PROSPECTS FOR CFD ON PETAFLOPS SYSTEMS*

DAVID E. KEYES †, DINESH K. KAUSHIK‡, AND BARRY F. SMITH§

Abstract. With teraflops-scale computational modeling expected to be routine by 2003–04, under the terms of the Accelerated Strategic Computing Initiative (ASCI) of the U.S. Department of Energy, and with teraflops-capable platforms already available to a small group of users, attention naturally focuses on the next symbolically important milestone, computing at rates of 10^{15} floating point operations per second, or "petaflop/s". For architectural designs that are in any sense extrapolations of today's, petaflops-scale computing will require approximately one-million-fold instruction-level concurrency. Given that cost-effective one-thousand-fold concurrency is challenging in practical computational fluid dynamics simulations today, algorithms are among the many possible bottlenecks to CFD on petaflops systems. After a general outline of the problems and prospects of petaflops computing, we examine the issue of algorithms for PDE computations in particular. A back-of-the-envelope parallel complexity analysis focuses on the latency of global synchronization steps in the implicit algorithm. We argue that the latency of synchronization steps is a fundamental, but addressable, challenge for PDE computations with static data structures, which are primarily determined by grids. We provide recent results with encouraging scalability for parallel implicit Euler simulations using the Newton-Krylov-Schwarz solver in the PETSc software library. The prospects for PDE simulations with dynamically evolving data structures are far less clear.

Key words. Parallel scientific computing, computational fluid dynamics, petaflops architectures

1. Introduction.

Future computing technology in general, and scientific computing technology in particular, will be characterized by highly parallel, hierarchical designs. This trend in design is a fairly straightforward consequence of two other trends: a desire to work with increasingly large data sets at increasing speeds and the imperative of cost-effectiveness. A system possessing large memory without a correspondingly large number of processors to act concurrently upon it is expensively out-of-balance. Fortunately, data use in most real programs has sufficient temporal and spatial locality to allow a distributed and hierarchical memory system, and this locality must be exploited at some level (by a combination of the applications programmer at the algorithmic level, the system software at the compiler and runtime levels, and the hardware). Research on petaflops[1] systems can be seen as paving the way for exploiting hierarchical parallelism at all levels. Indeed, "petaflops" has

*This report updates with new data and additional perspective a similar contribution by the same authors to appear in *CFD Review 1997*, M. Hafez, ed., Wiley, 1997.

†Associate Professor, Computer Science Department, Old Dominion University, and Associate Research Fellow, ICASE, keyes@icase.edu. Keyes' research was supported in part by the National Science Foundation under ECS-9527169 and by the National Aeronautics and Space Administration under NASA Contracts NAS1-19480 and NAS1-97046 while the author was in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23681-0001.

‡Graduate Research Assistant, Computer Science Department, Old Dominion University, and graduate intern, ICASE, kaushik@cs.odu.edu. Kaushik's research was supported in part by NASA contract NAGI-1692 and by a GAANN fellowship from the U. S. Department of Education.

§Computer Scientist, Mathematics and Computer Science Division at Argonne National Laboratory, bsmith@mcs.anl.gov. Smith's research was supported by U.S. Department of Energy, under Contract W-31-109-Eng-38.

[1]In order to distinguish the *plural* of "floating point operations" from the *rate* "floating point operations per second," the rate is customarily abbreviated "flop/s", with an explicit "/" for the "per". We retain this distinction when quoting measurements, but we do not distinguish between "petaflops" and "petaflop/s" when using the term as an adjective of scale. "Petaflops" will also be used in its general adjectival form to include the term "peta-ops," reflecting requirements to perform integer and logical computation at comparable rates, independently of or (often) in conjunction with floating point computation.

come to refer to body of research dealing with very highly parallel computing, since petaflops computers are likely to have between 10^4 and 10^6 processors, with deep memory hierarchies.

1.1. Petaflops Numerology. Casting petaflops-scale computing into popular terms is a worthwhile exercise even for the quantitatively elite, if for no other reason than that this staggering (and staggeringly expensive) capability must be explained to others. With apologies for drawing significance to any number with an arbitrary dimension attached (i.e., the second) except for its mnemonic value, we note that mainstream production scientific computing on workstations is carried out at approximately the *square-root* of 1 Pflop/s today: $\sqrt{10^{15}} \approx 31.5 \times 10^6$. The following commodity workstations perform the LINPACK-100 benchmark at a rate within a few percent of 31.5 Mflop/s [11]:

- SGI Indigo2 (200 MHz)
- IBM RS 6000-560 (50 MHz)
- DEC 3000-500 Alpha AXP (150 MHz)
- Sun Sparc 20 (90 MHz)

A typical sparse PDE computation performs somewhat below the dense LINPACK-100 rates, but with attention to cache residency through variable interleaving and subdomain blocking, it can come close.

There are also 31.5×10^6 seconds in a year, to within one-tenth of a percent. Therefore, a 1 Pflop/s computer could compute in one second what one of these workstations can compute in one year.

There are also 31.5×10^6 people presently living in the state of California, to within a few percent, based on an extrapolation from the 1990 federal census. Therefore, the processing power of a 1 Pflop/s computer (but not the requisite connectivity!) could be realized if everyone in California pooled a commodity scientific workstation to the task. This particular bit of numerology calls to mind that the electrical power consumption of a 1 Pflop/s computer built from commercial, off-the-shelf (COTS) components would be impressive.

As a final point of perspective, we note that the human brain has approximately 10^{12} neurons capable of firing at approximately 1 KHz, and is therefore a specialized peta-op/s "machine" weighing just three pounds and requiring far less power.

1.2. Interagency Petaflops Workshops. Since February 1994, there has been a systematic effort to explore the feasibility of and encourage the development of petaflops-scale computing by an informal interdisciplinary, interagency working group, subsets of which have met, typically for a week at a time, to consider:

- petaflops applications — what problems appear to require 1 Pflop/s or beyond for important benefits not achievable at smaller scales?
- petaflops architectures — how can balanced systems that store, transfer, and process the data of petaflops applications be supported with conceivable technologies?
- petaflops software — how can the gap between the complex hardware and the application community be spanned with tools that automate program preparation and execution?
- petaflops algorithms — how much concurrency can be exposed at various levels in a computational model and what fundamental requirements on capacity, bandwidth, latency, and processing arise from the underlying physics and mathematics?

The main contents of this report were originally created for, and have been informed by, the most recent of these meetings, the Petaflops Algorithms workshop in Williamsburg, VA, April 13-18, 1997. Fifty-five participants from federal agencies, universities, computer vendors, and other private computational organizations attempted to address the algorithmic research questions presented by potential of "affordable" petaflops systems by the year 2010.

The principal findings and recommendations have been outlined in [2], which concludes that petaflops computing is algorithmically feasible, in that at least some of today's key algorithms appear to be scalable to petaflops. Issues of interest to algorithmicists include the following, many of which are shared with the software and hardware communities:

- Concurrency
- Data locality
- Latency and synchronization
- Floating point accuracy (extended wordlength)
- Dynamic (data-adaptive) redistribution of workload
- Detailed performance analysis
- Algorithm improvement metrics
- New languages and constructs
- Role of numerical libraries
- Algorithmic adaptation to hardware failure

Participants made preliminary assessments of algorithm scalability, from as many diverse areas of high-performance computing as were represented, and applied a "triage"-style categorization: Class 1 – appearing to be scalable to petaflops systems, given appropriate effort; Class 2 – appearing scalable, provided certain significant research challenges are overcome; and Class 3 – appearing to possess major impediments to scalability, from our present perspective.

Many core algorithms from scientific computing were placed in Class 1 (scalable with appropriate effort), including: dense linear algebra algorithms; FFT algorithms (given sufficient global bandwidth); PDE solvers, based on static grids, including explicit and implicit schemes; sparse symmetric direct solvers, including positive definite and indefinite cases; sparse iterative solvers (given parallelizable preconditioners); "tree-code" algorithms for n-body problems and multipole or multiresolution methods; Monte Carlo algorithms for quantum chromodynamics; radiation transport algorithms; and certain highly concurrent classified (in the sense of national security) algorithms with *a priori* specifiable memory accesses.

Class 2 algorithms (scalable if significant challenges overcome) included a category of principal interest to CFD practitioners — namely, dynamic unstructured grid methods, including mesh generation, mesh adaptation and load balancing — along with several others: molecular dynamics algorithms; interior point-based linear programming methods; data mining, including associativity, clustering, and similarity search; sampling-based optimization, search, and genetic algorithms; branch and bound search algorithms; boundary element algorithms; symbolic algorithms, including Gröbner basis methods; discrete event simulation; certain further classified algorithms involving random memory accesses.

Into Class 3 (possessing major impediments to scalability) the participants placed: sparse unsymmetric Gaussian elimination, theorem-proving algorithms; sparse simplex linear programming algorithms; and integer relation and integer programming algorithms.

From these lists one may abstract the following contraindications for petaflops:

- Data dependencies that are random in characterization and determinable only at runtime (input-dependent dependencies);
- Insufficient speculative concurrency;
- Frequent uncoverable global synchronization;
- Multiphase algorithmic structure with disparate mappings of data to memories within alternating load-balanced phases; and
- Requirement of fast access to huge data sets by all processors.

Computational fluid dynamics as practiced at the contemporary state-of-the-art for problems with complex physics is sometimes characterized by this list. Adaptive methods cannot be statically balanced and mapped across processors, making incremental dynamic balancing and mapping necessary, together with performance monitoring and performance estimation to make cost-benefit analyses. Hybrid particle-field techniques often have unbalanced sequential phases when either the particle or the field computation is given priority over the other in data distribution. Lookup tables for complex state equations, constitutive relations, and cross-sections or reaction coefficients are often too large to replicate on each processor, but too nonlocally accessed to partition without sacrifice of efficiency.

In addition to these readily apparent contraindications, there is another complementary pair, of relevance to fluid dynamics simulations:

- Work requirements that scale faster than than $M^{4/3}$, where M is the main memory capacity; and
- Memory requirements that scale faster than $W^{3/4}$, where W is the (arithmetic) work complexity.

This constraint between memory and work scaling (or, alternatively, between memory and execution time scaling) is *not* likely to be as painful an issue for PDE-based computations as it may be for some others, since it reflects an architectural decision that is largely influenced to accommodate stencil-type computations on three-dimensional space-time grids (as we discuss further below). It is however, a new constraint, as applied in a two-sided manner. CFD practitioners are accustomed to *either* a memory or a time constraint, which they play up against — running the largest job that fits in memory for as much time as required on a dedicated system or running a job up against a temporal deadline with as much resolution as can be afforded. A tightly-coupled petaflops-capable system will be delicately balanced in its hardware configuration for a specific memory/processing rate model. Such systems will be too rare and too expensive to turn over in a dedicated fashion for an indefinite amount of time. They will also be too expensive to use without employing the full amount of memory most of the time. Algorithms that can trade space for time (such as methods that can vary discretization order, and thus the number of operations per grid vertex) will therefore extend more gracefully to an architecturally and economically constrained machine than algorithms that can only be run at a specific operation-count-to-memory ratio.

1.3. Technology Outlook. We conclude our introduction with a glimpse at a baseline COTS petaflops machine, and at a couple of nontraditional architectural directions. As we quote the educated guesses of others in this section, we begin with a caveat from Yogi Berra, philosopher in Baseball's Hall of Fame:

"Prediction is hard. Especially the future. . . "

In its projections for the year 2007 (the target year of its current ten-year window, as of this writing) the Semiconductor Industry Association (SIA) anticipates that individual clock rates will continue their historically gratifying ascent as far as approx-

imately 2GHz and then level off. This implies that at least 500,000-fold instruction concurrency is required (to achieve a product of 10^{15} operations per second), some of which will be found at the subprocessor level. Based on this number, and informed by other technology extrapolations, Stevens [24] has projected a COTS design. He envisions a 2,000-node system, with 32 processors per node, totaling 64,000 processors. This leaves approximately 8-fold concurrency to be found within a processor's own pipelined instruction stream (e.g., through multiple functional units). With 65 GB of shared memory per node, the system would have an aggregate of 130 TB. Approximately 80,000 disks (failing at the rate of approximately one every hour) would back this memory. The overall memory hierarchy (from processor registers to disks) would have 8 levels. The 2 GHz-clock multifunctional unit processors would be fed by approximately 240 GB/s of loads and 120 GB/s stores apiece (assuming dominantly triadic operations, $a \leftarrow b$ op c). This requires 180 data Bytes per cycle in and out of Level-1 cache, which would take up about 70% of an overall 2,048-bit wide path from L1 to CPU. Extrapolating from present pricing trends and practices, such a machine would cost approximately $32M for the CPUs and $174M for the overall system. Power consumption would be 11.5 MW and the annual power bill would be approximately $12M.

Sterling has led a design team that is looking well beyond COTS technology. The Hybrid Technology, Multi-threaded (HTMT) architecture [25] is looking towards a 100 GHz clock from quantum logic processors. At this rate, there will be a latency to DRAM of approximately 10,000 clocks. The 7-layer memory hierarchy of HTMT traverses the temperature spectrum from non-uniform random access (NURA) registers, cryogenic RAM (CRAM), at liquid helium temperatures, SRAM at liquid nitrogen temperatures, conventional DRAM, and high density holographic RAM, (HRAM), backed by disk. Programmer-specified "thread affinity" will reduce data hazards.

The Processor-in-Memory (PIM) design of Kogge et al. [20] will feature 100 TB of memory in 10,000 to 20,000 chips, each of which contains about 50 embedded "CPUs." The memory system will be like a live file with filters attached.

All designs are subject to the so-called "Tyranny of DRAM," which states that bandwidth between memory and the processors must be proportional to processor consumption of operands, even if latency is covered (through prefetching or some other technique). Many kernels, like the DAXPY and the FFT, do work that is a small constant (or at most a logarithmic) multiple of the size of the data set. The tyranny implies that progressively remote and slower levels of the memory system must provide proportionally wider pathways of data towards the CPU, so that the bandwidth product can be maintained during computational phases that cycle through the entire data set and do little work with each element.

2. Partial Differential Equation Archetypes and Parallel Complexity. Partial differential equations come in a wide variety, which explains why we have national laboratories instead of general purpose PDE libraries. Evolution equations come in time-hyperbolic and time-parabolic flavors, and equilibrium equations come in elliptic and spatially hyperbolic or parabolic flavors. Generally, hyperbolic equations are challenging to discretize since they support discontinuities, but easy to solve when addressed in characteristic form. Conversely, elliptic equations are easy to discretize, but challenging to solve, since their Green's functions are global: the solution at each point depends upon the data at all other points. The algorithms naturally employed for "pure" problems of these types vary considerably. CFD spans all of these regimes. Its problems can be of mixed type, varying by region, or of mixed

type by virtue of being multicomponent in a single region (e.g., a parabolic system with an elliptic constraint). In a prospective discussion such as this one, we cannot afford to be algorithmically comprehensive, and fortunately, we do not need to be in order to accomplish some computational complexity estimates of generic value, since PDE computations have a great deal of complexity regularity within their algorithmic variety, due to their field nature. The resource requirements of a PDE problem can usually be characterized by the following parameters, for which typical values are suggested for problems in the ASCI class:

- N_x, spatial grid points (10^4–10^9)
- N_t, temporal grid points (1–...)
- N_c, components per point (1–10^2)
- N_a, auxiliary storage per point (0–25)
- N_s, grid points in "stencil" (7–30)

In terms of these parameters, typical memory requirements would be some small number of copies of the fields (successive iterates, overwritten in a shifted or moving-windowed manner) together with a copy of the current Jacobian: $N_x \cdot (N_c + N_a) + N_x \cdot N_c^2 \cdot N_s$. (We assume with the N_c^2 term in the Jacobian that all components depend upon all other components). The work for an explicit code, or for an implicit code in which the linear system is solved through a sparse iterative means, is a small multiple of: $N_x \cdot N_t \cdot (N_a + N_c^2 \cdot N_s)$.

For equilibrium problems solved by "good" implicit methods, work W scales slightly superlinearly in the problem size (or main memory M); hence the Amdahl-Case Rule applies: $M \propto W$. For evolutionary problems, work scales with with problem size times the number of timesteps. CFL-type arguments place the latter on the order of the resolution of each spatial dimension. For 3D problems, therefore, $M \propto W^{3/4}$, which leads to the conventional petaflops "memory-thin" scaling rule. The actual constant of proportionality between M and W can be adjusted over a very wide range by both discretization order (high-order implies more work per point and per memory transfer) and by algorithmic tuning. If frequent time frames are to be captured, other resources — disk capacity and I/O rates — must both scale linearly with W. This is a more stringent scaling than for memory. For reasons of scope, we do not further address the scaling of peripherals; however, we note that significant research remains to be done with archiving data and I/O to support petaflops computing.

2.1. PDE Archetypes and Software Toolchain. The Computational Archetypes project at Caltech [9] has identified PDE archetypes according to the following classification:

- Local mesh computations
  - Concurrent
    - Explicit update schemes, diagonal relaxation schemes
    - Sparse matrix-vector multiplications
  - Sequential
    - Triangular relaxation schemes
    - Sparse approximate factorization schemes
- Global dimensionally-split computations
  - spectral schemes
  - ADI-like schemes
- Direct linear algebraic computations
  - Gaussian elimination in various orderings

With due respect to the importance of the latter, we concentrate on the prime archetypes for parallel CFD: concurrent local mesh computations, explicit and iterative implicit.

Before confining our attention to a few quantitative aspects of the solution algorithm, we note that solvers are just one link in a "toolchain" [19] for PDE computations worth doing at petaflops scales. This toolchain involves:

- Geometric modeling and grid generation
- Discretization (and automated code generation)
- Error estimation and adaptive refinement (h- and/or p-type)
- Task assignment
  - Domain partitioning
  - Subdomain-to-processor mapping
- Solution
  - Grid and operator "coarsening"
  - Automated or interactive steering
- Visualization, postprocessing, and application interfacing
- Parallel performance analysis

The toolchain metaphor is useful in reminding that the solver is not all there is to a parallel computation, and may not be the most difficult part. Furthermore, the difficulty of one link may be affected by decisions in another, e.g., a solver may have to work harder in conjunction with a poor grid generator. The overall outcome of a computation may be limited by any weak link, making it difficult to attach relative merits to individual components. The toolchain metaphor is possibly misleading in that not all links are important in all problems, and not all important relationships are between links adjacent in list.

We make a few additional remarks on the toolchain, abstracting CFD-relevant remarks from [19]. Software components of the chain tend to be modular, with well-defined interfaces, because of both good design principles and the impossibility of any one individual or team being expert in all components. A few full, vertically integrated parallel toolchain environments exist today. Amdahl's "rake" eventually forces parallelization of all components; certainly, at least, for petaflops. As one tool is perfected, the parallel bottleneck shifts to another. Significant sharing and reuse of components occurs horizontally (across groups) at the "low" end of the toolchain. For instance grid generators and partitioners are easy to share since they interface to the rest of the environment through intermediate disk files. At higher levels, the compatibility of inner data structures becomes an issue, which limits sharing. Some reuse of software between components occurs vertically, such as between mesh generation and improvement algorithms, and between these and the solver. Though data-structure-specific, common operations are sufficiently generic to become candidates for vertical software reuse within a group (e.g., intermesh transfer operators, error estimators, and solvers for error estimators and for actual solution updates). The parallel scalability requirement discourages the use of graph algorithms that make frequent use of global information, such as eigenvectors. Instead, heavy use is made of maximal independent sets, which can be constructed primarily by a local, greedy algorithm, with local mediation at subdomain interfaces. Trees are generally avoided as primary data structures in important inner-loop nearest-neighbor operations of PDE-based codes. Crucial trade-offs exist between time to access grid and geometry information and total memory usage; redundant data structures can reduce indirection at the price of extra storage.

2.2. Algorithms for PDEs. An explicit PDE solution algorithm has the following algebraic structure in moving from iterate $\ell - 1$ to iterate ℓ:

$$\mathbf{u}^\ell = \mathbf{u}^{\ell-1} - \Delta t^\ell \cdot \mathbf{f}(\mathbf{u}^{\ell-1}),$$

or, for higher temporal order schemes, a more general, fully known right-hand side:

$$\mathbf{u}^\ell = \mathbf{F}(\mathbf{u}^{\ell-1}, \mathbf{u}^{\ell-2}, \ldots).$$

Let N be the discrete dimension of a 3D problem and P the number of processors. Assume that the domain is of unit aspect ratio so that the number of degrees of freedom along an edge is $N^{1/3}$, and that the subdomain-to-processor assignment is isotropic, as well. The concurrency is pointwise, $\mathcal{O}(N)$. Since the stencil is localized, the communication-to-computation ratio enjoys surface-to-volume scaling: $\mathcal{O}\left((\frac{N}{P})^{-1/3}\right)$. The communication range is nearest-neighbor, except for timestep selection, which typically involves a global CFL stability check. The synchronization frequency is therefore once per timestep, $\mathcal{O}\left((\frac{N}{P})^{-1}\right)$. Storage per point is low — just a small multiple of N, itself. The data locality in the stencil update operations can be exploited both "horizontally" (across processors) and "vertically" (in cache). Load balancing is a straightforward matter of equipartitioning gridpoints while cutting the minimal number of edges, for static quasi-uniform meshes. Load balance becomes nontrivial when grid adaptivity is combined with the synchronization step of timestep selection.

The discrete framework for an implicit PDE solution algorithm has the form:

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

with $\Delta t^\ell \to \infty$ as $\ell \to \infty$. We assume that pseudo-timestepping is used to advance towards a steady state. An implicit method may also be time-accurate, which generally leads to an easier problem than the steady-state problem, since the Jacobian matrix for the left-hand side is more diagonally dominant when the timestep is small. The sequence of nonlinear problems, $\ell = 1, 2, \ldots$, is solved with an inexact Newton method. The resulting Jacobian systems for the Newton corrections are solved with a Krylov method, relying only on matrix-vector multiplications, so the stencil-based sparsity is not destroyed by fill-in. The Krylov method needs to be preconditioned for acceptable inner iteration convergence rates, and the preconditioning is the "make-or-break" aspect of an implicit code. The other phases parallelize well already, being made up of DAXPYs, DDOTs, and sparse MATVECs.

The job of the preconditioner is to approximate the action of the Jacobian inverse in a way that does not make it the dominant consumer of memory or cycles in the overall algorithm. The true inverse A^{-1} is usually dense, reflecting the global Green's function of the continuous PDE operator approximated by A. Given $Ax = b$, we want B approximating A^{-1} and a rescaled system $BAx = Bb$ (left preconditioning) or $ABy = b,\ x = By$ (right preconditioning). Though formally expressible as a matrix, the preconditioner is usually implemented as a vector-in, vector-out subroutine. A good preconditioner saves both time and space by permitting fewer iterations in the innermost loop and smaller storage for the Krylov subspace. An Additive Schwarz preconditioner [6] accomplishes this in a localized manner, with an approximate solve in each subdomain of a partitioning of the global PDE domain. Optimal Schwarz methods also require solution of a global problem of small discrete dimension. Applying a preconditioner in an Additive Schwarz manner increases flop rates over a global preconditioner, since the smaller subdomain blocks maintain better cache residency.


Newton


Krylov


Schwarz

The pioneers of NKS methods.

Combining a Schwarz preconditioner with a Krylov iteration method inside an inexact Newton method leads to a recently assembled synergistic parallelizable nonlinear boundary value problem solver with a classical name: Newton-Krylov-Schwarz (NKS).

When nested within a pseudo-transient continuation scheme to globalize the Newton method [18], the implicit framework has four levels:

```
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 DAXPY update
    CHECK NONLINEAR CONVERGENCE
  enddo
enddo
```

The operations written in uppercase customarily involve global synchronizations.

The concurrency is pointwise, $\mathcal{O}(N)$, in most algorithmic phases *but only subdomainwise,* $\mathcal{O}(P)$*, in the preconditioner phase.* The communication-to-computation ratio is still mainly surface-to-volume, $\mathcal{O}\left((\frac{N}{P})^{-1/3}\right)$. Communication is still mainly nearest-neighbor in range, but convergence checking, orthogonalization/conjugation steps in the Krylov method, and the optional global problems add nonlocal communication. The synchronization frequency is often more than once per mesh-sweep, *up to the Krylov dimension (*K*),* $\mathcal{O}\left(K(\frac{N}{P})^{-1}\right)$. Similarly, *storage per point is higher by a factor of* $\mathcal{O}(K)$. Locality can still be fully exploited horizontally and vertically, and load balance is still straightforward for any static mesh.

2.3. Parallel Complexity Analysis. Given complexity estimates of the leading terms of:

- the concurrent computation,
- the communication-to-computation ratio, and
- the synchronization frequency,

and a model of the architecture including:

- internode communication (network topology and protocol reflecting horizontal memory structure), and
- on-node computation (effective performance parameters including vertical memory structure),

one can formulate optimal concurrency and optimal execution time estimates for parallel PDE computations, on per-iteration basis or overall (by taking into account any granularity dependence in the convergence rate).

For an algebraically simple example that is sufficient to elucidate the main issues in algorithm design, we consider a 2D stencil-based PDE simulation and construct a model for its parallel performance based on computation and communication costs. The basic parameters are as follows:

- n grid points in each direction, total memory $N = \mathcal{O}(n^2)$,
- p processors in each direction, total processors $P = p^2$,
- memory per node requirements $\mathcal{O}(n^2/p^2)$,
- execution time per iteration An^2/p^2 (A includes factors like number of components at each point, number of points in stencil, number of auxiliary arrays, amount of subdomain overlap),
- n/p grid points on a side of a single processor's subdomain,
- neighbor communication per iteration (neglecting latency) Bn/p, and
- cost of an individual reduction per iteration (assumed to be logarithmic in p with the frequency of global reductions included in the coefficient) $C \log p$.

A, B, and C are all expressed in the same dimensionless units, for instance, multiples of the scalar floating point multiply-add.

Putting the components together, the total wall-clock time per iteration is

$$T(n,p) = A\frac{n^2}{p^2} + B\frac{n}{p} + C\log p.$$

The first two terms fall as p increases; the last term rises slowly. An optimal p is found where $\frac{\partial T}{\partial p} = 0$, or

$$-2A\frac{n^2}{p^3} - B\frac{n}{p^2} + \frac{C}{p} = 0,$$

or

$$p_{opt} = \frac{B}{2C}\left[1 + \sqrt{1 + 8AC/B^2}\right] \cdot n.$$

Observe that p can usefully grow proportionally to n without limitation. The larger the problem size, the more processors that can be employed with the effect of reducing the execution time. In this limited sense, stencil-based PDE computations are scalable to arbitrary problem sizes and numbers of processors. The optimal running time is

$$T(n, p_{\text{opt}}(n)) = \frac{A}{\rho^2} + \frac{B}{\rho} + C\log(\rho n),$$

where $\rho = \frac{B}{2C}\left[1 + \sqrt{1 + 8AC/B^2}\right]$. This optimal time is *not* constant as the problem size (and number of processors) increases, but it degrades only logarithmically.

To simplify, consider the limit of infinite bandwidth so that the (asynchronous) nearest-neighbor exchanges take no time. Then,

$$p_{opt} = \sqrt{2A/C} \cdot n,$$

and

$$T(n, p_{\text{opt}}(n)) = C\left[\frac{1}{2} + \log(\sqrt{2A/C} \cdot n)\right].$$

This simple analysis is on a per-iteration basis; a fuller analysis would multiply this cost by an iteration count estimate that generally depends upon n and p. We observe that although an algorithm made up of this mix of operations is formally scalable, the number of processors amongst which the problem should be divided varies inversely with C, the coefficient of the global synchronization term, and running time varies proportionally. Recall that the main difference in complexity per iteration between explicit and implicit methods in this context is the much greater frequency of synchronization for implicit methods. One of the main benefits provided in return for this synchronization is freedom from CFL limitations, and hence the prospect of an iteration count that is not constrained by the resolution of the grid.

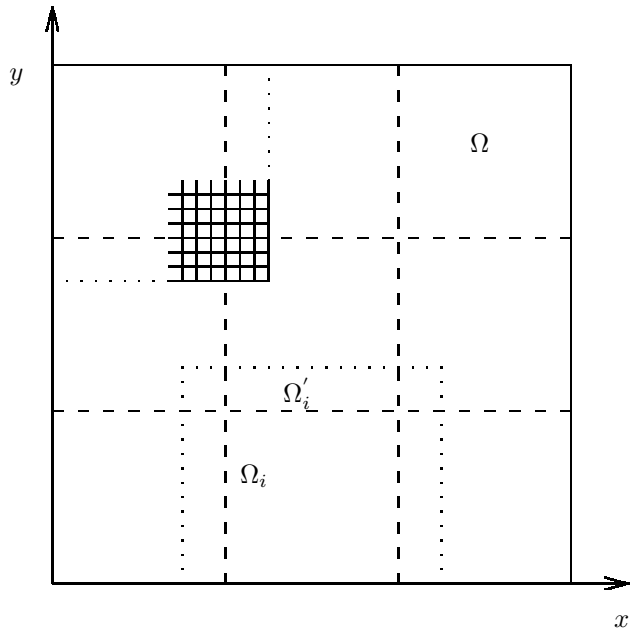
The synchronization cost is made of two parts: the hardware and software latency of accessing remote data when the data is, in fact, ready, and the synchronization delay when the data is not ready. Since they are difficult to distinguish in practice, we lump them together under the term "latency" and consider strategies for latency tolerance.

2.4. Latency Tolerance. From an architect's perspective [10], there are two classes of strategies for tolerating latency: amortization (block data transfers) and hiding or covering (precommunication, proceeding past an outstanding communication in the same thread, and multithreading). The requirements for tolerating latency are excess concurrency in the program (beyond the number of processors being used) and excess capacity in the memory and communication architecture, in order to stage operands near the processors.

Any architectural strategy has an algorithmic counterpart, which can be expressed in a sufficiently rich high-level language. For instance, prefetching is partially under programmer control in some recent commercially available language extensions. In addition, however, algorithmicists have a unique strategy, not available to architects by definition: reformulation of the problem to create concurrency. Algorithmicists may note that not all nonzeros are created equal, and can create additional concurrency by neglecting nonzero couplings in a system matrix when they stand in the way. Algorithmicists may also accept a (sufficiently rapidly converging) outer iteration that restores the coupling in a less synchronous way, if it improves the concurrency of the iteration body. The reduction in the cost per iteration must more than offset the cost of the restorative outer iterations. An understanding of the convergence behavior of the problem, especially the dependence of the convergence behavior on special exploitable structure, such as heterogeneity (region-dependent variation) and anisotropy (direction-dependent variation), is required in order to intelligently suppress nonzero data dependencies. We briefly mention some ideas for latency-tolerant preconditioners, latency-tolerant accelerators, and latency-tolerant formulations.

The Additive Schwarz method (ASM) named above as the innermost component of the implicit NKS method is a perfect illustration of latency-tolerant preconditioner. We take a closer look at the construction of this method.

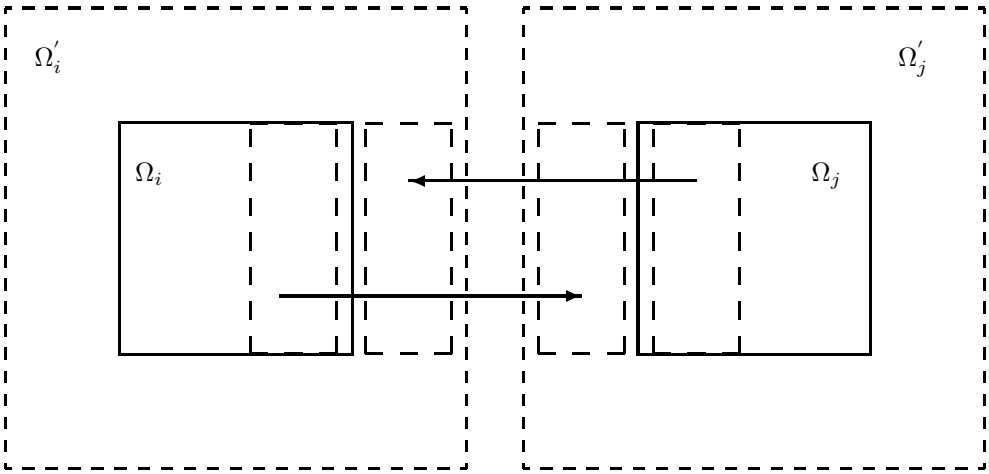
The operator B is formed out of (approximate) local solves on overlapping subdomains. The figure below shows a domain Ω decomposed into nine subdomains Ω_i, which are extended into overlapping subdomains Ω_i' that are cut off at the original boundary. The fine mesh spacing is indicated in one of the overlapping subdomains. This example is for a matching discretization in the overlapping subdomains, but nonmatching discretizations can be accommodated.



Let R_i and R_i^T be Boolean gather and scatter operations, mapping between a global vector discretized on the fine mesh and its i^{th} subdomain support, and let

$$B = \sum_i R_i^T \tilde{A_i}^{-1} R_i.$$

The concurrency thus created is proportional to the number of subdomains. Part of the action of the R_i is indicated schematically in the figure below. The bold right segment of Ω_i and the bold left segment of Ω_j are the same physical points. The overlapping subdomains are shown pulled apart, and the padding of each with interior data of the other is indicated by the arrows and dashed rectangles. (The width of the overlap is exaggerated for clarity in this illustration.)



The amount of overlap obviously determines the amount of communication and the amount of redundant computation (on non-owned, buffered points).

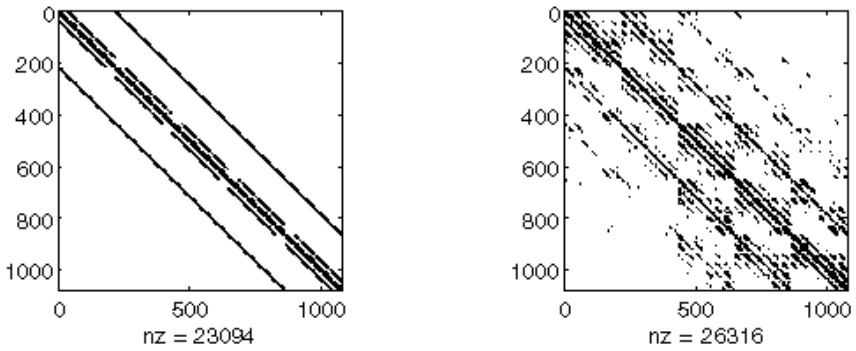
A two-level form of Additive Schwarz is provably optimal in convergence rate for some problems [23], but requires an exact solve on a coarsened grid. Convergence theorems for scalar 3D elliptically dominated systems may be summarized as follows, where I estimates the number of iterations as a function of problem size N and number of subdomains (and processors) P:

- No preconditioning: $I \propto N^{1/3}$
- Zero-overlap Schwarz preconditioning: $I \propto (NP)^{1/6}$
- Generous-overlap Schwarz preconditioning: $I \propto (P)^{1/3}$
- Two-level, generous overlap Schwarz preconditioning: $I = \mathcal{O}(1)$

The PETSc library [3, 4] includes portable parallel parameterized implementations of Schwarz preconditioners, including the new, more communication efficient, Restricted Additive Schwarz (RAS) method [8].

Another example of a latency-tolerant preconditioner is the form of the Sparse Approximate Inverse (SPAI) recently developed in [14]. Here B is formed in explicit, forward-multiply form by performing a sparsity-constrained norm minimization of $||AB - I||_F$. The minimization decouples into N independent least squares problems, one for each row of B. An adaptively chosen sparsity pattern, such that $||Ab_k - e_k||_2 < \epsilon$ leads to $\kappa(AB) \leq \sqrt{\frac{1+\delta}{1-\delta}}$, where $\delta \propto N\epsilon^2$. ϵ is chosen as a compromise between storage and convergence rate. The requirement on the smallness of ϵ appears pessimistic (in that B becomes denser as ϵ becomes smaller), but SPAI is worthwhile beyond the hypotheses of the theorem, just as Additive Schwarz is worthwhile with overlaps much smaller than required by the theory for optimality.

The concurrency created by SPAI is pointwise, in both the construction and the application of B. A parallel implementation of SPAI is described in [5]. (The next public release of PETSc will contain an interface to this package.) The sparsity profiles of an original matrix A and its SPAI, with a comparable number of differently positioned nonzeros are shown below (from [14]):



Modified forms of the classical Krylov accelerators of conjugate gradients (CG) and generalized minimal residuals (GMRES) can provide latency-tolerant accelerators. Krylov methods find the best solution to an N-dimensional problem in a K-dimensional Krylov space ($K \ll N$). Conventional Krylov methods orthogonalize (or conjugate) at every step to build up a well conditioned Krylov basis and to update the expansion coefficients of the solution in the enlarged basis. In infinite precision, this orthogonalization can be delayed for many steps at a time and "made up" in one multicomponent global reduction [12], some options for which are available in PETSc. In finite precision, delayed orthogonalization may be destabilizing, but for the low-accuracy requirements of an inner loop of a Newton method it may be tolerable, since the basis is flushed before it gets large. Furthermore, the requirement of performing all pairwise orthogonalizations may be avoided by construction during part of the iteration if the bases are generated from sparse seed vectors with sparse system matrices A. Many other tradeoffs of stability for reduced synchronization frequency have yet to be carefully investigated on realistic problems. Petaflops scale CFD will require a systematic assault on the synchronicity of Krylov basis generation.

The formulations of PDE algorithms, themselves may be made more latency-tolerant in ways that do not compromise the ultimate accuracy of the result, but only the minimal number of iterations required to achieve it. Many synchronization steps in conventional algorithms (e.g., convergence tests, global timestep selection) can be hidden by speculative computation of the next step based on a conservative prediction of the outcome. Such conservative predictions (that an iteration has not converged, or that a timestep cannot be increased) allow by-passing tests that would be recommended for minimal computational complexity if communication were free; but their communication costs may not justify the resulting instant adaptation.

Much work in PDE codes with complex physical models is related to updating auxiliary quantities used in Jacobian assembly, such as flop-intensive constitutive laws or communication-intensive table lookups. These can be "lagged" to slightly stale (or very stale) values with latency savings and acceptable convergence rate consequences.

Message-number versus message-volume trade-offs can be resolved in architecturally optimal ways, given latency and bandwidth models.

Furthermore, a "neighbor-computes" paradigm may sometimes be better than an "owner-computes" in cases in which the output of the computation is small but the inputs (residing on the neighbors) are large.

3. Case Study in the Parallel Port of an NKS-based CFD Code. Discussions of petaflops-scale computing ring hollow if not accompanied by experiences on contemporary parallel platforms that demonstrate that the currently provided

technology has been absorbed. We therefore include in this report some parallel performance results for a NASA unstructured grid CFD code that is used to study the high-lift, low-speed behavior of aircraft in take-off and landing configurations. Our primary test case, possessing only 1.4 million degrees of freedom, is miniscule on the petaflops scale, but we will show scalability of algorithmic convergence rate and per-iteration performance over a wide range of numbers of processors, which we have every reason to believe can be extended as the hardware becomes available.

The demonstration code, FUN3D [1], is a tetrahedral vertex-centered unstructured grid code developed by W. K. Anderson of the NASA Langley Research Center for compressible and incompressible Euler and Navier-Stokes equations. FUN3D uses a control volume discretization with variable-order Roe schemes for approximating the convective fluxes and a Galerkin discretization for the viscous terms. Our parallel experience with FUN3D is with the incompressible Euler subset thus far, but nothing in the solution algorithms or software changes for the other cases. Of course, convergence rate will vary with conditioning, as determined by Mach and Reynolds numbers and the correspondingly induced grid adaptivity. Furthermore, robustness becomes more of an issue in problems admitting shocks or making use of turbulence models. The lack of nonlinear robustness is a fact of life that is largely outside of the domain of parallel scalability. In fact, when nonlinear robustness is restored in the usual manner, through pseudo-transient continuation, the conditioning of the linear inner iterations is enhanced, and parallel scalability may be improved. In some sense, the Euler code, with its smaller number of flops per point per iteration and its aggressive trajectory towards the steady state limit may be a *more*, not less, severe test of scalability.

The solution algorithm we employ is pseudo-transient Newton-Krylov-Schwarz (ΨNKS), with point-block ILU(0) on the subdomains for the action of $\tilde{A_i}^{-1}$ (in the customary Schwarz notation; see above). The original code possesses a pseudo-transient Newton-Krylov solver already. Our reformulation of the global point-block ILU(0) of the original FUN3D into the Schwarz framework of the PETSc version is the primary source of additional concurrency. The timestep grows from an initial CFL of 10 towards infinity according to the switched evolution/relaxation (SER) heuristic of Van Leer & Mulder [21]. Our ΨNKS solver operates in a matrix-free, split-discretization mode, whereby the Jacobian-vector MATVEC operations required by the GMRES method are approximated by finite-differenced Fréchet derivatives of the nonlinear residual vector. The action of the Jacobian is therefore always "fresh." However, the submatrices used to construct the point-block ILU(0) factors on the subdomains as part of the Schwarz preconditioning are based on a lower-order discretization than the one used in the residual vector, itself. This is a common approach in practical codes, and the requisite distinctions within the residual and Jacobian subroutine calling sequences were available already in the FUN3D legacy version.

Conversion of the legacy FUN3D into the distributed memory PETSc version was begun in October 1996 and first demonstrated in March 1997. Since then, it has been undergoing continual enhancement, largely with respect to single-node aspects, namely blocking, variable interlacing, and edge-reordering for higher cache efficiency. The original five-month, part-time effort included: learning about FUN3D and its mesh preprocessor, learning the MeTiS unstructured grid partitioning tool, adding and testing new functionality in PETSc (which had heretofore been used with structured grid codes; see, e.g. [13]), and restructuring FUN3D from a vector to a cache orientation. Porting a legacy unstructured code into the PETSc framework would take

considerably less time today. Approximately 3,300 of the original 14,400 lines (primarily in FORTRAN77) of FUN3D are retained in the PETSc version. The retained lines are primarily SPMD "node code" for flux and Jacobian evaluations, plus some file I/O routines. PETSc solvers replace the rest. Parallel I/O and post-processing are challenges that remain.

3.1. Summary of Results on the Cray T3E and the IBM SP. We excerpt from a fuller report to appear elsewhere a pair of tables for a 1.4-million degree-of-freedom problem converged to near machine precision in approximately 6.5 minutes, using approximately 1600 global fine-grid flux balance operations (or "work units" in the multigrid sense) on 128 processors of a T3E or 80 processors of an SP. Relative efficiencies of 75% to 85% are obtained over this range The physical configuration is a three-dimensional ONERA M6 wing up against a symmetry plane. This configuration has been extensively studied by our colleagues at NASA and ICASE, and throughout the international aerospace industry generally, as a standard case. Our tetrahedral Euler grids were generated by D. Mavriplis of ICASE. The grid of the problem most thoroughly reported on herein contains 357,900 vertices, which implies that a vector of four unknowns per vertex has dimension 1,431,600. We also present some results for a problem eight times larger, containing approximately 11 million degrees of freedom. (We can run this largest case only on the largest configurations of processors, which does not permit wide scalability studies at present.) We used a maximum Krylov dimension of 20 vectors per pseudo-timestep. The maximum CFL used in the SER pseudo-timestepping strategy is 10,000. The pseudo-timestepping is a nontrivial feature of the algorithm, since the norm of the steady state residual does not decrease monotonically in the largest grid case. (In practice, we might employ mesh sequencing so that the largest grid case is initialized from the converged solution on a coarser grid. In the limit, such sequencing permits the finer grid simulation to be initialized within the domain of convergence of Newton's method.)

The first table, for the Cray T3E, shows a relative efficiency in going from the smallest processor number for which the problem fits (16 nodes) to the largest available (128 nodes), of 85%. Each iteration represents one pseudo-timestep, including one Newton correction, and up to 20 Schwarz-preconditioned GMRES steps.

Cray T3E Performance (357,900 vertices)

procs	its	exe	speedup	η_{alg}	η_{impl}	$\eta_{overall}$
16	77	2587.95s	1.00	1.00	1.00	1.00
24	78	1792.34s	1.44	0.99	0.97	0.96
32	75	1262.01s	2.05	1.03	1.00	1.03
40	75	1043.55s	2.48	1.03	0.97	0.99
48	76	885.91s	2.92	1.01	0.96	0.97
64	75	662.06s	3.91	1.03	0.95	0.98
80	78	559.93s	4.62	0.99	0.94	0.92
96	79	491.40s	5.27	0.97	0.90	0.88
128	82	382.30s	6.77	0.94	0.90	0.85

Convergence is defined as a relative reduction in the norm of the steady-state nonlinear residual of the conservation laws by a factor of 10^{-10}. The convergence rate typically degrades slightly as number of processors is increased, due to introduction of increased concurrency in the preconditioner, which is partition-dependent, in general. We briefly explain the efficiency metrics in the last three columns of the tables.

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

- What scaling is to be used as the number of processors is varied?
  - overall fixed-size problem
  - varying size problem with fixed memory per processor
  - 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
  - adjust parameters to perform best on each given number of processors

In our implementations of NKS, we always adjust the subdomain blocking parameter to match the number of processors, one subdomain per processor; this causes the number of iterations to vary, especially since our subdomain partitionings are not nested. The effect of the changing-strength preconditioner should be examined independently of the general effect of parallel overhead, by considering separate algorithmic and implementation efficiency factors.

The customary definition of relative efficiency in going from q to p processors ($p > q$) is

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

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:

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

The second table, for the IBM SP[2], shows a relative efficiency of 75% in going from 8 to 80 nodes. The SP has 32-bit integers, rather than the 64-bit integers of the T3E, so the integer-intensive unstructured-grid problem fits on just eight nodes. The average per node computation rate of the SP is about 50% greater than that of the T3E for the current cache-optimized version of the code.

IBM SP Performance (357,900 vertices)

procs	its	exe	speedup	η_{alg}	η_{impl}	$\eta_{overall}$
8	70	2897.46s	1.00	1.00	1.00	1.00
10	73	2405.66s	1.20	0.96	1.00	0.96
16	78	1670.67s	1.73	0.90	0.97	0.87
20	73	1233.06s	2.35	0.96	0.98	0.94
32	74	797.46s	3.63	0.95	0.96	0.91
40	75	672.90s	4.31	0.93	0.92	0.86
48	75	569.94s	5.08	0.93	0.91	0.85
64	74	437.72s	6.62	0.95	0.87	0.83
80	77	386.83s	7.49	0.91	0.82	0.75

[2]The configuration consists, more precisely, of 80 120MHz P2SC nodes with two 128 MB memory cards each connected by a TB3 switch, and is available at Argonne National Lab.

Algorithmic efficiency (ratio of iteration count of the less decomposed domain to the more decomposed domain – using the "best" algorithm for each processor granularity) is in excess of 90% over this range. The main reason that the iteration count is only weakly dependent upon granularity is that the pseudo-timestepping over the early part of the iteration provides some parabolicity.

Implementation efficiency is in excess of 82% over the experimental range, and near unit efficiency is maintained over the early part of the range. Implementation efficiency is a balance of two opposing effects in modern distributed memory architectures. It may improve slightly as processors are added, due to smaller workingsets on each processor, with resulting better cache residency. Implementation efficiency ultimately degrades as communication-to-computation ratio increases for a fixed-size problem after the benefits of cache residency saturate.

The low (82%) implementation efficiency for the 80-processor SP can be accounted for almost completely by communication overhead. PETSc provides detailed profiling capabilities that provide the communication timings. The percentage of wallclock time spent in communication and synchronization on 80 processors of the SP is:

- 6% on nearest-neighbor communication to set ghostpoint values needed in function and Jacobian stencil computation (implemented using PETSc's vector scatter operations);
- 13% on globally synchronized reduction operations, further subdivided into:
  - 5% on norms, required in convergence tests, in vector normalizations in GMRES, and in differencing parameter selection in matrix-free MATVECs, and
  - 8% on groups of inner products, required in the classical Gram-Schmidt orthogonalization in GMRES. (Note that the percentage lost to inner products would be much higher if the modified Gram-Schmidt (recommended in [22] for numerical stability reasons but not needed in this application) were used, since the modified version synchronizes on each individual inner product.)

The effect on efficiency of the neighbor and global communications required in implicit methods for the parallel solution of PDEs is clearly seen from this profiling. There is, of course, *some* concurrency available in the scatter, norm, and inner product operations, so the overall efficiency deficit is not quite as large as the percentage occupied by these three main contributors. However, reducing them would sharply increase efficiency. We would expect an explicit code that was tuned to synchronize only rarely on timestep updates to obtain upwards of 90% fixed-size efficiency on the SP, instead of 82%.

The IBM SP has communications performance (in both bandwidth and latency) that is particularly poor in relation to its excellent computational performance. However, on any parallel computer with thousands of processors, algorithms requiring frequent global reductions will be of major concern.

Since we possess a sequence of unstructured Euler grids, we can perform a Gustafson-style scalability study by varying the number of processors and the discrete problem dimension in proportion. We note that the concept of Gustafson-style scalability does not extend perfectly cleanly to nonlinear PDEs, since added resolution brings out added physics and (generally) poorer conditioning, which may cause a shift in the "market basket" of kernel operations as the work in the nonlinear and linear phases varies. However, our shockless Euler simulation is a reasonably clean setting for this study, if corrected for iteration count. The table below shows three computations on

the T3E over a range of 40 in problem and processor size, while maintaining approximately 4500 vertices per processor.

Cray T3E Performance — Gustafson scaling

vert	procs	vert/proc	its	exe	exe/it
357,900	80	4474	78	559.93s	7.18s
53,961	12	4497	36	265.72s	7.38s
9,428	2	4714	19	131.07s	6.89s

The good news in this experiment is contained in the final column, which shows the average time per parallelized pseudo-time NKS outer iteration for problems with similarly sized local workingsets. Less than a 7% variation in performance occurs over a factor of nearly 40 in scale. Provided that synchronization latency can be controlled as the number of processors is increased, via the ideas discussed in the previous section and many others not yet invented, we expect that indefinite scaling is possible. We insert the caveat that most petaflops-scale PDE computations will not be homogeneous, but will consist of interacting tasks with different types of physics and algorithmics. Predictions of scalability are invariably problem-dependent when such interactions need to be taken into account. Furthermore, most petaflops-scale PDE computations will require dynamically adaptive gridding, and the adaptivity phase may not scale anywhere near as gracefully as the solution phase exhibits here.

We have concentrated in this report on distributed aspects of high performance computing — specifically on potential limits to attainable computational rates coming from bottlenecks to concurrency exploitation. From a processor perspective we have looked outward rather than inward. Since the aggregate computational rate is a product of the concurrency and the rate at which computation occurs in a single active thread, we should discuss the per-node performance of the code. On the 80-node IBM SP the sustained floating point performance of the PDE solver (excluding the initial I/O and grid setup and excluding terminal I/O) is 5.5 Gflop/s — or 69 Mflop/s per node in sustained parallel implicit mode. We claim that this is excellent performance for a sparse matrix code and we know of only a handful of highly tuned CFD codes that are claimed by others to execute with comparable per-node performance on the same hardware. Nevertheless, it is only 14% of the machine's peak performance.[3] It required considerable effort to get the per-node performance this high. Compiling and running the FUN3D code — which was written for vector machines, not cache-based microprocessors — out of the box on the same hardware, in serial, yields only 2% of peak performance. Like most codes that are not tuned for cache locality, it runs closer to the speed of the memory system than to the speed of the processor.

On 8 processors the sustained performance of the cache-tuned FUN3D is about 16% percent of peak, and extrapolating to one processor (by means of comparison of 1- and 8-processor performance on a smaller problem), the sustained performance on one processor would be about 18% of peak. We conclude from this that improved per-node performance of sparse PDE applications on cache-based microprocessors represents an opportunity for a factor of four or five, apart from replication of processors. The problem is a familiar one with a welcome cause — iterative solution algorithms are themselves highly efficient in terms of the total number of operations performed per word of storage. However, algorithms, compilers, and runtime systems must now be

[3]Each 120MHz processor issues up to four floating point instructions per clock for a theoretical peak of 480 Mflop/s per processor. However, the particular configuration at Argonne is "thin", possessing only half of the maximum possible processor-memory bandwidth.

coordinated to minimize the number of times a word is transferred between cache and main memory. The 18% extrapolated peak per-node performance is obtained after code optimizations including blocking, geometric reordering (of gridpoints), algebraic reordering (field interlacing), and unrolling, which are beyond the scope of this chapter and will be described in detail elsewhere.

The degradation of per-node performance with increasing numbers of processors (from 18% to 14% of peak in going from 1 to 80 processors) stands in contrast to our early experiences with the code, before the four optimizations just mentioned. Previously, we routinely obtained superunitary parallel efficiencies powered by better cache locality due simply to smaller workingsets per node. A dubious (in the parallel context) reward for cache optimization is that it improves the single-processor (large memory per node) performance more than the multi-processor performance. However, the effect of the cache is so important that it is not insightful to quote parallel efficiencies on anything but a cache-tuned code. Only after a code is tuned for good cache performance, can the effect of surface-to-volume (communication-to-computation) ratio be measured. For instance, on 64 SP processors, the case with 1.4 million degrees of freedom executed at a sustained aggregate 4.9 Gflop/s, whereas the case with 11 million degrees of freedom executed at a sustained aggregate rate of 5.5 Gflop/s.

We conclude this section by presenting fixed-size scalings for the finest grid case that we have run to date on the IBM SP and on the Cray T3E. The 2.8 million vertex grid is nested in the 0.36 million vertex grid used in the scalability studies above, by subdivision of each tetrahedron into eight. It is the largest grid yet generated by our colleagues at NASA Langley for an implicit wing computation. Coordinate and index data (including 18 million edges) occupies an 857 MByte file.

On the SP, the problem does not fit comfortably in core on less than 64 processors (to the nearest power of 2); on the T3E, with its long integers, that number is 128 processors. Our SP (at Argonne) contains 80 processors and our T3E (at NERSC) contains 512, so scalings of 1.25 and 4.0 are possible, respectively.

Though a factor of 1.25 in processor number is a very inconclusive range over which to perform scaling studies, we note a near-perfect speedup on the SP:

IBM SP Performance (2,761,774 vertices)

procs	its	exe	speedup	η_{alg}	η_{impl}	$\eta_{overall}$	Gflop/s
64	163	9,160.91s	1.00	1.00	1.00	1.00	5.5
80	162	7,330.73s	1.25	1.01	0.99	1.00	6.8

On the T3E, we note a speedup of 3.34 out of 4.0:

Cray T3E Performance (2,761,774 vertices)

procs	its	exe	speedup	η_{alg}	η_{impl}	$\eta_{overall}$	Gflop/s
128	164	6,048.37s	1.00	1.00	1.00	1.00	8.5
256	166	3,242.10s	1.87	0.99	0.94	0.93	16.6
512	171	1,811.13s	3.34	0.96	0.87	0.83	32.1

It is interesting to note the source of the degradation in going from 128 to 512 processors, since much finer granularities will be required in Petaflops architectures. The maximum over all processors of the time spent at global synchronization points (reductions — mostly inner products and norms) is 12% of the maximum over all processors of the wall-clock execution time. This is almost entirely idle time arising from load imbalance, not actual communication time, as demonstrated by inserting

barriers before the global reductions and noting that the resulting fraction of wall-clock time for global reductions drops below 1%. Closer examination of partitioning and profiling data shows that although the distribution of "owned" vertices is nearly perfectly balanced, and with it the "useful" work, the distribution of ghosted nodes can be very imbalanced, and with it, the overhead work and the local communication requirements. In other words, the partitioning objective of minimizing total edges cut while equidistributing vertices does *not*, in general, equidistribute the execution time between synchronization points, maily due to the skew among the processors in ghost vertex responsibilities. This example of the necessity of supporting multiple objectives (or multiple constraints) in mesh partitioning has been communicated to the authors of major partitioning packages, who have been hearing it from other sources, as well. We expect that a similar computation after such higher level needs are accommodated in the partitioner will achieve close to 95% overall efficiency on 512 nodes.

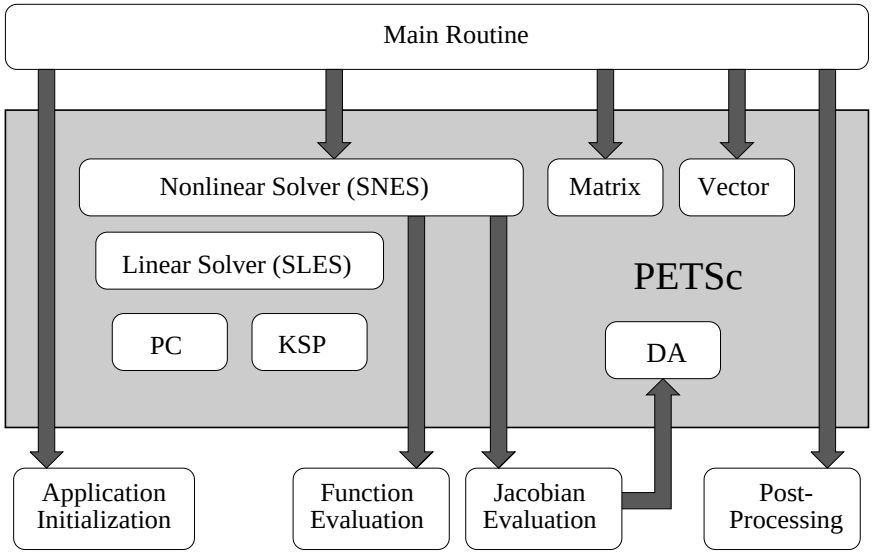
As a point of humility, we note that the performance of this code on one of the best hardware platforms available as of the date of writing is a factor of approximately 31,000 shy of 1 Petaflop/s.

4. Parallel Implementation Using PETSc. The parallelization paradigm we illustrate above in approaching a legacy code is a compromise between the "compiler does all" and the "hand-coded by expert" approaches. We employ the "Portable, Extensible Toolkit for Scientific Computing" (PETSc) [3, 4], a library that attempts to handle, in a highly efficient way, through a uniform interface, 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, pre-allocating 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 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 linesearch 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 version 2, first released in June 1995, has been downloaded thousands of times by users worldwide. PETSc has features relevant to computational fluid dynamicists, 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 ΨNKS application appears below. The arrows represent calls that cross the boundary between application-specific code and PETSc library code; all other details 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 state vectors. 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 **u**, but are most conveniently stored instead of recalculated, such as constitutive quantities.)



From our experience in writing and rewriting PDE codes for cache-based distributed memory machines, we have the following recommendations, which will undoubtedly continue to be relevant as codes are written in anticipation of an ultimate petaflops port.

- Replace global vector-based disk-striped data orderings (e.g., node colorings) with cache-based data orderings (e.g., subblocks) at the outer level.
- Interlace unknown fields so that most rapid ordering is within a point, not between points.
- Use the most convenient naming (global or local) for each given task, maintaining translation capability:
  - Physical boundary conditions rely on global names.
  - Many interior operations can be carried over from the uniprocessor code to SPMD node code by a simple "1-to-n" loop, with remapped entity relations (e.g., "vertices of edges", "edges of cells").
- Apply memory conservation aggressively; consider recomputation in cache rather than storage in memory.
- Micromanage storage based on knowledge of horizontal (e.g., network node) and vertical (e.g., cache) boundaries.

These recommendations do not provide explicit recognition for parallelism at the multiple functional unit level within a processor (and therefore within a cache). Within this level, vertex colorings can be applied to provide more fine-grained concurrency in local stencil updates.

5. Nontraditional Sources of Concurrency. We step back briefly from our narrow focus on data parallelism through spatial decomposition of a PDE grid to consider less traditional means of discovering the million-fold concurrency that will be required for petaflops-scale computing.

Time-parallelism is a counterintuitive but demonstrably interesting source of concurrency, even in evolutionary, causal simulations. A key idea of time-parallelism is

that not all of the work that goes into producing a converged solution at time level ℓ is sequentially captive to a converged solution at time level $\ell - 1$. When an iterative method is employed, different components of the error may converge at different stages, and useful work may conceivably begin at level ℓ before the solution at $\ell - 1$ is completely globally converged. This is particularly true in nonlinear problems. The direction, volume, and granularity of interprocessor communications in temporal parallelism are different from those of spatial parallelism, as are the memory scalings, since multiple time-frames of the problem proportional to the temporary concurrency must be kept in fast memory. For reasons of scope, we do not pursue the corresponding parallel complexities here, but refer to [16, 17].

In addition to the data parallelism within an individual PDE analysis, there is data parallelism *between* PDE analyses when the analyses are evaluations of objective functions or enforcements of state variable constraints within a computational optimization context. Computational fluid dynamics is not about individual large-scale analyses, done fast and well-resolved and "thrown over the wall." Both the results *and* their sensitivities are desired. Often multiple forcings (right-hand sides) are available *a priori*, rather than sequentially, which permits concurrent evaluation. Petaflops-scale computing for CFD will arrive in the form of 100 quasi-independent analyses running on 10,000 1Gflop/s processors earlier than in the form of 1 analysis running on 1,000,000 1Gflop/s processors.

Finally, we recall that computational fluid dynamics is not bound to a PDE formulation. The continuum approach is convenient, but not fundamental. In a flat, global memory system, it is natural to solve Poisson equations; in a hierarchical, distributed memory system, it is less natural. Nature is statistical, and enforces elliptic constraints like incompressibility through fast local collision processes. Among major phenomena in CFD only radiation is fundamentally "action at a distance." Lattice gas models have had a discouraging history, perhaps because they are too highly quantized, requiring massive statistics, and because their fundamental operations cannot exploit floating point hardware. Lattice Boltzmann models, on the other hand, seem highly promising. They are still quantized in space and time, but not in particle number, as quantized particles are replaced with continuous probability distribution functions. Lattice Boltzmann models possess ideal petaflops-scale concurrency properties: their two phases or relaxation and advection are alternatively completely local and nearest-neighbor in nature. There is no inherent global synchronization, except for assembling a visualization.

6. Summary Observations. The PDE-based algorithms for general purpose CFD simulations that we use today will in theory[4] scale to petaflops, particularly as the equilibrium simulations that are prevalent today go over to evolutionary simulations, with the superior linear conditioning properties of the latter in implicit contexts. The pressure to find latency tolerant algorithms intensifies. Longer word lengths (e.g., 128-bit floats) anticipated for petaflops-scale architectures, for more finely resolved — and typically worse-conditioned — problems, can assist in those forms of latency toleration, such as delayed orthogonalization, that are destabilizing. Solution algorithms are, in some sense, the "easy" part of highly parallel computing, and thornier issues such as parallel I/O and parallel dynamic redistribution schemes may ultimately determine the practical limits to scalability.

[4]We are warned by Philosopher Berra: *"In theory there is no difference between theory and practice. In practice, there is."*

Summarizing the "state-of-the-art" of architectures and programming environments, as they affect parallel CFD, we believe that:

- Vector-awareness is *out*; cache-awareness is *in*; but vector-awareness *will return* in subtle ways having to do with highly multiple-issue processors.
- Except for the Tera machine and the presently installed vector base, near-term large-scale computer acquisitions will be based on commodity cache-based processors.
- Driven by ASCI, large-scale systems will be of distributed-shared memory (DSM) type: shared in local clusters on a node, with the nodes connected by a fast network.
- Codes written for the Message Passing Interface (MPI) are considered "legacy" already and will therefore continue to be supported in the DSM environment; MPI-2 will gracefully extend MPI to effective use of DSM and to parallel I/O.
- High-performance Fortran (HPF) and parallel compilers are not yet up to the performance of message-passing codes, except in limited settings with lots of structure to the memory addressing [15]. Hybrid HPF/MPI codes are possible steps along the evolutionary process, with high-level languages automating the expression and compiler detection of structured-address concurrency at lower levels of the PDE modeling.
- Automated source-to-source parallel translators, such as the University of Greenwich CAPTools project (which adds MPI calls to a sequential F77 input) may attain 80–95% of the benefits of the best manual practice [27], but the result is limited to the concurrency extractable from the original algorithm, like HPF. In many cases, the legacy algorithm should, itself, be replaced.
- Computational steering will be an important aspect of petaflops-scale simulations and will appear in the form of interpreted scripts that control SPMD compiled executables.

With respect to algorithms, we believe that:

- Explicit time integration is a solved problem, except for dynamic mesh adaptivity.
- Implicit methods remain a major challenge, since:
  - Today's algorithms leave something to be desired in convergence rate, and
  - All "good" implicit algorithms have *some* global synchronization.
- Data parallelism from domain decomposition is unquestionably the main source of locality-preserving concurrency, but optimal smoothers and preconditioners violate strict data locality.
- New forms of *algorithmic* latency tolerance must be found.
- Exotic methods should be considered at petaflops scales.

With respect to the interaction of algorithms with applications we believe that the ripest remaining advances are interdisciplinary:

- Ordering, partitioning, and coarsening must adapt to coefficients (grid spacing and flow magnitude and direction) for convergence rate improvement.
- Trade-offs between pseudo-time iteration, nonlinear iteration, linear iteration, and preconditioner iteration must be understood and exploited.

With respect to the interaction of algorithms with architecture, we believe that:

- Algorithmicists must learn to think natively in parallel and avoid introducing unnecessary sequential constraints.

- Algorithmicists should inform their choices with a detailed knowledge of the memory hierarchy and interconnection network of their target architecture. It should be possible to develop very portable software, but that software will have tuning parameters that are determined by hardware thresholds.

Acknowledgements. The authors owe a large debt of gratitude to W. Kyle Anderson of the Fluid Mechanics and Acoustics Division of the NASA Langley Research Center for providing FUN3D as the legacy code that drove several aspects of this work. Satish Balay, Bill Gropp, and Lois McInnes of Argonne National Laboratory co-developed the PETSc software employed in this paper, together with Smith, under the the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38. Keyes and Smith also collaborated on Section III of this article while in residence at the Institute for Mathematics and its Applications at the University of Minnesota during June, 1997. Program development time on the NASA SP2s was provided through the Ames/Langley NAS Metacenter, under the Computational Aero Sciences section of the High Performance Computing and Communication Program.

REFERENCES

[1] W. K. Anderson (1997), FUN2D/3D (homepage).
`http://fmad-www.larc.nasa.gov/~wanderso/Fun/fun.html`

[2] D. H. Bailey, et al. (1997), *The 1997 Petaflops Algorithms Workshop Summary Report.*
`http://www.ccic.gov/cicrd/pca-wg/pal97.html`

[3] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith (1997), *The Portable, Extensible Toolkit for Scientific Computing, version 2.0.17* (code and documentation).
`http://www.mcs.anl.gov/petsc`

[4] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith (1997), *Efficient Management of Parallelism in Object-Oriented Numerical Software Libraries*, Modern Software Tools in Scientific Computing, E. Arge, A. M. Bruaset and H. P. Langtangen, eds., Birkhauser Press, pp. 163–201.
`ftp://info.mcs.anl.gov/pub/petsc/scitools96.ps.gz`

[5] S. T. Barnard and Robert L. Clay (1997), *A Portable MPI Implementation of the SPAI Preconditioner in ISIS++*, Technical Report NAS-97-002, NASA Ames Research Center.
`http://science.nas.nasa.gov/Pubs/TechReports/NASreports/NAS-97-002/NAS-97-002.html`

[6] X.-C. Cai (1989), *Some Domain Decomposition Algorithms for Nonselfadjoint Elliptic and Parabolic Partial Differential Equations*, Technical Report 461, Courant Institute, NYU.

[7] X.-C. Cai, D. E. Keyes, and V. Venkatakrishnan (1995), *Newton-Krylov-Schwarz: An Implicit Solver for CFD*, in "Proceedings of the Eighth International Conference on Domain Decomposition Methods" (R. Glowinski et al., eds.), Wiley, New York, pp. 387–400; also ICASE TR 95-87.
`ftp://ftp.icase.edu/pub/techreports/95/95-87.ps`

[8] X.-C. Cai and M. Sarkis (1997), *A Restricted Additive Schwarz Preconditioner for Nonsymmetric Linear Systems*, Technical Report CU-CS 843-97, Computer Sci. Dept., Univ. of Colorado, Boulder.
`http://www.cs.colorado/edu/cai/public_html/papers/ras_v0.ps`

[9] M. Chandy, et al. (1996), *The Caltech Archetypes/eText Project.*
`http://www.etext.caltech.edu/`

[10] D. E. Culler, J. P. Singh, and A. Gupta (1997), *Parallel Computer Architecture*, Morgan-Kaufman Press (to appear).
`http://HTTP.CS.Berkeley.EDU/~culler/book.alpha/`

[11] J. J. Dongarra (1997), *Performance of Various Computers Using Standard Linear Equations Software*, Technical Report CS-89-85 Computer Science Dept., Univ. of Tennessee,

Knoxville.
http://www.netlib.org/benchmark/performance.ps

[12] J. Erhel (1995), *A parallel GMRES version for general sparse matrices*, ETNA **3**: 160–176.
http://etna.mcs.kent.edu/vol.3.1995/pp160-176.dir/pp160-176.ps

[13] W. D. Gropp, L. C. McInnes, M. D. Tidriri, and D. E. Keyes (1997), *Parallel Implicit PDE Computations: Algorithms and Software*, Proceedings of Parallel CFD'97, A. Ecer, et al., eds., Elsevier (to appear).
http://www.icase.edu/~keyes/papers/pcfd97.ps

[14] M. J. Grote and T. Huckle (1997), *Parallel Preconditioning with Sparse Approximate Inverses*, SIAM J. Sci. Comput. **18**:838–853.
http://www-sccm.stanford.edu/Students/grote/grote/spai.ps.gz

[15] M. E. Hayder, D. E. Keyes, and P. Mehrotra (1997), *A Comparison of PETSc Library and HPF Implementations of an Archetypal PDE Computation*, Proceedings of 4th National Symposium on Large-Scale Analysis and Design on High Performance Computers and Workstations (to appear).

[16] G. Horton (1994), *Time-parallelism for the massively parallel solution of parabolic PDEs*, in "Applications of High Performance Computers in Science and Engineering", D. Power, ed., Computational Mechanics Publications, Southampton (UK).

[17] G. Horton, S. Vandewalle, and P. H. Worley (1995), *An algorithm with polylog parallel complexity for solving parabolic partial differential equations*, SIAM J. Sci. Stat. Comput. **16**: 531–541.

[18] C. T. Kelley and D. E. Keyes (1996), *Convergence Analysis of Pseudo-Transient Continuation*, SIAM J. Num. Anal. (to appear); also ICASE TR 96-46.
ftp://ftp.icase.edu/pub/techreports/96/96-46.ps

[19] D. E. Keyes and B. F. Smith (1997), *Final Report on "A Workshop on Parallel Unstructured Grid Computations"*, NASA Contract Report NAS1-19858-92.
http://www.icase.edu/~keyes/unstr.ps

[20] P. M. Kogge, J. B. Brockman, V. Freeh, and S. C. Bass (1997), *Petaflops, Algorithms, and PIMs*, Proc. of the 1997 Petaflops Algorithms Workshop.

[21] W. Mulder and B. Van Leer (1985), *Experiments with Implicit Upwind Methods for the Euler Equations*, J. Comp. Phys. **59**: 232–246.

[22] Y. Saad and M. H. Schultz (1986), *GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems*, SIAM J. Sci. Stat. Comput. **7**: 865–869.

[23] B. F. Smith, P. E. Bjørstad, and W. D. Gropp (1996), *Domain Decomposition: Parallel Multilevel Algorithms for Elliptic Partial Differential Equations*, Cambridge Univ. Press.

[24] R. W. Stevens (1997), *Hardware Projects for COTS-based Designs*, Proc. of the 1997 Petaflops Algorithms Workshop.

[25] T. Sterling (1997), *Hybrid Technology Multithreaded Architecture: Issues for Algorithms and Programming*, Proc. of the 1997 Petaflops Algorithms Workshop.

[26] T. Sterling, P. Messina, and P. H. Smith (1995), *Enabling Technologies for Petaflops Computing*, MIT Press.

[27] J. C. Yan (1997), *By Hand or Not By Hand — A Case Study of Computer Aided Parallelization Tools for CFD Applications*, in "Proc. of the Intl. Conf. on Parallel and Distributed Processing Techniques and Applications" (H. R. Arabnia, ed.), CSREA, pp. 364–372.