRODUCT

The sparse matrix-vector product is an important part of many iterative solvers used in scientific computing. While a detailed performance modeling of this operation can be complex, particularly

when data reference patterns are included [50, 51, 54], a simplified analysis can still yield upper bounds on the achievable performance of this operation. To illustrate the effect of memory system performance, we consider a generalized sparse matrix-vector multiply that multiplies a $m \times n$ matrix by N vectors, each with n elements. This code, along with operation counts, is shown in Figure 7.

Estimating the Memory Bandwidth Bound

To estimate the memory bandwidth required by this code, we make some simplifying assumptions. We assume that there are no conflict misses, meaning that each matrix and vector element is loaded into cache only once. We also assume that the processor never waits on a memory reference, that is, any number of loads and stores are satisfied in a single cycle.

For the algorithm presented in Figure 7, the matrix is stored in compressed row storage format (similar to PETSc's AIJ format [7]). For each iteration of the inner loop in Figure 7, we transfer one integer (`ja` array) and $N + 1$ doubles (one matrix element and N vector elements), and we do N floating-point multiply-add (`fmadd`) operations or $2N$ flops. Finally, we store the N output vector elements. We get the following estimate of the data volume:

$$\begin{aligned}\text{Total Bytes Transferred} &= m * \text{sizeof\_int} + 2 * m * N * \text{sizeof\_double} \\ &\quad + N_{nz} * (\text{sizeof\_int} + \text{sizeof\_double}) \\ &= 4 * (m + N_{nz}) + 8 * (2 * m * N + N_{nz}).\end{aligned}$$

where we have assumed that the matrix is square of dimension m. This equation provides an estimate of the bandwidth required by the processor to do $2N_{nz}$ N flops at the peak speed:

$$\text{Bytes Transferred/fmadd} = \left(16 + \frac{4}{N}\right)\frac{m}{N_{nz}} + \frac{12}{N}.$$

Alternatively, given the memory performance, we can predict the maximum achievable perfor-

```
for (i = 0, i < m; i++) {                        // loop over rows
  jrow = ia(i)
  ncol = ia(i+1) - ia(i)                         // 1 Of, AT, Ld, Iop
  Initialize, sum1, ..., sumN                    // N Ld
  for (j = 0; j < ncol; j++) {                   // 1 Ld
    fetch ja(jrow), a(jrow),
          x1(ja(jrow)), ..., xN(ja(jrow))        // 1 Of, N+2 AT, N+2 Ld
    do N fmadd (floating multiply add)           // 2N Fop
    jrow++
  }                                              // 1 Iop, 1 Br
  Store sum1, ..., sumN in
        y1(i), ..., yN(i)                        // 1 Of, N AT, N St
}                                                // 1 Iop, 1 Br
```

Figure 7: General form of sparse matrix-vector product algorithm. The storage format is AIJ or compressed row storage. The matrix has m rows and N_{nz} non-zero elements and gets multiplied with N vectors. The comments at the end of each line show the assembly level instructions the current statement generates, where `AT` is address translation, `Br` is branch, `Iop` is integer operation, `Fop` is floating-point operation, `Of` is offset calculation, `LD` is load, and `St` is store.

Format	Number of vectors	Bytes/fmadd	Bandwidth (MB/s)		Mflops/s	
			Required	Achieved	Ideal	Achieved
AIJ	1	12.36	3090	276	58	45
AIJ	4	3.31	827	221	216	120
BAIJ	1	9.31	2327	297	84	66
BAIJ	4	2.54	635	229	305	175

Table 1: Effect of memory bandwidth on the performance of sparse matrix-vector product on the SGI Origin2000 (250 MHz R10000 processor). The STREAM benchmark memory bandwidth [41] is 358 MB/s; this value is used to calculate the ideal Mflops/s. The achieved values of memory bandwidth and Mflops/s are measured by using hardware counters on this machine. Our experiments show that we can multiply four vectors in 1.5 times the time needed to multiply one vector.

mance, as follows:

$$M_{BW} = \frac{2}{(16 + \frac{4}{N})\frac{m}{N_{nz}} + \frac{12}{N}} \times BW. \tag{1}$$

where M_{BW} is measured in Mflops/sec and BW stands for the available memory bandwidth in Mbytes/s, as measured by the STREAM [41] benchmark. (The raw bandwidth based on memory bus frequency and width is not a suitable choice because it cannot be sustained in any application.)

In Table 1, we show the memory bandwidth required for peak performance and the achievable performance for a matrix in AIJ format with 90,708 rows and 5,047,120 non-zero entries on an SGI Origin2000 (unless otherwise mentioned, this matrix is used in all subsequent computations). The matrix is a typical Jacobian from a PETSc-FUN3D application (incompressible version) with four unknowns per vertex. The same table also shows the memory bandwidth requirement for

the block storage format (BAIJ) for this matrix with a block size of four; in this format, the `ja` array is smaller by a factor of the block size. We observe that the blocking helps significantly by cutting down on the memory bandwidth requirement. Having more than one vector also requires less memory bandwidth and boosts the performance: we can multiply four vectors in about 1.5 times the time needed to multiply one vector.

Estimating the Operation Issue Limitation

To analyze this performance bound, we assume that all the data items are in primary cache (equivalent to assuming infinite memory bandwidth). Referring to the sparse matrix-vector algorithm in Figure 7, we obtain the following composition of the workload for each iteration of the inner loop:

- $N + 5$ integer operations
- $2 * N$ floating-point operations (N `fmadd` instructions)
- $N + 2$ loads and stores

Most contemporary processors can issue only a single load or store in one cycle. Since the number of floating-point instructions is less than the number of memory references, the code must take at least as many cycles as the number of loads and stores. The performance bound is as follows:

$$M_{IS} = \frac{2N_{nz}N}{N_{nz}(N+2)+m} \times \text{Clock Frequency}. \tag{2}$$

where M_{IS} is in Mflops/sec and Clock Frequency is measured in MHz.

Estimating the Fraction of Floating Point Instructions

In order to estimate the limitation on floating point performance due to the high fraction of integer instructions in simulations using unstructured meshes, we need to study the assembly level code.

Figure 7 shows the matrix-vector algorithm along with the number and kind of instructions each statement will generate.

$$\text{Total number of instructions} = m * (3 * N + 8) + N_{nz} * (4 * N + 9) \tag{3}$$

On machines (such as the SGI Origin2000, IBM SP, etc.) with an fmadd instruction (where a floating point multiply-add is done with one instruction), we get the following estimate of the fraction of the floating point instructions.

$$\text{Fraction of floating point instructions} = \frac{2 * N * N_{nz}}{m * (3 * N + 8) + N_{nz} * (4 * N + 9)} \tag{4}$$

This fraction of floating point work will be twice the above value on the machines without fmadd instruction (like SUN Ultra II, Pentium II etc.). For the matrix that we have been considering, this fraction turns out be 18% for the AIJ case and 34% for the BAIJ case.

Because of the prevalence of superscalar processors (which can issue several instructions in one cycle), the number of floating point instructions alone does not tell us much about the performance we can expect. Nevertheless, it is a rough indicator of how much floating point work is there for the processor. It also suggests that it may be necessary to reduce the non-floating part of the workload to push the performance beyond a certain level. This point is explored further in the next section.

Performance Comparison

In Figure 8, 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 in Equation 1, and the performance based on operation issue limitation in Equation 2. For the sparse matrix-vector multiply, the memory-bandwidth

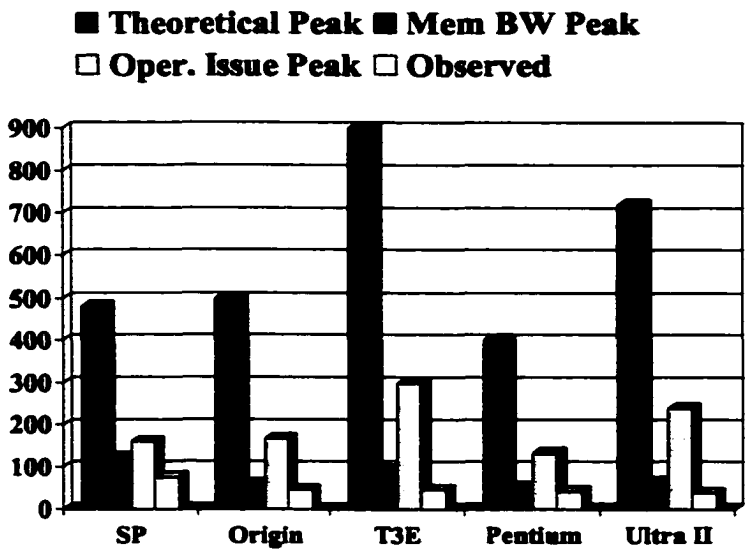


Figure 8: Three performance bounds for sparse matrix-vector product. The bounds based on memory bandwidth and instruction scheduling are much closer to the observed performance than the theoretical peak of the processor. Only one vector ($N = 1$) is considered here. The matrix size has m = 90,708 rows and N_{nz} = 5,047,120 nonzero entries. The processors are a 120 MHz IBM SP (P2SC "thin", 128 KB L1), a 250 MHz Origin2000 (R10000, 32 KB L1, and 4 MB L2), a 450 MHz T3E (DEC Alpha 21164, 8 KB L1, 96 KB unified L2), a 400 MHz Pentium II (running Windows NT 4.0, 16 KB L1, and 512 KB L2), and a 360 MHz SUN Ultra II (4 MB external cache); memory bandwidth values are taken from the STREAM benchmark Web site.

limit on performance clearly 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 likely to be invalid.

IV.2 CACHE MISS ANALYSIS OF THE MATRIX VECTOR PRODUCT

In this section, we discuss the different types of cache misses encountered in the sparse matrix vector algorithm. For simplicity, let us model the cache misses only for the level nearest to the

main memory since these are very expensive (the cache lines are loaded from memory). In the case of a single level cache processor, this model will directly apply to the primary cache.

Compulsory (or cold) Misses

For the matrix, most of the misses are compulsory misses. If we include both a and ja arrays (with the assumption that size of integer is half of size of double), then

$$\text{Number of Compulsory Misses for Matrix} = \frac{1.5N_{nz}}{w_{sc}} + \frac{n}{2w_{sc}}. \tag{5}$$

where the first term accounts for the data (a) and ja arrays and the second term arises from ia array. Similarly, for N input and N output vectors of size n, we have

$$\text{Number of Compulsory Misses for Vectors} = \frac{2Nn}{w_{sc}}. \tag{6}$$

If $2Nn \ll 1.5N_{nz}$, the compulsory misses for the matrix dominate. For the R10000 processor with cache line size w_{sc} = 16 double words and total cache size, C_{sc} = 512,000 double words, we can estimate the number of compulsory cache misses for the sample PETSc-FUN3D matrix (N_{nz} = 5,047,120, n = 90,708) with four vectors (N = 4). The compulsory misses for the matrix and vectors are 476,003 and 45,354 respectively.

Conflict (or Interference) Misses

These can be divided into two components :

- Self Interference: since the matrix is very large and is loaded only once, all the misses caused are compulsory. Therefore, even though there is self interference while loading the cache lines for the matrix, it is not significant since each cache line is used only once.

  For the vectors, these misses may be quite significant. In general, depending on the cache size (especially when it can not hold an entire vector), an input (and output) vector may map

on itself.

- Cross Interference: These are the misses caused when a vector maps on another vector in cache. When cache size is large enough to hold a vector, the cross interference between different vectors can be very serious.

Since the vector cache lines have the possibility of reuse, we would like to minimize the number of conflict misses. This can be done by suitable storage formats for the vectors (spatial locality) and proper scheduling of the instructions (temporal locality). To achieve this goal, let us first model the conflict misses for a banded matrix (that can serve as an upper bound on the number of conflict misses for the actual matrix with which we have been experimenting).

Modeling the Conflict Misses for a Banded Matrix

Consider a banded matrix with a row and column bandwidth of β. The memory reference patterns for the loop over rows can be visualized as a moving window (of size β) over the vector array. For each new row, a new vector element is referenced and the very first item in the moving window is not needed. Thus every w_{sc} rows a new cache line is needed while first cache line can be replaced. It should be noted that only those misses are be counted as conflict misses that replace cache lines needed again in the future. Here are some important observations regarding the reusability of cache lines :

- Each cache line satisfies $\beta \cdot w_{sc}$ references or is reused $\beta \cdot w_{sc}$ times provided it is not one of the first or last $\left\lceil \frac{\beta}{w_{sc}} \right\rceil$ cache lines.
- A cache line belonging to the set of first $\left\lceil \frac{\beta}{w_{sc}} \right\rceil$ cache lines satisfies $i \cdot w_{sc}$ references, $\forall i \in \left(1, \left\lceil \frac{\beta}{w_{sc}} \right\rceil\right)$

- A cache line belonging to the set of last $\left\lceil \frac{\beta}{w_{sc}} \right\rceil$ cache lines satisfies $\left(\left\lceil \frac{n}{w_{sc}} \right\rceil - i + 1 \right) \cdot w_{sc}$ references $\forall i \in \left(\left\lceil \frac{n-\beta}{w_{sc}} \right\rceil, \left\lceil \frac{n}{w_{sc}} \right\rceil \right)$

- At any time, we need only $\left\lceil \frac{\beta}{w_{sc}} \right\rceil$ cache lines of a vector to get the optimal performance, i.e. the working set size for each vector is $\left\lceil \frac{\beta}{w_{sc}} \right\rceil$ cache lines. This fact will be helpful in deciding if a cache miss is a real penalty or not.

Now depending on the cache size C_{sc} (in words), there are the following cases of interest:

- $N \cdot n < C_{sc}$ (i.e., when all the input vectors fit in the cache).

- $N \cdot \beta < C_{sc} < N \cdot n$ (when the working sets of all the input vectors fit in the cache).

- $N \cdot \beta > C_{sc}$ (when the working sets of some vectors fit in the cache).

Let us assume the following simple cache mapping policy:

$$\text{Cache Address} = (\text{Block Address}) \bmod (\text{Number of Sets in the Cache}) . \tag{7}$$

For the time being, let us assume a direct mapped cache. This assumption is reasonable for the two way mapped caches for the matrix-vector product (matvec) example, assuming one block in each set is occupied by matrix elements.

Consider two addresses (the term address refers to the block address There will be a conflict miss if

$$a \bmod C_{sc} = b \bmod C_{sc} \quad or \quad a \equiv b \bmod C_{sc}.$$

Case 1 when $N \cdot n < C_{sc}$

Here all the input vectors fit in cache. In this case, there will be no conflict misses among the vectors. The majority of the cache misses for this case are compulsory.

The R10000 processor with L2 cache size, C_{sc} = 512,000 (double words) and the GridC matrix with n = 90,708 falls into this category. The estimated number of total compulsory cache misses in this case (including matrix, input and output vectors), as estimated earlier, is 521,357. The total cache misses actually measured (using hardware counters) is 550,551. The difference 29,194 may be accounted by the cross interference between the matrix and the input and output vectors.

Case 2 when $N \cdot \beta < C_{sc} < N \cdot n$

Here the working set of all the input vectors fit in the cache and ideally, there should be no cache misses if we allocate the vectors carefully in the memory. But since the working sets are allocated at a distance in memory, these might conflict. We study the following two situations to understand this case better.

- Within a row (j loop): Assume that the vectors are laid out in the memory one after another. Also, without the loss of generality, we assume that the first address in the vector array maps to the first cache line. Now consider a memory location j for the first vector where j lies in the moving window $\left[i - \frac{\beta}{2}, i + \frac{\beta}{2}\right]$ for the row i (assuming a structurally symmetric matrix). Then there will be a cache miss if

$$\left\lceil \frac{j}{w_{sc}} \right\rceil \equiv \left\lceil \frac{j + I_v n}{w_{sc}} \right\rceil \bmod C_w \quad \forall I_v = 1, \ldots, N-1. \tag{8}$$

  where C_w is the cache size in terms of cache blocks. This relation basically implies that there will be a conflict miss when C_{sc} divides all the integral multiples of vector size n. Since C_{sc} is in powers of 2, this means that n is some multiple of a power of 2. We should avoid having such values for n. If we cannot, then some padding (which is discussed next) should be used to get the conflict-free mapping of vector cache lines.

- Loop over rows: While going through one iteration of the i-loop, the working set of each

vector is $\left\lceil \frac{\beta}{w_{sc}} \right\rceil$ cache lines. Since vectors are allocated one after another in memory, there is a danger that the working sets of two or more input vectors might overlap despite the fact that we have enough room for the working sets of all the vectors. One way to avoid this is to use padding at the end of a vector so that its successor maps beyond its working set in cache. To illustrate this, we consider the two vector ($N = 2$) case for simplicity. If p is the amount of padding in double words, then in order to avoid a conflict miss, the following should hold $\forall j \in \left[i - \frac{\beta}{2}, i + \frac{\beta}{2}\right]$

$$\left(\left\lceil \frac{j + n + p}{w_{sc}} \right\rceil \bmod C_w - \left\lceil \frac{j}{w_{sc}} \right\rceil \bmod C_w \right) \geq \frac{\beta}{w_{sc}}. \tag{9}$$

This simplifies to the following inequality,

$$(n + p) \bmod C_{sc} \geq \beta. \tag{10}$$

We observe that the above relation may lead to wasted (virtual) address space. This also suggests that our storage format for the vectors (non-interlaced, one after another) in memory is simple but may cause large number of conflict misses, even when the working set of all the vectors fit in the cache. We are investigating better storage patterns that will eliminate (or reduce) the conflict misses. The interlaced storage format is ideal for this case but is otherwise cumbersome to manage in the MatMultVec algorithm where we do not want to decide the number of vectors beforehand.

Case 3 when $N \cdot \beta > C_{sc}$

Let us estimate the conflict misses when we have only one vector (i.e. $N = 1$). In order to avoid a conflict miss, we need at least $\left\lceil \frac{\beta}{w_{sc}} \right\rceil$ cache lines. For each row, the number of replaced cache lines is $\left\lceil \frac{\beta - C_{sc}}{w_{sc}} \right\rceil$. Therefore, the total number of cache misses for the single vector case

is $n\left\lceil\frac{\beta-C_{sc}}{w_{sc}}\right\rceil$. Now assume that we do proper variable size padding to reduce the conflict cache misses (as discussed earlier in Case 2). Then, we can generalize the above expression to the case of N vectors since $N\beta$ is the minimum space needed in cache to avoid conflict misses.

Therefore, number of conflict misses for the N vectors is $n\left\lceil\frac{N\beta-C_{sc}}{w_{sc}}\right\rceil$. This is the minimum number of conflict misses we will have for this case. We need to add this extra data traffic while estimating the memory bandwidth bound.

The number of cache misses saved by using more than one vector (MatMultVec algorithm in Figure 7) is $N-1$ times the compulsory cache misses for the matrix i.e. $(N-1)\cdot\left\lceil\frac{1.5N_{nz}}{w_{sc}}\right\rceil$.

Therefore, the MatMultVec algorithm will be useful for this case only when

$$(N-1)\cdot\left\lceil\frac{1.5N_{nz}}{w_{sc}}\right\rceil \geq n\left\lceil\frac{N\beta-C_{sc}}{w_{sc}}\right\rceil \tag{11}$$

This inequality can be used to determine the appropriate value for the number of vectors that should be used to achieve any gain in performance.

IV.3 SEQUENTIAL PERFORMANCE ANALYSIS OF PETSC-FUN3D

A full scale PDE code has several computational phases (mostly expressed as some loop in a subroutine). Each phase may suffer from memory bandwidth limitation or operation issue limitation or both. When a computational phase has very modest memory bandwidth requirement, prefetching should be done to improve the performance. Many of the processors have direct hardware and software support for prefetching. If the processor has lot of non-floating point work to do, it will suffer from an instruction scheduling limitation.

In this section, we study two important computational phases (responsible for approximately 75% of the overall execution time) of PETSc-FUN3D and attempt to evaluate the performance bounds for both of these.

Triangular Matrix Solver Phase

In this phase, the problem $Ax = b$ is solved for a factored matrix (for PETSc-FUN3D it is the preconditioner matrix). First, a forward solve is done with the lower triangular factor followed by a backward solve with the upper triangular factor. In principle, the backward and forward solves are similar to the matrix-vector product kernel discussed in the previous section. The bounds arising from available memory bandwidth and instruction scheduling stay the same. As is observed for the matrix vector multiplication, it is the memory bandwidth that limits the performance in this phase of PETSc-FUN3D.

Flux Calculation

This flux calculation phase is the heart of any unstructured mesh solver and accounts for over 50% of the overall execution time in PETSc-FUN3D. Since PETSc-FUN3D is vertex-centered code, the flow variables are stored at nodes. While making a pass over an edge, the flow variables from the vertex based arrays are read, a large amount of floating point work is done, and finally, the residual values at each vertex of the edge are updated (see Figure 9). A close look at the assembly code yields the following mix of the workload for each iteration of the loop over edges:

- 519 total instructions
- 111 integer operations
- 250 floating point instructions doing 305 flops (there are 55 fmadd instructions)
- 155 memory references

This part of the code is unique (relative to the rest) in the sense that it does many floating point operations on the data items that are loaded into the cache. Our estimate and measurements using hardware counters on the R10000 processor show that this phase needs a memory bandwidth of

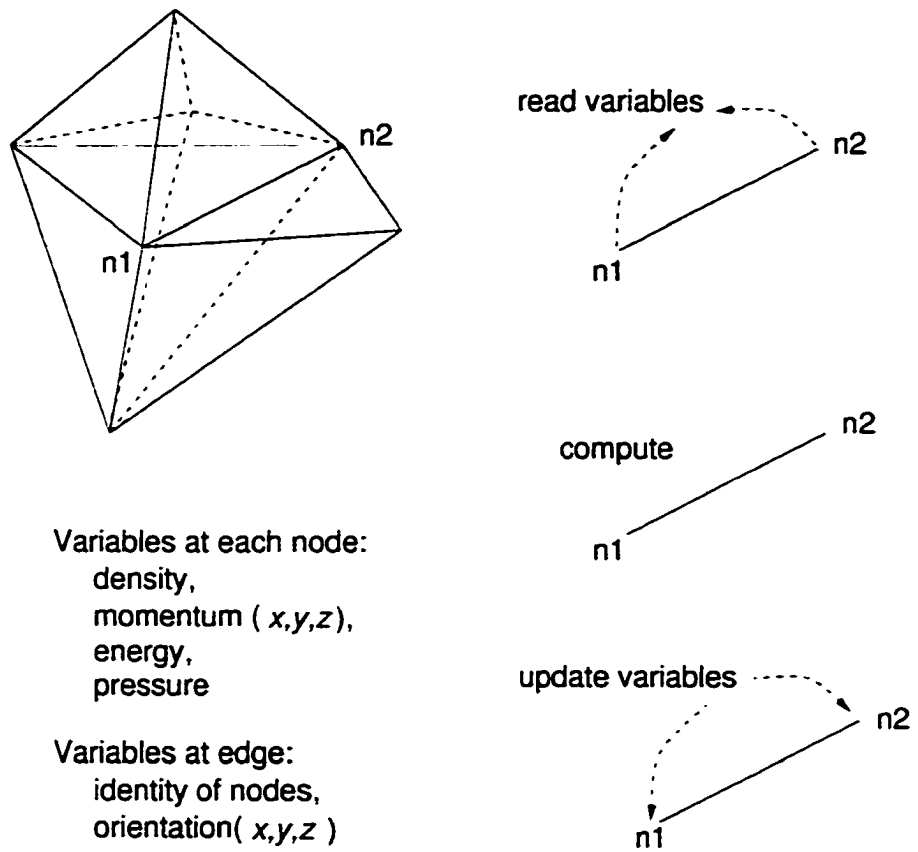


Figure 9: Illustration of flux calculation in PETSc-FUN3D. In each iteration of the loop over edges, solution variables stored at the vertices are read, large amount of floating point work is done, and then residual values are stored back at the vertices.

only 57 MB/s (which is a very modest requirement as compared to what is available on most modern processors). Hence this phase does not suffer from memory bandwidth limitation.

In order to evaluate the instruction scheduling limitation, we need to schedule these instructions in the best possible way on modern superscalar pipelined processors. This is not an easy task, especially when we need to fill out the slots for high latency instructions (like sqrt, div etc.) and observe various other constraints (e.g. [47, 42]). Fortunately, many compilers provide this information as comments in the assembly code. For example, on SGI Origin2000, when we compile the code with cache optimizations turned off (consistent with our assumption of data items being available in primary cache for the purpose of estimating this bound), the compiler estimates that the above work can be done in about 325 cycles. This leads to a performance bound of 235 Mflops/s (47% of the peak on 250 MHz processor). We actually measure 209 Mflops/s using hardware counters. This shows that the performance in this phase of the computation is actually restricted by the instruction scheduling limitation.

IV.4 PERFORMANCE TUNING FOR PETSC-FUN3D

In this section, we describe the details of the performance tuning process. Our approach is largely experimental, guided in part by the performance models that we have developed in [22]. We present the data layouts that can reduce the number of cache misses, tend to use low memory bandwidth, and show better scalability, especially when the number of subdomains becomes large.

Reducing the Cache Misses

Since the gap between memory and CPU speeds is ever widening [30], it is crucial to 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 (discussed below) have proved very useful in improving the performance of the FUN3D code, which was originally tuned for vector machines.

Interlacing

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

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

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 translation look-aside buffer (TLB – a small cache that translates virtual address into the physical address of the pages in memory) misses [30], and (3) it decreases the size of the working set of the data cache(s), which reduces the number of conflict misses.

We illustrate these benefits for the sparse matrix-vector product (discussed earlier in this Chapter). We assume that the matrix of N rows is stored in CSR or compressed sparse row format. Although the matrix is sparse, the vector it multiplies is dense. We analyze only the data cache that is closest to the main memory (e.g., secondary or L2 cache, in case of a two-level cache hierarchy).

In the noninterlaced case, the resulting matrix is of very wide bandwidth close to N. This makes the working set of the matrix-vector product operation in the cache close to N/W_{sc}, where

W_{sc} is the cache block size in double words. If C_{sc} is the cache capacity in double words, then the number of conflict cache misses is bounded by

$$N \left\lceil \frac{N - C_{sc}}{W_{sc}} \right\rceil \tag{12}$$

when $N \geq C_{sc}$.

In the interlaced case, the unknowns at a grid point are stored together. With a good node reordering strategy, the matrix resulting out of some discretization of a PDE can be made to have a narrow bandwidth, β, which is much smaller than N. This results in the fewer conflict misses, bounded by the following expression:

$$N \left\lceil \frac{\beta - C_{sc}}{W_{sc}} \right\rceil \tag{13}$$

when $\beta \geq C_{sc}$.

We can derive similar expressions for the bounds (12 and 13) on TLB misses where C_{sc} will be replaced by the number of page table entries (PTE), C_{TLB} and W_{sc} by the memory page size, W_{mem}. Since the interlaced storage works on the data items closely spaced in memory, it causes fewer TLB misses as compared to those in the noninterlaced case.

Structural Blocking

Once the field data is 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 multiplication case earlier (also see [22]), the structural blocking significantly reduces the number of integer loads and enhances the reuse of the data items in registers (see Figures 11 and 12; especially notice the effect of blocking on the graduated loads and stores per floating point operation). This, in turn, reduces the required memory bandwidth for optimal performance.

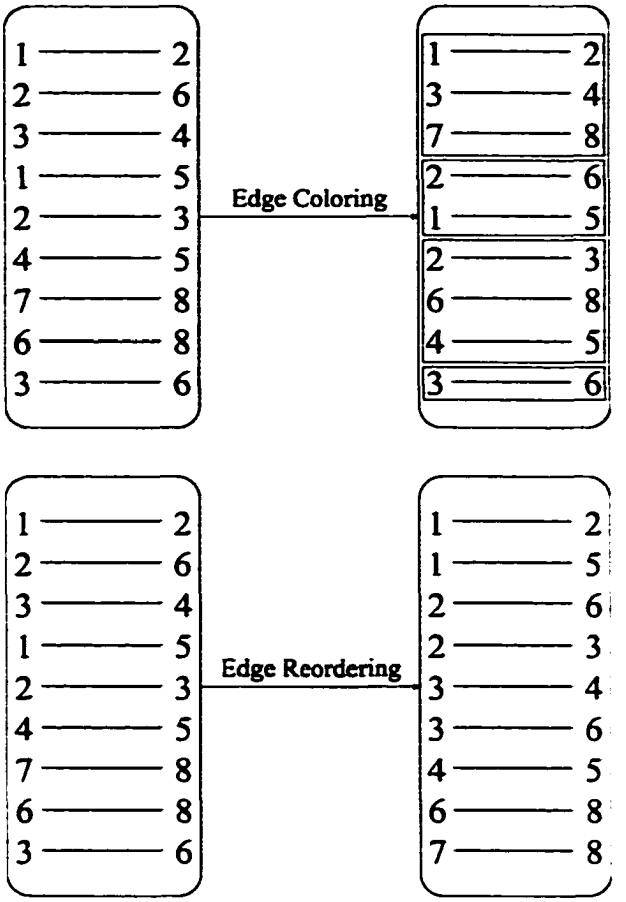


Figure 10: Illustration of edge coloring (**top**) and reordering (**bottom**). Each box, after edge coloring, represents a color.

Edge and Node Reorderings

In the original FUN3D code, the edges are colored (see Figure 10) for good vector performance. No pair of nodes in the same discretization stencil shares a color. This 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 (of the original code) is spent servicing TLB misses. As shown in Figure 11, this problem is effectively addressed by reordering the edges.

The edge reordering we have used is obtained by sorting 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. And since a loop over edges goes over a node's neighbors first, this (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) [16], which is known for reducing cache misses by creating more spatial locality. This ordering also results in reduced bandwidth of the preconditioner matrix, which in turn decreases the size of working set of the sparse matrix vector product operation.

IV.5 SAMPLE SEQUENTIAL PERFORMANCE

Table 2 shows the effectiveness of these techniques (interlacing, blocking, and edge reordering) on one processor of the SGI Origin2000. 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 [34]). To further understand these results, we carried out hardware counter profiling on a R10000 processor. Figures 11 and 12 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.

Enhancements			Results			
Field	Structural	Edge	Incompressible		Compressible	
Interlacing	Blocking	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

Table 2: Effect of cache optimization techniques on the sequential performance of PETSc-FUN3D code on SGI Origin2000. The execution times are for Euler flow over an M6 wing with a grid of 22,677 vertices (90,708 DOFs incompressible; 113,385 DOFs compressible). The SGI Origin2000 (MIPS R10000) processor has 250 MHz clock and cache sizes of 32 KB data, 32 KB instruction, and 4 MB L2. Activation of a layout enhancement is indicated by "×" in the corresponding column. Improvement ratios are averages over the entire code; different subroutines benefit to different degrees.

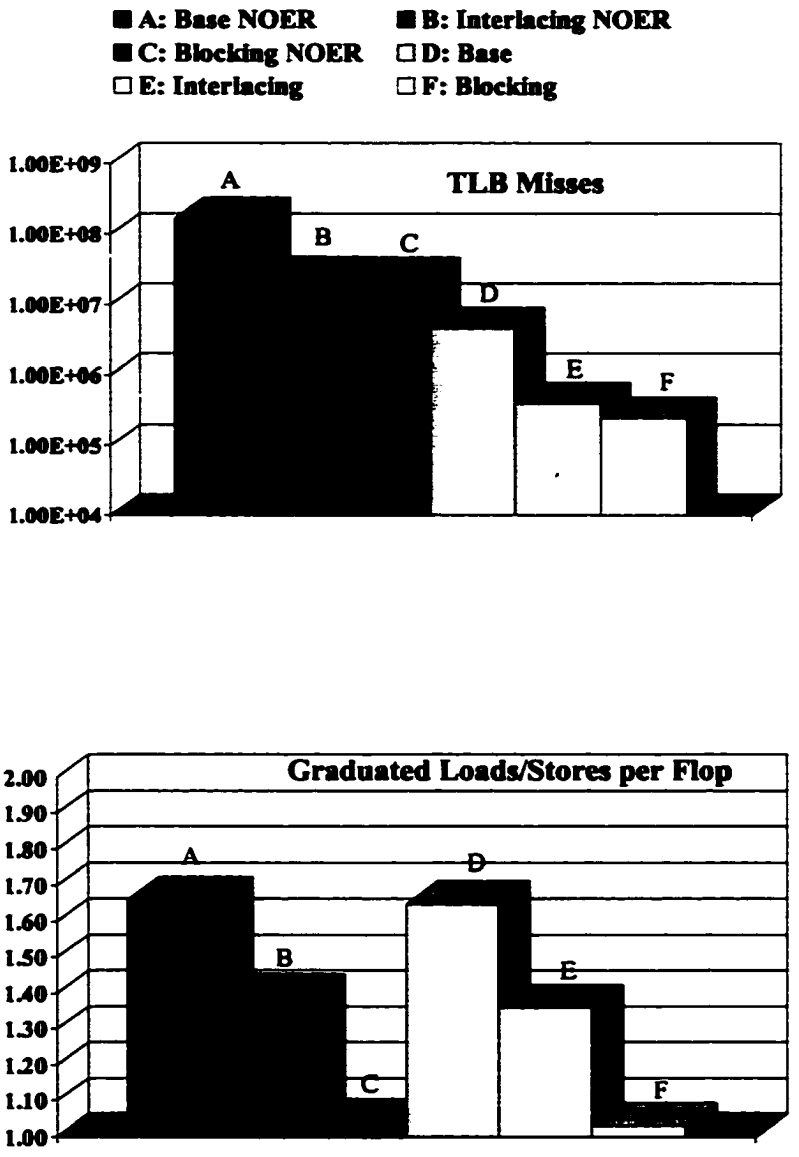


Figure 11: Measured values of TLB misses and memory references per floating point operation on one processor of an SGI Origin2000 for a 22,677 vertex mesh using hardware counters. **Top:** TLB misses (log scale). **Bottom :** Graduated (completed) loads and stores per floating point instruction. "NOER" denotes *no* edge ordering, otherwise edges are reordered by default.

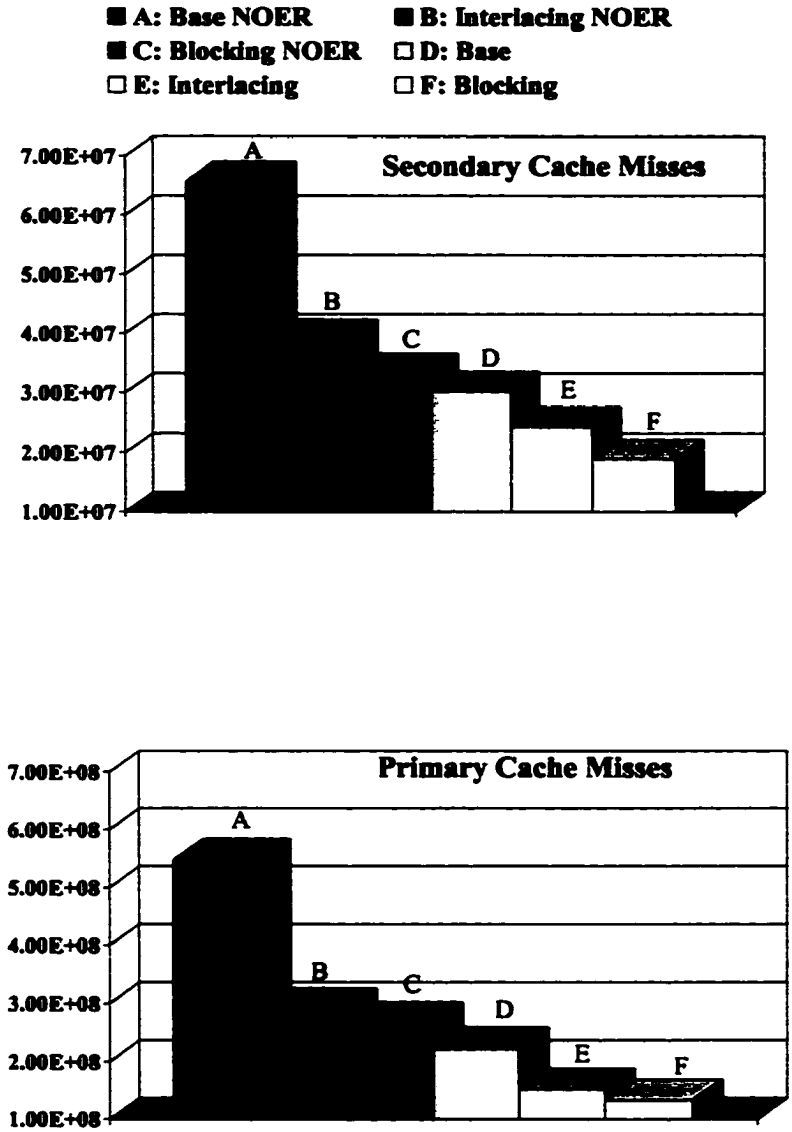


Figure 12: Measured values of primary and secondary cache misses floating point operation on one processor of an Origin2000 for a 22,677 vertex mesh using hardware counters. **Top:** Secondary cache misses. **Bottom :** Primary cache misses. "NOER" denotes *no* edge ordering, otherwise edges are reordered by default.

CHAPTER V

PARALLEL PERFORMANCE STUDIES AND RELATED ISSUES

Achieving high sustained performance, in terms of solutions per second, requires attention to three factors. The first is good per-processor performance on contemporary cache-based microprocessors, which was discussed in Chapter IV. The second 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 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 since the requirement of a scalable implementation generally forces parameterized changes in the algorithm as the number of processors grows. However, if the convergence is allowed to degrade, the overall execution is not scalable, and this must be countered algorithmically. Having devoted Chapter IV to per-node performance, we only consider the last two factors in the overall performance in Sections V.4 and V.5, respectively.

V.1 DEFINITION OF PARALLEL SCALABILITY

Conflicting definitions of parallel efficiency abound, depending upon two choices:

- What scaling is to be used as the number of processors is varied [15]?
  - problem-constrained scaling (fixed overall problem size)
  - memory-constrained scaling (varying size problem with fixed memory per processor)
  - time-constrained scaling (varying size problem with fixed work per processor)
- What form of the algorithm is to be used as number of processor is varied?

- reproduce the sequential arithmetic exactly (this assumes that floating point operations are associative)
- adjust parameters to perform best on each given number of processors

In this work, we have considered the problem-constrained scaling with the following definition of relative parallel efficiency in going from q to p processors ($p > q$)

$$\eta(p|q) = \frac{q \cdot T(q)}{p \cdot T(p)},$$

where $T(p)$ is the overall execution time on p processors (directly measurable). Factoring $T(p)$ into $I(p)$, the number of iterations, and $C(p)$, the average cost per iteration, the algorithmic efficiency is an indicator of preconditioning quality (directly measurable):

$$\eta_{alg}(p|q) = \frac{I(q)}{I(p)}.$$

Implementation efficiency is the remaining (inferred) factor:

$$\eta_{impl}(p|q) = \frac{q \cdot C(q)}{p \cdot C(p)}.$$

In our implementations of the ΨNKS algorithm, we always adjust the subdomain blocking parameter to match the number of MPI processes, one subdomain per process; this causes the number of iterations to vary, especially since our subdomain partitionings are not nested. The base case (with q processes) is always chosen when the local problem fits into the main memory of each processor. In our experience, the choice of a proper base case often eliminates spurious superlinear speedups that are often seen in problem-constrained scalability studies on small number of processors.

V.2 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, communica-

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

PETSc uses the best timers available 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 the data, 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 just fits into the local memory. We have employed smaller sequential cases to optimize cached data reuse [33, 34] to minimize the execution time. As already stated, 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.

V.3 LARGE-SCALE DEMONSTRATION RUNS

A fixed-size problem (with 2.8 million vertices and 11 million degrees of freedom) is run on large ASCI Red configurations (about 3200 nodes, each with dual Pentium Pro 333 MHz processors) with sample scaling results shown in Figure 13. The implementation efficiency is 91% in going from 256 to 3072 nodes. The preconditioner used in these results is block Jacobi with ILU(0) in each subdomain. However, we have now discovered that the block Jacobi with ILU(1) gives better

execution times (see Table 8). Also the preconditioner matrix was stored in double precision in these results. We show in Section V.7 that single precision storage gives better performance.

For the data in Figure 13, 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 (and 262 when the preconditioner is stored in single precision) . The `-procs 3` option enables one to run two MPI processes on a node (one on each processor) Using these two options (`-proc 2 and -procs 3`, we were able to compare the message passing and hybrid programming model that we discuss in Section V.8.

Figure 14 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 1,000 over the range shown. We point out that for just 1,000 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).

Scalability Across Flow Regimes

Trans-Mach convergence comparisons on the GridB mesh problem are given in Tables 3 and 4. Here efficiencies are normalized by the number of time steps, to factor convergence degradation out of the performance picture and measure implementation factors alone (though convergence

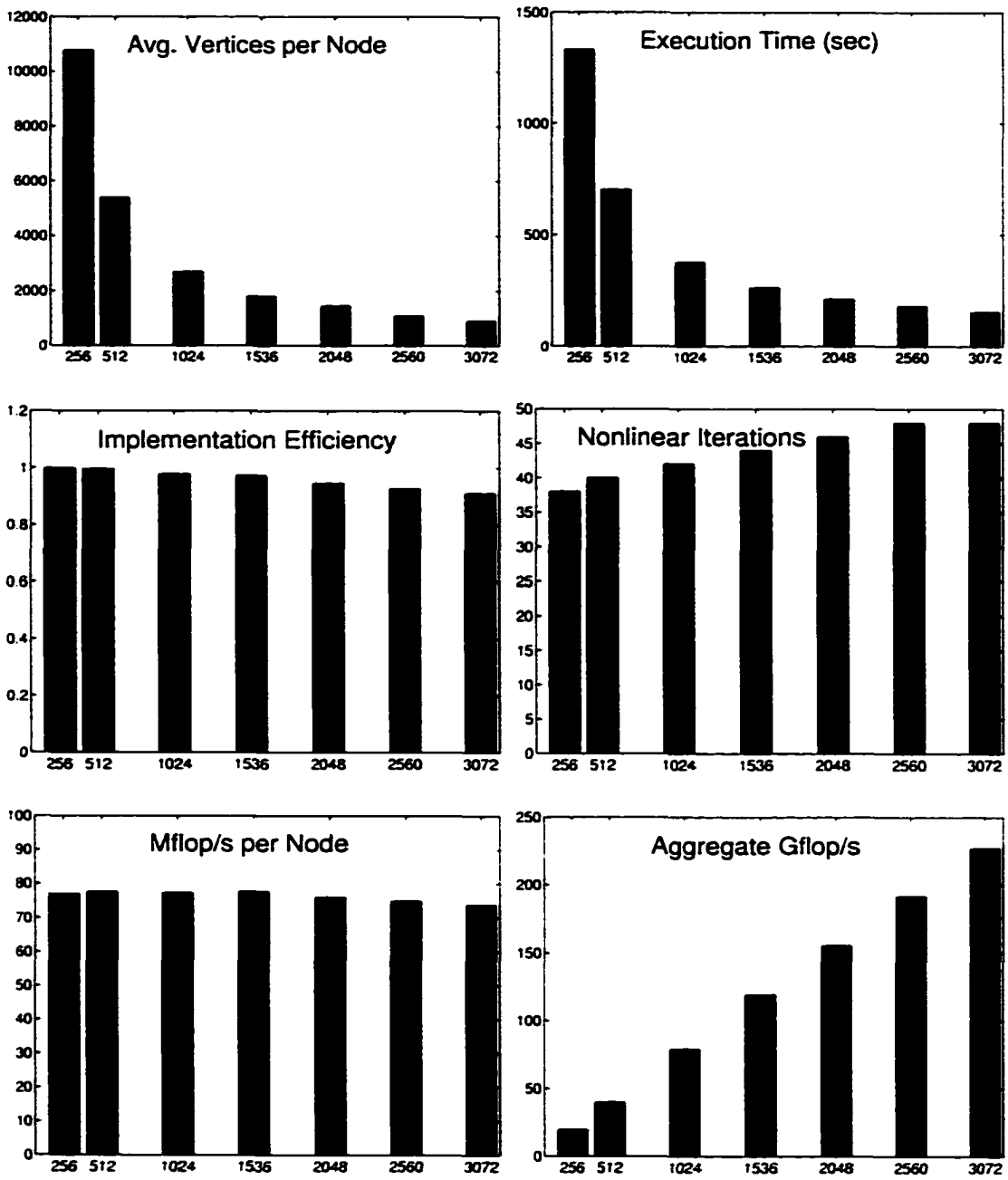


Figure 13: Average vertices of the mesh owned by each processor and five parallel performance metrics for a fixed-size problem on a 2.8 million vertex mesh, run on up to 3072 nodes of ASCI Red (each node consisting of two 333 MHz Pentium Pro processors).

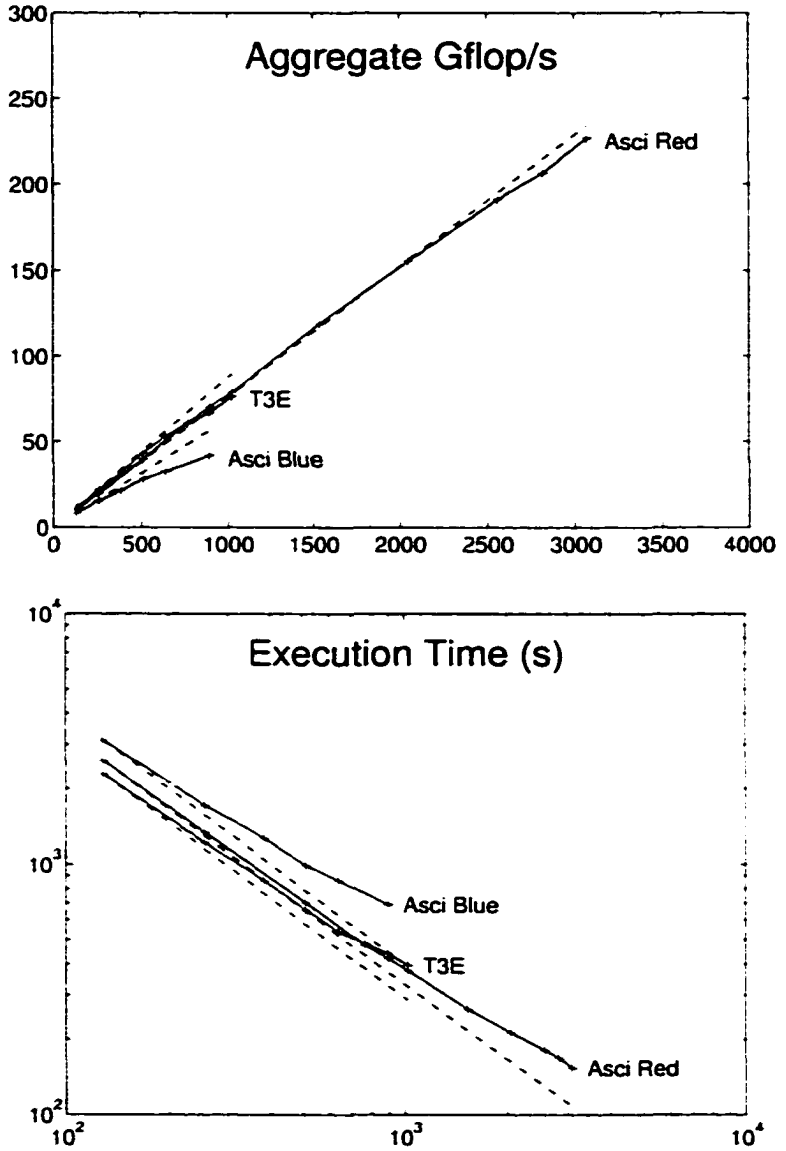


Figure 14: Gigaflop/s ratings and execution times on ASCI Red (up to 3072 dual processor nodes), ASCI Pacific Blue (up to 768 processors), and a Cray T3E (up to 1024 processors) for a 2.8M-vertex case, along with dashed lines indicating "perfect" scalings.

degradation with increasing granularity is modest). The number of steps increases dramatically with the nonlinearity of the flow, as the Mach number rises; however, the linear work per step decreases on average. Reasons for this include: more steps spent in the cheaper, first-order discretization phase of the continuation process, smaller CFL in early steps, and the increased hyperbolicity of the flow. The compressible Jacobian is far more complex to evaluate, but it also concentrates locality, achieving much higher computational rates than the corresponding incompressible Jacobian.

V.4 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 [37], 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 Ethernet 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.

No. Procs.	Steps	Time per Step	Per-Step Speedup	Impl. Eff.	FcnEval Mflop/s	JacEval Mflop/s
Incompressible (4 × 4 blocks)						
16	19	41.6s	—	—	2,630	359
32	19	20.3s	2.05	1.02	5,366	736
48	21	14.1s	2.95	0.98	7,938	1,080
64	21	11.2s	3.71	0.93	10,545	1,398
80	21	10.1s	4.13	0.83	11,661	1,592
Subsonic (Mach 0.30) (5 × 5 blocks)						
16	17	55.4s	—	—	2,002	2,698
32	19	29.8s	1.86	0.93	3,921	5,214
48	19	20.5s	2.71	0.90	5,879	7,770
64	20	14.3s	3.88	0.97	8,180	10,743
80	20	12.7s	4.36	0.87	9,452	12,485

Table 3: Parallel scalability across flow regimes – incompressible and subsonic flow over M6 wing on SGI Origin2000 with a fixed-size grid of 357,900 vertices (1,431,600 DOFs incompressible, 1,789,500 DOFs compressible).

No. Procs.	Steps	Time per Step	Per-Step Speedup	Impl. Eff.	FcnEval Mflop/s	JacEval Mflop/s
Transonic (Mach 0.84) (5 × 5 blocks)						
16	55	29.4s	—	—	2,009	2,736
32	56	15.4s	1.91	0.95	4,145	5,437
48	56	11.0s	2.66	0.89	5,942	7,961
64	57	8.7s	3.39	0.85	8,103	10,531
80	57	7.4s	3.99	0.80	9,856	12,774
Supersonic (Mach 1.20) (5 × 5 blocks)						
16	80	19.2s	—	—	2,025	2,679
32	81	10.6s	1.81	0.90	3,906	5,275
48	81	7.1s	2.72	0.91	6,140	7,961
64	82	5.8s	3.31	0.83	7,957	10,398
80	80	4.6s	4.20	0.84	9,940	12,889

Table 4: Parallel scalability across flow regimes – transonic and supersonic flow over M6 wing on SGI Origin2000 with a fixed-size grid of 357,900 vertices and 1,789,500 DOFs.

If we assume that the load is perfectly balanced and the network is scalable, then 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.

Scalability Bottlenecks

In Table 5, 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 (only one processor from each node is used for these results). The intent here is to identify the factors that retard scalability. The overall parallel efficiency (denoted by $\eta_{overall}$) is broken into two components: η_{alg} measures the degradation in the parallel efficiency due to the increased iteration count (Section V.5) of this (non-coarse-grid-enhanced) NKS algorithm as the number of subdomains increases, while η_{impl} measures the degradation coming from all other nonscalable factors such as global reductions, load imbalance (implicit synchronizations), and hardware limitations.

From Table 5, we observe that the buffer-to-buffer time for global reductions for these runs is relatively small and does not grow on ASCI Red's 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% to 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 3.6 gigabytes, while for 3072 subdomains it is 14.2 gigabytes. Although more

Number of Processors	Its	Time	Speedup	Efficiency		
				$\eta_{overall}$	η_{alg}	η_{impl}
128	22	2,039s	1.00	1.00	1.00	1.00
256	24	1,144s	1.78	0.89	0.92	0.97
512	26	638s	3.20	0.80	0.85	0.94
1024	29	362s	5.63	0.70	0.76	0.93
2048	32	208s	9.78	0.61	0.69	0.89
3072	34	159s	12.81	0.53	0.65	0.82

Number of Processors	Percent Times for			Scatter Scalability	
	Global Reductions	Implicit Synchronizations	Ghost Point Scatters	Total Data Sent per Iteration (GB)	Application Level Effective Bandwidth per Node (MB/s)
128	5	4	3	3.6	6.9
256	3	6	4	5.0	7.5
512	3	7	5	7.1	6.0
1024	3	10	6	9.4	7.5
2048	3	11	8	11.7	5.7
3072	5	14	10	14.2	4.6

Table 5: Scalability bottlenecks for large scale runs on ASCI Red. The mesh employed here has 2.8 million vertices and 19 million edges. 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.

network wires are available when more processors are employed, scatter time increases. If problem size and 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.

The final column in Table 5 shows the scalability of the "application level effective bandwidth" that is computed by dividing the total amount of data transferred by the time spent in scatter operation. It includes the message packing and unpacking times plus any contention in the communication. That is why it is far lower than the achievable bandwidth (as measured by the "Ping-Pong" test from the message passing performance (MPP) [25] tests) of the networking hardware. The Ping-Pong test measures the point to point unidirectional bandwidth between any two processors in a communicator group. It is clear that the Ping-Pong test results in Table 6 are not representative of the actual communication pattern encountered in the scatter operation. To better understand this issue, we have carried out the "Halo" test (from the MPP test suite) on 64 nodes of the ASCI Red machine. In this test, a processor exchanges messages with a fixed number of neighbors, moving data from/to contiguous memory buffers. For the Halo test results in Table 6, each node communicated with 8 other nodes (which is a good estimate of the neighbors a processor in PETSc-FUN3D will need to communicate). The message lengths for both these tests (Ping-Pong and Halo) have been varied between 2KB to 32 KB since the average length of a message in the runs for Table 5 decreases from 23 KB to 3 KB as the number of processor goes up from 128 to 3072. We observe that the bandwidth obtained in the Halo test is significantly less than that obtained in the Ping-Pong test. This loss in performance perhaps can be attributed to the fact that a processor communicates with more than one neighbor at the same time in the Halo test. In addition, as stated earlier, the scatter operation involves the overhead of packing and unpacking of messages at the rate limited by the achievable memory bandwidth (about 145 MB/s as measured by the STREAM benchmark [41]).

Message Length, KB	Bandwidth, MB/s	
	Ping-Pong	Halo
2	93	70
4	145	94
8	183	92
16	235	106
32	274	114

Table 6: MPP test results on 64 nodes of ASCI Red. The Ping-Pong results measure the unidirectional bandwidth. The Halo test results (measuring the bidirectional bandwidth) are more representative of the communication pattern encountered in the scatter operation.

V.5 CONVERGENCE SCALABILITY

The convergence rates and, therefore, the overall parallel efficiencies of additive Schwarz methods are often dependent on subdomain granularity (see Table 7). 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.

Convergence of Schwarz Methods

For a general exposition of Schwarz methods for linear problems, see [48]. 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 7 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-

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

Table 7: 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.

energy-norm solution near discontinuities at subdomain boundaries. The 2-level method projects out low-wavenumber 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 $O(P)$. Unlike recursive multilevel methods, a 2-level Schwarz method may have a rather fine coarse grid, for example, $H = O(h^{1/2})$, 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.

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, expo-

nent of timestep evolution law

- Newton parameters: convergence tolerance on each time step, 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

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 [43]. To be specific, within each residual reduction phase of computation, we adjust the timestep according to

$$N_{CFL}^{\ell} = N_{CFL}^{0} \left(\frac{\|f(u^0)\|}{\|f(u^{\ell-1})\|} \right)^p .$$

where ℓ is the current time step number, $\|f(u^{\ell-1})\|$ is 2-norm of the residual vector at the previous time step $(\ell - 1)$, and p is a tunable exponent close to unity. Figure 15 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 order. This order of accuracy applies to the flux calculation; the preconditioner matrix is always built out of a first-order analytical Jacobian matrix.

Parameters for Krylov Solver

We use an inexact Newton method on each timestep [17]; 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

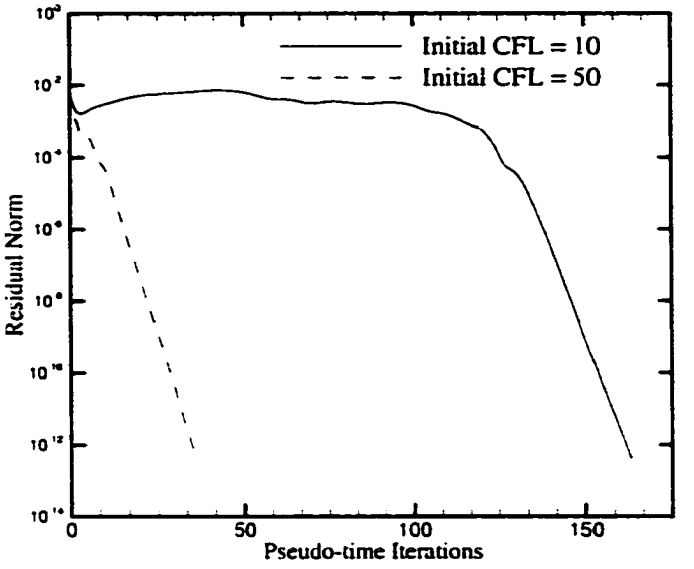


Figure 15: Residual norm versus iteration count for a 2.8M-vertex case, showing the effect of initial CFL number on convergence rate. The convergence tuning of nonlinear problems is notoriously case specific.

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. Several thousand fine-grid flux evaluations are typically required to achieve 10^{-10} residual reduction on a million-vertex Euler problem.

Additive Schwarz Preconditioner

Table 8 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 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 8 we see that the larger overlap and more fill helps 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 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) [14], 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 8 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.

Other Algorithmic Tuning Parameters

In [24] 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 [31] 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 or the parallel efficiency. There are many tradeoffs that enhance Mflop/s rates or efficiency but retard execution completion.

V.6 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-

ILU(0) in Each Subdomain

Number of Processors	Overlap 0 Time	Overlap 0 Linear Its	Overlap 1 Time	Overlap 1 Linear Its	Overlap 2 Time	Overlap 2 Linear Its
32	688s	930	***661s***	816	696s	813
64	***371s***	993	374s	876	418s	887
128	***210s***	1052	230s	988	222s	872

ILU(1) in Each Subdomain

Number of Processors	Overlap 0 Time	Overlap 0 Linear Its	Overlap 1 Time	Overlap 1 Linear Its	Overlap 2 Time	Overlap 2 Linear Its
32	598s	674	***564s***	549	617s	532
64	***334s***	746	335s	617	359s	551
128	***177s***	807	178s	630	200s	555

ILU(2) in Each Subdomain

Number of Processors	Overlap 0 Time	Overlap 0 Linear Its	Overlap 1 Time	Overlap 1 Linear Its	Overlap 2 Time	Overlap 2 Linear Its
32	***688s***	527	786s	441	—	—
64	***386s***	608	441s	488	531s	448
128	***193s***	631	272s	540	313s	472

Table 8: 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. The best execution times for each ILU fill level and number of processors are in ***boldface italics*** in each row.

MeTiS algorithm in the MeTiS package [32], 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. Figure 16 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 V.5.

V.7 REDUCING THE REQUIRED MEMORY BANDWIDTH

The CFD application spends almost all of its time in two phases: flux computations, to evaluate conservation law residuals, and sparse linear algebraic kernels, to solve the Newton equations for an iterative correction to the solution. The linear algebraic kernels run at close to the aggregate memory-bandwidth limit on performance (as determined by the STREAM benchmarks [41]), and the flux computations are bounded by instruction scheduling, that is, the number of basic operations that can be performed in a single clock cycle (see the analysis in [22]).

To improve the performance of the sparse triangular matrix solution phase (and of other similar phases where memory bandwidth is a bottleneck), we store elements of the preconditioner for the Jacobian matrix in single precision. In our "matrix-free" implementation, the Jacobian itself is never explicitly needed; see [28]. All computation with the preconditioner is still done in full

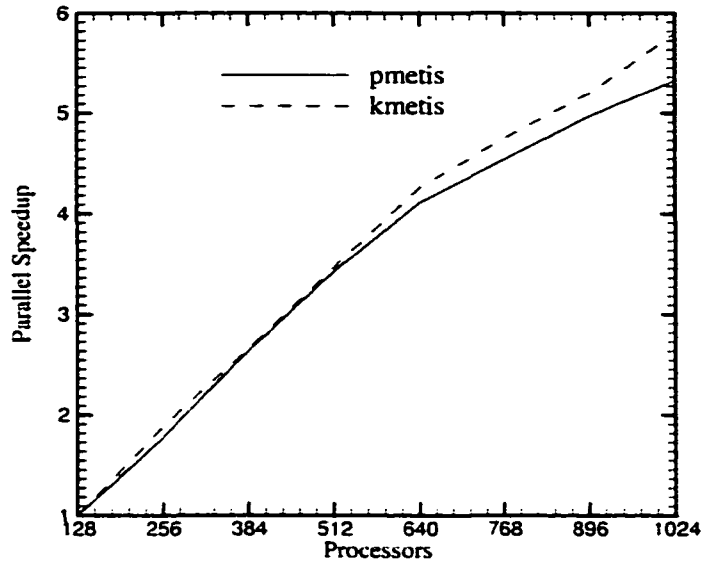


Figure 16: Parallel speedup relative to 128 processors on a 600 MHz Cray T3E for a 2.8M vertex case, showing the effect of partitioning algorithms k-MeTiS, and p-MeTiS.

(double) precision. The performance advantages are shown in Table 9, where the single precision storage version runs at almost twice the rate of the double precision storage version, clearly identifying memory bandwidth as the bottleneck. The percentage of overall time in the linear solver ranges from 29.9% to 25.4% for the double precision storage version to 20.7% to 15.1% for the single. The number of time steps needed to converge is not affected, since the preconditioner is already very approximate by design.

V.8 DOMAIN-BASED AND/OR INSTRUCTION-LEVEL PARALLELISM

The performance results above are based on subdomain parallelism using the Message Passing Interface (MPI) [26]. With the availability of large-scale SMP clusters, the different software models for parallel programming require a fresh assessment. For physically distributed memory

Number of Processors	Computational Phase			
	Linear Solve		Overall	
	Double	Single	Double	Single
16	223s	136s	746s	657s
32	117s	67s	373s	331s
64	60s	34s	205s	181s
120	31s	16s	122s	106s

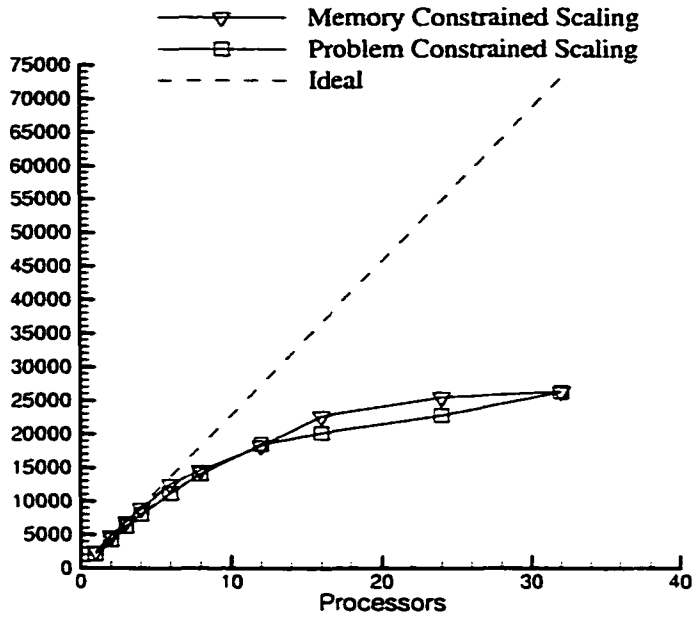
Table 9: Execution times on a 250 MHz Origin 2000 for 357,900 vertex case with single or double precision storage of the preconditioner matrix. The results suggest that the linear solver time is bottlenecked by memory bandwidth. This conclusion is supported by analytical estimates in [22].

machines, the message passing interface (MPI) has been a natural and very successful software model. For another category of machines with distributed shared memory and nonuniform memory access, both MPI and OpenMP have been used with respectable parallel scalability. However, for clusters with two or more SMPs on a single node, the hybrid programming model with 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 among the nodes seems natural. Several researchers (e.g., [10, 40]) have used this mixed model with reasonable success.

Two extremes of execution on hybrid architectures are often employed, due to their programming simplicity. At one extreme is the scenario in which the user explicitly manages the memory updates among different processes; this is typically done by using MPI, but can also be implemented with OpenMP. The advantage of this approach is good performance and excellent scalability since network transactions can be performed at large granularity. Even when the user explicitly

manages the memory updates, OpenMP can potentially offer the benefit of lower communication latencies by avoiding some extraneous copies and synchronizations. The other extreme is the case in which the system manages updates among different threads (or processes), e.g., the shared memory model with OpenMP. Here the term "system" refers to the hardware or the operating system, but most commonly a combination of the two. The advantages are the ease of programming, possibly lower communication overhead, and no unnecessary copies. However, performance and scalability are open issues. For example, the user may need to employ a technique such as coloring to create nonoverlapping units of work to get reasonable performance. In the hybrid programming model, some updates are managed by the user (e.g., via MPI or OpenMP) and the rest by the system (e.g., via OpenMP).

In this section, we evaluate the hybrid programming model using memory performance as a metric. The performance of many scientific computing codes is dependent on the performance of the memory subsystem, including the available memory bandwidth, memory latency, number and sizes of caches, etc. In addition, scheduling of memory transactions can also play a large role in the performance of a code. Ideally, the load/store instructions should be issued as early as possible. However, because of hardware (number of load/store units) or software (poor quality assembly code) limitations, these instructions may be issued significantly late, when it is not possible to cover their high latency, resulting in poor overall performance. OpenMP has the potential of better memory subsystem performance since it can schedule the threads for better cache locality or hide the latency of a cache miss. However, if memory bandwidth is the critical resource, extra threads only compete with each other, actually degrading performance relative to one thread (see Figure 17).



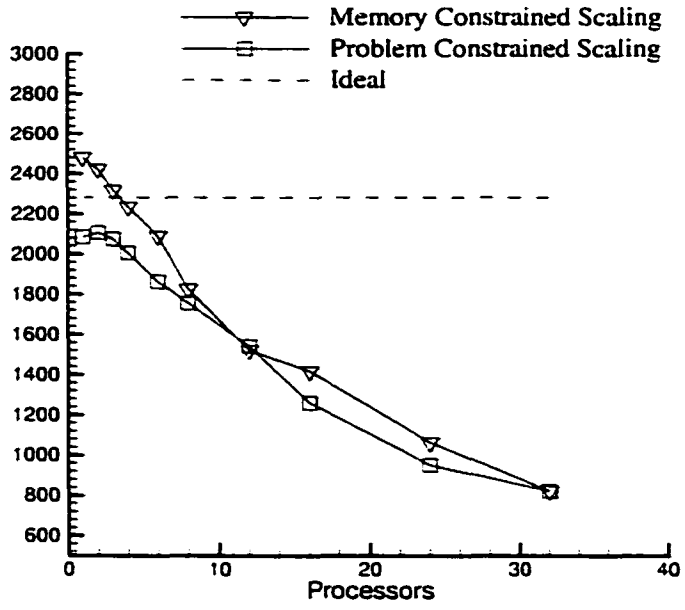


Figure 17: Memory bandwidth (MBytes/sec) as measured by STREAM benchmark [41] on one node of IBM Power 4 (32 processors per node each running at 1.3 GHz clock). **Top:** Total bandwidth. **Bottom:** Bandwidth per processor.

Implementation Issues

While implementing the hybrid model, the following issues should be considered.

- **False sharing**: this happens when two (or more) threads running on different processors work on different words in the same cache line. If thread 1 writes a word (thus invalidating the cache line), an additional cache miss occurs on the other processor (running thread 2) even if it does not need the word written by thread 1 [15].
- **Cache locality**: a thread should be able to reuse a cache line multiple times before it gets invalidated or replaced by another cache line.
- **Work division among threads**: work can be divided by compiler or manually (as is done for pure message passing case); also this division can be static or dynamic.
- **Updates Management**: there are many possibilities for managing the updates
  - Private data — initialization and reductions are memory bandwidth limited
  - Shared data — updates need to be synchronized
  - Data decomposition — a shared array is divided into different pieces; a thread is assigned to one or more pieces with exclusive write permission (other threads can only read those pieces).

Hybrid Model in PETSc-FUN3D

We investigate the mixed model by employing OpenMP in the flux calculation phase only (Figure 9 in Chapter IV). 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 IV.3). Three different implementations of the hybrid model are considered:

- **Edge Coloring**: see left side in Figure 10 for an illustration of edge coloring; all the threads can work simultaneously in the same color. The work among threads is divided by compiler (using OpenMP directives). We observe that this arrangement leads to poor cache locality but updates are independent.

- **Edge Reordering with redundant work**: see right side in Figure 10 for an example of edge reordering. Here updates to the shared data are managed through private work arrays for each thread. Each thread writes to its private work arrays that are gathered into the shared arrays during a sequential reduction phase (which introduces some redundant work). Again work is divided by compiler (using OpenMP directives). This case has excellent cache locality but updates are sequential.

- **MeTiS divided work for each thread**: here each MPI process calls MeTiS to further subdivide the work among threads; the boundary data is replicated for each thread. We apply the "Owner computes" rule for every thread to resolve the updates (similar to pure message passing case). This case has reasonable cache locality (which can be enhanced by suitable vertex and edge reorderings).

In Table 10, we compare the performance of flux evaluation and replication of "ghostpoint" data (VecScatter) phases 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 the flux evaluation phase.

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 but 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 is 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 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 some 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.

Flux Evaluation Time in seconds					
	MPI Processes per Node		MPI/OpenMP with 2 Threads per Node		
Nodes	1	2	Edge Coloring	Edge Reordering	MeTiS Division
256	451	258	352	255	231
512	249	153	172	137	127
1024	148	88	89	83	76
2048	85	48	47	45	44
3072	61	40	33	32	33

Vector Scatter Time in seconds					
	MPI Processes per Node		MPI/OpenMP with 2 Threads per Node		
Nodes	1	2	Edge Coloring	Edge Reordering	MeTiS Division
256	59	74	71	59	62
512	38	58	48	38	39
1024	35	48	41	33	33
2048	37	51	38	39	38
3072	32	51	30	30	30

Table 10: Execution time on the 333 MHz Pentium Pro ASCI Red machine for function evaluations only for a 2.8M-vertex case, comparing the performance of the hybrid (MPI/OpenMP) and the distributed memory (MPI alone) programming models.

CHAPTER VI

CONCLUSIONS

This work demonstrates software reuse of a legacy PDE code through the integration of the NASA code FUN3D into the PETSc framework. This new code (PETSc-FUN3D) achieves high levels of performance on many large-scale machines (ASCI Red, Cray T3E, SGI Origin, etc.). We employ the pseudo-transient continuation Newton-Krylov-Schwarz (ΨNKS) algorithm to solve the Euler (compressible and incompressible) equations of fluid flow in parallel. We analyze the performance of this code with a nontraditional emphasis on memory references, in addition to the traditional focus on minimizing the floating point work. We take the view that the floating point work is almost free because it gets overlapped with cycles spent while fetching operands from memory on modern superscalar architectures. On the other hand, because of large scale, we must employ algorithms whose convergence rate is nearly optimal.

To understand why many PDE based codes achieve only a small fraction of peak performance, we pay attention to the memory hierarchy of a processor. Using PETSc-FUN3D as an example, we demonstrate that interlacing and blocking of field variables and edge reordering are very effective in dealing with the limitations arising from memory bandwidth and instruction scheduling. Interlacing reduces the size of the working set at any level of cache by making the memory references closely spaced. Blocking reduces the memory bandwidth requirement by cutting the number of loads. Our experimental results on SGI Origin 2000 show that *blocking* graduates more loads and stores per floating point instruction. Edge reordering creates more temporal locality in memory references and cuts down significantly the TLB misses and data cache misses.

The impact of these data reorganizing strategies is further supported by providing simple performance models for a critical kernel of scientific computing, the sparse matrix-vector (MATVEC) product operation. The *cache conflict analysis* for banded matrix vector product case helps us to

understand how to minimize the cache misses by using appropriate vector size and padding values. While carrying out performance optimizations, it is important to know what is the "achievable" performance for a particular algorithm on a given architecture. The performance models developed for the sparse MATVEC case help us to judge the performance of two important computational phases of PETSc-FUN3D. The triangular solution phase runs close to the memory bandwidth limit while the flux evaluation (which has high floating point work to do per memory reference) is bounded by the instruction scheduling limit.

We identify scalability bottlenecks for a fixed-size problem when the number of processors grows to several thousand. The implicit synchronizations and vector scatter/gather operations turn out to be the primary nonscaling phases. The implicit synchronizations get worse primarily because of the slight load imbalances that arise as the problem size per processor gets smaller. The mesh partitioning strategy that tries to minimize this load imbalance gives better scalability at large granularity. The vector/gather operation involves message packing before sending and message unpacking after receiving. Both of these steps are memory bandwidth bound, since the message is extracted from a large global vector (touching many noncontiguous cache lines) during the packing stage and then put back into another large vector during the unpacking stage.

The algorithmic efficiency of the Schwarz preconditioner depends on the number of subdomains, amount of overlap among the subdomains, and the allowed fill level (k) in the each subdomain-based incomplete factorization (ILU(k)). For PETSc-FUN3D on the ASCI Red machine, we find that the block Jacobi (additive Schwarz with zero overlap) and ILU(1) give the best execution time, especially when the number of subdomains is large.

The hybrid (mixed message passing and shared memory) programming model achieves better performance than the corresponding pure message passing case, primarily because it allows each subdomain to be chunkier (with better computation to communication ratio), relative to further

subdivision. In addition, we need to pay attention to the work division among threads, cache locality, and update management. We recommend using this model only in the computational phases that are not memory bandwidth limited (otherwise, threads will compete for the available memory bandwidth, involving more synchronizations at the hardware level). In addition, better load balancing can be achieved by dynamic thread scheduling.

In summary, this work highlights many important performance issues relevant to the PDE-based scalable solvers. The critical directions for future research are: (1) less synchronous algorithms, (2) memory latency tolerant algorithms, and (3) hybrid programming model for large number of threads per node. More detailed performance models are also needed, since knowing the "achievable" performance levels for different kernels can aid in identifying the optimizations that can be done at compile and run times.

The critical nature of this research is reflected in the research agenda of the U.S. Department of Energy through two of the multiyear multi-institution Scientific Discovery through Advanced Computing (SciDAC) projects: Terascale Optimal PDE Simulation (TOPS) and Performance Evaluation Research Center (PERC). The developments described in this dissertation have influenced the direction of these projects. Though architectures continue to evolve in their hardware details, our work addresses the trends that are becoming more pronounced as we follow the technology roadmap—the increasing gap between processor and memory performance and increased concurrency. As we anticipate machines capable of hundreds of teraflops, consisting of tens of thousands of processors with memory latencies of hundreds of cycles within the next three years, we are ready with analyses, algorithms, and software to engage them for PDE-based simulations.

REFERENCES

[1] W. K. Anderson and D. L. Bonhaus. An implicit upwind algorithm for computing turbulent flows on unstructured grids. *Computers and Fluids*, 23:1–21, 1994.

[2] W. K. Anderson, W. D. Gropp, D. K. Kaushik D. E. Keyes, and B. F. Smith. Achieving high sustained performance in an unstructured mesh CFD application. In *Proceedings of Supercomputing 1999*. IEEE Computer Society, 1999. Gordon Bell Prize Award Paper in Special Category.

[3] 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.

[4] D. F. Bailey. How to fool the masses when reporting results on parallel computers. *Supercomputing Review*, pages 54–55, 1991.

[5] S. Balay, K. R. Buschelman, W. D. Gropp, D. K. Kaushik, M. G. Knepley, L. C. McInnes, and B. F. Smith. PETSc home page. `http://www.mcs.anl.gov/petsc`, 2002.

[6] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient management of parallelism in object-oriented numerical software libraries. In *Modern Software Tools in Scientific Computing*, pages 163–201. Birkhauser, 1997.

[7] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. PETSc users manual. Technical Report ANL-95/11 – Revision 2.1.3, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439, 2002. (see `http://www.mcs.anl.gov/petsc`).

[8] C. Bischof, A. Carle, G. Corliss, A. Griewank, and P. Hovland. ADIFOR - generating derivative codes from fortran programs. *Scientific Programming*, 1:1–29, 1992.

[9] C. Bischof, L. Roh, and A. Mauer. ADIC — an extensible automatic differentiation tool for ANSI-C. Technical Report Technical Report ANL/MCS-P626-1196, Argonne National Laboratory, 1996.

[10] S. W. Bova, C. P. Breshears, C. E. Cuicchi, Z. Demirbilek, and H. A. Gabb. Dual-level parallel analysis of harbor wave response using MPI and OpenMP. *International Journal of High Performance Computing Applications*, 14:49–64, 2000.

[11] A. Brandt. Multi-level adaptive solutions to boundary value problems. *Math. Comp.*, 31:333, 1977.

[12] A. Brandt. Multigrid Techniques: 1984 Guide with Applications to Fluid Dynamics. Technical report, von Karman Institue, 1984.

[13] X.-C. Cai. Some domain decomposition algorithms for nonselfadjoint elliptic and parabolic partial differential equations. Technical Report 461, Courant Institute, New York, 1989.

[14] X.-C. Cai and M. Sarkis. A restricted additive Schwarz preconditioner for general sparse linear systems. *SIAM Journal of Scientific Computing*, 21:792–797, 1999.

[15] D. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture*. Morgan Kaufmann Publishers, 1996.

[16] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 24th National Conference of the ACM*, 1969.

[17] R. S. Dembo, S. C. Eisenstat, and T. Steihaug. Inexact Newton methods. *SIAM Journal of Numerical Analysis*, 19:400–408, 1982.

[18] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of Fortran basic linear algebra subprograms: Model implementation and test programs. *ACM Transactions on Mathematical Software*, 14:18–32, 1988.

[19] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. A set of level-3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16:1–28, 1988.

[20] M. Dryja and O. B. Widlund. An additive variant of the Schwarz alternating method for the case of many subregions. Technical Report 339, Department of Computer Science, Courant Institute, 1987.

[21] G. Golub and J. M. Ortega. *Scientific Computing*. Academic Press, Inc., 1993.

[22] W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith. Toward realistic performance bounds for implicit CFD codes. In D. Keyes, A. Ecer, J. Periaux, N. Satofuka, and P. Fox, editors, *Proceedings of Parallel CFD'99*, pages 233–240. Elsevier, 1999.

[23] W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith. High performance parallel implicit cfd. *Journal of Parallel Computing*, 27:337–362, 2001.

[24] W. D. Gropp, D. K. Kaushik D. E. Keyes, and B. F. Smith. Performance modeling and tuning of an unstructured mesh CFD application. In *Proceedings of Supercomputing 2000*. IEEE Computer Society, 2000.

[25] W. D. Gropp and E. Lusk. Reproducible measurements of MPI performance characteristics. In Jack Dongarra, Emilio Luque, and Tomàs Margalef, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 1697 of *Lecture Notes in Computer Science*, pages 11–18. Springer Verlag, 1999. 6th European PVM/MPI Users' Group Meeting, Barcelona, Spain, September 1999.

[26] W. D. Gropp, E. Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*, 2nd edition. MIT Press, Cambridge, MA, 1999.

[27] 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.

[28] W. D. Gropp, L. C. McInnes, M. D. Tidriri, and D. E. Keyes. Globalized Newton-Krylov-Schwarz algorithms and software for parallel implicit CFD. *International Journal of High Performance Computing Applications*, 14:102–136, 2000.

[29] W. Hackbusch. *Iterative Methods for Large Sparse Linear Systems*. Springer, 1993.

[30] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1996.

[31] P. D. Hough, T. G. Kolda, and V. J. Torczon. Asynchronous parallel pattern search for nonlinear optimization. Technical Report SAND2000-8213, Sandia National Laboratories, January 2000.

[32] G. Karypis and V. Kumar. A fast and high quality scheme for partitioning irregular graphs. *SIAM Journal of Scientific Computing*, 20:359–392, 1999.

[33] 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., editors, *Proceedings of the 10th International Conference on Domain Decomposition Methods*, pages 311–319. Wiley, 1997.

[34] 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., editors, *Proceedings of the 11th International Conference on Domain Decomposition Methods*, pages 513–520. Domain Decomposition Press, Bergen, 1999.

[35] C. T. Kelley and D. E. Keyes. Convergence analysis of pseudo-transient continuation. *SIAM Journal of Numerical Analysis*, 35:508–523, 1998.

[36] D. E. Keyes. Parallel Numerical Algorithms: An Introduction. In David E. Keyes, Ahmed Sameh, and V. Venkatakrishnan, editors, *Parallel Numerical Algorithms*. Kluwer Academic Publishers, 1996.

[37] D. E. Keyes. How scalable is domain decomposition in practice? In C.-H. Lai et al., editors, *Proceedings of the 11th International Conference on Domain Decomposition Methods*. Domain Decomposition Press, Bergen, 1999.

[38] D. E. Keyes. Terascale implicit methods for partial differential equations. In X. Feng and T. Schulze, editors, *Contemporary Mathematics*, volume 306, pages 29–84. AMS, 2002.

[39] D. E. Keyes, D. K. Kaushik, and B. F. Smith. Perspective for cfd on petsflops systems. In et. al. M. Hafez, editor, *CFD Review*, pages 1079–1096. World Scientific, 1998.

[40] D. J. Mavriplis. Parallel unstructured mesh analysis of high-lift configurations. Technical Report 2000-0923, AIAA, 2000.

[41] J. D. McCalpin. STREAM: Sustainable memory bandwidth in high performance computers. Technical report, University of Virginia, 1995. `http://www.cs.virginia.edu/stream`.

[42] 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.

[43] W. Mulder and B. Van Leer. Experiments with implicit upwind methods for the Euler equations. *Journal of Computational Physics*, 59:232–246, May 1985.

[44] Y. N. Patt, S. J. Patel, M. Evers, D. H. Friendly, and J. Stark. One billion transistors, one uniprocessor, one chip. *Computer*, 30(9):51–57, September 1997.

[45] Y. Saad and M. H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal of Scientific and Statistical Computing*, 7(3):856–869, July 1986.

[46] V. Schmitt and F. Charpin. Pressure distributions on the ONERA M6 wing at transonic Mach numbers. Technical Report AR-138, AGARD, May 1979.

[47] 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.

[48] B. F. Smith, P. Bjørstad, and W. Gropp. *Domain Decomposition: Parallel Multilevel Algorithms for Elliptic Partial Differential Equations*. Cambridge University Press, 1996.

[49] J. Stoer and R. Bulirsch. *Introduction to Numerical Analysis*. Springer-Verlag, New York, 1992.

[50] O. Temam and W. Jalby. Characterizing the behavior of sparse algorithms on caches. In *Proceedings of Supercomputing 1992*, pages 578–587. IEEE Computer Society, 1992.

[51] S. Toledo. Improving the memory-system performance of sparse-matrix vector multiplication. *IBM Journal of Research and Development*, 41:711–725, 1997.

[52] U. Trottenberg, A. Schuller, and C. Oosterlee. *Multigrid*. Academic, 2000.

[53] H. M. Tufo and P. F. Fischer. Fast parallel direct solvers for coarse grid problems. *Journal of Parallel and Distributed Computing*, 61:151–177, 2001.

[54] J. White and P. Sadayappan. On improving the performance of sparse matrix-vector multiplication. In *Proceedings of the 4th International Conference on High Performance Computing (HiPC '97)*, pages 578–587. IEEE Computer Society, 1997.

VITA

Dinesh Kaushik was born and raised in Bhiwani, India. He received his Bachelor and Master of Technology degrees in Aerospace Engineering in 1988 and 1991, respectively, from the *Indian Institute of Technology Kanpur*. He worked as a computational scientist at the *Indian Space Research Organization* during 1990-94, where he was part of a team designing thermal protective systems for the solid rocket engines of satellite launch vehicles. He joined the Computer Science Department at the Old Dominion University in Fall, 1995 as a masters student and became a doctoral student in Fall, 1996. He has been working full time as Research Associate at Argonne National Laboratory since January, 1999. His primary research interests are large-scale scientific computing, hybrid parallel programming model, and performance modeling and prediction.

The Department of Computer Science can be contacted by mail, telephone, or e-mail.

Department of Computer Science
Old Dominion University
Norfolk, VA 23529
(757) 683-3915
csgpd@odu.edu