# Four Horizons for Enhancing the Performance of Parallel Simulations Based on Partial Differential Equations

David E. Keyes

Department of Mathematics & Statistics
Old Dominion University, Norfolk VA 23529-0077, USA,
Institute for Scientific Computing Research
Lawrence Livermore National Laboratory, Livermore, CA 94551-9989 USA
Institute for Computer Applications in Science & Engineering
NASA Langley Research Center, Hampton, VA 23681-2199, USA
keyes@icase.edu,
http://www.math.odu.edu/~keyes

**Abstract.** Simulations of PDE-based systems, such as flight vehicles, the global climate, petroleum reservoirs, semiconductor devices, and nuclear weapons, typically perform an order of magnitude or more below other scientific simulations (e.g., from chemistry and physics) with dense linear algebra or N-body kernels at their core. In this presentation, we briefly review the algorithmic structure of typical PDE solvers that is responsible for this situation and consider possible architectural and algorithmic sources for performance improvement. Some of these improvements are also applicable to other types of simulations, but we examine their consequences for PDEs: potential to exploit orders of magnitude more processor-memory units, better organization of the simulation for today's and likely near-future hierarchical memories, alternative formulations of the discrete systems to be solved, and new horizons in adaptivity. Each category is motivated by recent experiences in computational aerodynamics at the 1 Teraflop/s scale.

## 1 Introduction

While certain linear algebra and computational chemistry problems whose computational work requirements are superlinear in memory requirements have executed at 1 Teraflop/s, simulations of PDE-based systems remain "mired" in the hundreds of Gigaflop/s on the same machines. A review the algorithmic structure of typical PDE solvers that is responsible for this situation suggests possible avenues for performance improvement towards the achievement of the remaining four orders of magnitude required to reach 1 Petaflop/s.

An ideal 1 Teraflop/s computer of today would be characterized by approximately 1,000 processors of 1 Gflop/s each. (However, due to inefficiencies within the processors, a machine sustaining 1 Teraflop/s of useful computation is more practically characterized as about 4,000 processors of 250 Mflop/s each.) There

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1–17, 2000.

are two extreme pathways by which to reach 1 Petaflop/s from here: 1,000,000 processors of 1 Gflop/s each (only wider), or 10,000 processors of 100 Gflop/s each (mainly deeper). From the point of view of PDE simulations on Eulerian grids, either should suit.

We begin in §2 with a brief and anecdotal review of progress in high-end computational PDE solution and a characterization of the computational structure and complexity of grid-based PDE algorithms.

A simple bulk-synchronous scaling argument (§3) suggests that continued expansion of the number of processors is feasible as long as the architecture provides a global reduction operation whose time-complexity is sublinear in the number of processors. However, the cost-effectiveness of this brute-force approach towards petaflop/s is highly sensitive to frequency and latency of global reduction operations, and to modest departures from perfect load balance.

Looking internal to a processor (§4), we argue that there are only two intermediate levels of the memory hierarchy that are essential to a typical domain-decomposed PDE simulation, and therefore that most of the system cost and performance cost for maintaining a deep multilevel memory hierarchy could be better invested in improving access to the relevant workingsets, associated with individual local stencils (matrix rows) and entire subdomains. Improvement of local memory bandwidth and multithreading — together with intelligent prefetching, perhaps through processors in memory to exploit it — could contribute approximately an order of magnitude of performance within a processor relative to present architectures. Sparse problems will never have the locality advantages of dense problems, but it is only necessary to stream data at the rate at which the processor can consume it, and what sparse problems lack in locality, they can make up for by scheduling. With statically discretized PDEs, the access patterns are persistent. The usual ramping up of processor clock rates and the width or multiplicity of instructions issued are other obvious avenues for per-processor computational rate improvement, but only if memory bandwidth is raised proportionally.

Besides these two classes of architectural improvements — more and better-suited processor/memory elements — we consider two classes of algorithmic improvements: some that improve the raw flop rate and some that increase the scientific value of what can be squeezed out of the average flop.

In the first category (§5), we mention higher-order discretization schemes, especially of discontinuous or mortar type, orderings that improve data locality, and iterative methods that are less synchronous than today's.

It can be argued that the second category of algorithmic improvements does not belong in a discussion focused on computational rates, at all. However, since the ultimate purpose of computing is insight, not petaflop/s, it must be mentioned as part of a balanced program, especially since it is not conveniently orthogonal to the other approaches. We therefore include a brief pitch (§6) for revolutionary improvements in the practical use of problem-driven algorithmic adaptivity in PDE solvers — not just better system software support for well understood discretization-error driven adaptivity, but true polyalgorithmic and

multiple-model adaptivity. To plan for a "bee-line" port of existing PDE solvers to petaflop/s architectures and to ignore the demands of the next generation of solvers will lead to petaflop/s platforms whose effectiveness in scientific and engineering computing might be scientifically equivalent only to less powerful but more versatile platforms. The danger of such a pyrrhic victory is real.

Each of the four "sources" of performance improvement mentioned above to aid in advancing from current hundreds of Gflop/s to 1 Pflop/s is illustrated with precursory examples from computational aerodynamics. Such codes have been executed in the hundreds of Gflop/s on up to 6144 processors of the ASCI Red machine of Intel and also on smaller partitions of the ASCI Blue machines of IBM and SGI and large Cray T3Es. (Machines of these architecture families comprise 7 of the Top 10 and 63 of the Top 100 installed machines worldwide, as of June 2000 [3].) Aerodynamics codes share in the challenges of other successfully parallelized PDE applications, though not comprehensive of all difficulties. They have also been used to compare numerous uniprocessors and examine vertical aspects of the memory system. Computational aerodynamics is therefore proposed as typical of workloads (nonlinear, unstructured, multicomponent, multiscale, etc.) that ultimately motivate the engineering side of high-end computing.

Our purpose is not to argue for specific algorithms or programming models, much less specific codes, but to identify algorithm/architecture stresspoints and to provide a requirements target for designers of tomorrow's systems.

## 2 Background and Complexity of PDEs

Many of the "Grand Challenges" of computational science are formulated as PDEs (possibly among alternative formulations). However, PDE simulations have frequently been absent in Bell Prize competitions (see Fig. 1). PDE simulations require a balance among architectural components that is not necessarily met in a machine designed to "max out" on traditional benchmarks. The justification for building petaflop/s architectures undoubtedly will (and should) include PDE applications. However, cost effective use of petaflop/s machines for PDEs requires further attention to architectural and algorithmic matters. In particular, a memory-centric, rather than operation-centric view of computation needs further promotion.

### 2.1 PDE Varieties and Complexities

The systems of PDEs that are important to high-end computation are of two main classifications: *evolution* (e.g., time hyperbolic, time parabolic) or *equilibrium* (e.g, elliptic, spatially hyperbolic or parabolic). These types can change type from region to region, or can be mixed in the sense of having subsystems of different types (e.g., parabolic with elliptic constraint, as in incompressible Navier-Stokes). They can be scalar or multicomponent, linear or nonlinear, but with all of the algorithmic accompanying variety, memory and work requirements after discretization can often be characterized in terms of five discrete parameters:

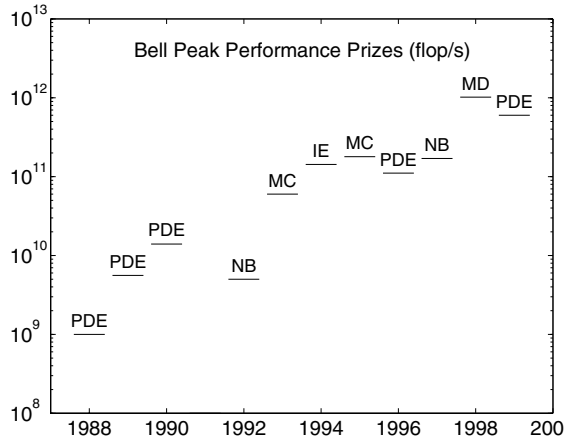



**Fig. 1.** Bell Prize Peak Performance computations for the decade spanning 1 Gflop/s to 1 Tflop/s. Legend: "PDE" – partial differential equations, "IE" integral equations, "NB" n-body problems, "MC" Monte Carlo problems, "MD" molecular dynamics. See Table 1 for further details.

**Table 1.** Bell Prize Peak Performance application and architecture summary. Prior to 1999, PDEs had successfully competed against other applications with more intensive data reuse only on special-purpose machines (vector or SIMD) in static, explicit formulations. ("NWT" is the Japanese Numerical Wind Tunnel.) In 1999, three of four finalists were PDE-based, all on SPMD hierarchical distributed memory machines.

| Year | Type | Application | Gflop/s | System | No. procs. |
|---|---|---|---|---|---|
| 1988 | *PDE* | structures | 1.0 | *Cray Y-MP* | 8 |
| 1989 | *PDE* | seismic | 5.6 | *CM-2* | 2,048 |
| 1990 | *PDE* | seismic | 14 | *CM-2* | 2,048 |
| 1992 | NB | gravitation | 5.4 | Delta | 512 |
| 1993 | MC | Boltzmann | 60 | CM-5 | 1,024 |
| 1994 | IE | structures | 143 | Paragon | 1,904 |
| 1995 | MC | QCD | 179 | NWT | 128 |
| 1996 | *PDE* | CFD | 111 | *NWT* | 160 |
| 1997 | NB | gravitation | 170 | ASCI Red | 4,096 |
| 1998 | MD | magnetism | 1,020 | T3E-1200 | 1,536 |
| 1999 | *PDE* | CFD | 627 | *ASCI BluePac* | 5,832 |

- $N_x$, number of spatial grid points ($10^6$–$10^9$)
- $N_t$, number of temporal grid points (1–unbounded)
- $N_c$, number of unknown components defined at each gridpoint (1–$10^2$)
- $N_a$, auxiliary storage per point (0–$10^2$)
- $N_s$, gridpoints in one conservation law "stencil" (5–50)

In these terms, memory requirements, $M$, are approximately $N_x \cdot (N_c + N_a + N_c^2 \cdot N_s)$. This is sufficient to store the entire physical state and allows workspace for an implicit Jacobian of the dense coupling of the unknowns that participate in the same stencil, but not enough for a full factorization of the same Jacobian. Computational work, $W$, is approximately $N_x \cdot N_t \cdot (N_c + N_a + N_c^2 \cdot (N_c + N_s))$. The last term represents updating of the unknowns and auxiliaries (equation-of-state and constitutive data, as well as temporarily stored fluxes) at each gridpoint on each timestep, as well as some preconditioner work on the sparse Jacobian of dense point-blocks.

From these two simple estimates comes a basic resource scaling "law" for PDEs. For equilibrium problems, in which solution values are prescribed on the boundary and interior values are adjusted to satisfy conservation laws in each cell-sized control volume, the work scales with the number of cells times the number of iteration steps. For optimal algorithms, the iteration count is constant independent of the fineness of the spatial discretization, but for commonplace and marginally "reasonable" implicit methods, the number of iteration steps is proportional to resolution in single spatial dimension. An intuitive way to appreciate this is that in pointwise exchanges of conserved quantities, it requires as many steps as there are points along the minimal path of mesh edges for the boundary values to be felt in the deepest interior, or for errors in the interior to be swept to the boundary, where they are absorbed. (Multilevel methods, when effectively designed, propagate these numerical signals on all spatial scales at once.) For evolutionary problems, work scales with the number of cells or vertices size times the number of time steps. CFL-type arguments place latter on order of resolution in single spatial dimension. In either case, for 3D problems, the iteration or time dimension is like an extra power of a single the spatial dimension, so Work $\propto$ (Memory)$^{4/3}$, with $N_c$, $N_a$, and $N_s$ regarded as fixed. The proportionality constant can be adjusted over a very wide range by both discretization (high-order implies more work per point and per memory transfer) and by algorithmic tuning. This is in contrast to the classical Amdahl-Case Rule, that would have work and memory directly proportional. It is architecturally significant, since it implies that a petaflop/s-class machine can be somewhat "thin" on total memory, which is otherwise the most expensive part of the machine. However, memory bandwidth is still at a premium, as discussed later. In architectural practice, memory and processing power are usually increased in fixed proportion, by adding given processor-memory elements. Due to this discrepency between the linear and superlinear growth of work with memory, it is not trivial to design a single processor-memory unit for a wide range of problem sizes.

If frequent time frames are to be captured, other resources — disk capacity and I/O rates — must both scale linearly with $W$, more stringently than for memory.

### 2.2 Typical PDE Tasks

A typical PDE solver spends most of its time apart from pre- and post-processing and I/O in four phases, which are described here in the language of a vertex-centered code:

- Edge-based "stencil op" loops (resp., dual edge-based if cell-centered), such as residual evaluation, approximate Jacobian evaluation, and Jacobian-vector product (often replaced with matrix-free form, involving residual evaluation)
- Vertex-based loops (resp., cell-based, if cell-centered), such as state vector and auxiliary vector updates
- Sparse, narrow-band recurrences, including approximate factorization and back substitution
- Global reductions: vector inner products and norms, including orthogonalization/conjugation and convergence progress and stability checks

The edge-based loops require near-neighbor exchanges of data for the construction of fluxes. They reuse data from memory better than the vertex-based and sparse recurrence stages, but today they are typically limited by a shortage of load/store units in the processor relative to arithmetic units and cache-available operands. The edge-based loop is key to performance optimization, since in a code that is not dominated by linear algebra this is where the largest amount of time is spent, and also since it contains a vast excess of instruction-level concurrency (or "slackness").

The vertex-based loops and sparse recurrences are purely local in parallel implementations, and therefore free of interprocessor communication. However, they typically stress the memory bandwidth within a processor/memory system the most, and are typically limited by memory bandwidth in their execution rates.

The global reductions are the bane of scalability, since they require some type of all-to-all communication. However, their communication and arithmetic volume is extremely low. The vast majority of flops go into the first three phases listed.

The insight that edge-based loops are load/store-limited, vertex-based loops and recurrences memory bandwith-limited, and reductions communication-limited is key to understanding and improving the performance of PDE codes. The effect of an individual "fix" may not be seen in most of the code until after an unrelated obstacle is removed.

### 2.3 Concurrency, Communication, and Synchronization

Explicit PDE solvers have the generic form:

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

where $\mathbf{u}^\ell$ is the vector of unknowns at time level $\ell$, $\mathbf{f}$ is the flux function, and $\Delta t^\ell$ is the $\ell^{th}$ timestep. Let a domain of $N$ discrete data be partitioned over an ensemble of $P$ processors, with $N/P$ data per processor. Since $\ell$-level quantities appear only on the left-hand side, concurrency is pointwise, $\mathcal{O}(N)$. The communication-to-computation ratio has surface-to-volume scaling: $\mathcal{O}\left((\frac{N}{P})^{-1/3}\right)$. The range of communication for an explicit code is nearest-neighbor, except for stability checks in global time-step computation. The computation is bulk-synchronous, with synchronization frequency once per time-step, or $\mathcal{O}\left((\frac{N}{P})^{-1}\right)$. Observe that both communication-to-computation ratio and communication frequency are constant in a scaling of fixed memory per node. However, if $P$ is increased with fixed $N$, they rise. Storage per point is low, compared to an implicit method. Load balance is straightforward for static quasi-uniform grids with uniform physics. Grid adaptivity or spatial nonuniformities in the cost to evaluate $\mathbf{f}$ make load balance potentially nontrivial. Adaptive load-balancing is a crucial issue in much (though not all) real-world computing, and its complexity is beyond this review. However, when a computational grid and its partitions are quasi-steady, the analysis of adaptive load-balancing can be usefully decoupled from the analysis of the rest of the solution algorithm.

Domain-decomposed implicit PDE solvers have the form:

$$\frac{\mathbf{u}^l}{\Delta t^l} + \mathbf{f}(\mathbf{u}^l) = \frac{\mathbf{u}^{l-1}}{\Delta t^l}, \quad \Delta t^l \to \infty.$$

Concurrency is pointwise, $\mathcal{O}(N)$, except in the algebraic recurrence phase, where it is only subdomainwise, $\mathcal{O}(P)$. The communication-to-computation ratio is still *mainly* surface-to-volume, $\mathcal{O}\left((\frac{N}{P})^{-1/3}\right)$. Communication still *mainly* nearest-neighbor, but convergence checking, orthogonalization/conjugation steps, and hierarchically coarsened problems add nonlocal communication. The synchronization frequency is usually more than once per grid-sweep, up to the dimension of the Krylov subspace of the linear solver, since global conjugations need to be performed to build up the latter: $\mathcal{O}\left(K(\frac{N}{P})^{-1}\right)$. The storage per point is higher, by factor of $\mathcal{O}(K)$. Load balance issues are the same as for the explicit case.

The most important message from this section is that a large variety of practically important PDEs can be characterized rather simply in terms of memory and operation complexity and relative distribution of communication and computational work. These simplifications are directly related to quasi-static grid-based data structures and the spatially and temporally uniform way in which the vast majority of points interior to a subdomain are handled.

## 3 Source #1: Expanded Number of Processors

As popularized in [5], Amdahl's law can be defeated if serial (or bounded concurrency) sections make up a nonincreasing fraction of total work as problem size and processor count scale together. This is the case for most explicit or iterative implicit PDE solvers parallelized by decomposition into subdomains. Simple, back-of-envelope parallel complexity analyses show that processors can

be increased as rapidly, or almost as rapidly, as problem size, assuming load is perfectly balanced. There is, however, an important caveat that tempers the use of large Beowulf-type clusters: the processor network must also be scalable. Of course, this applies to the protocols, as well as to hardware. In fact, the entire remaining four orders of magnitude to get to 1 Pflop/s could be met by hardware expansion alone. However, it is important to remember that this does not mean that fixed-size applications of today would run $10^4$ times faster; this argument is based on memory-problem size scaling.

Though given elsewhere [7], a back-of-the-envelope scalability demonstration for bulk-synchronized PDE stencil computations is sufficiently simple and compelling to repeat here. The crucial observation is that both explicit and implicit PDE solvers periodically cycle between compute-intensive and communicate-intensive phases, making up one macro iteration.

Given complexity estimates of the leading terms of the concurrent computation (per iteration phase), the concurrent communication, and the synchronization frequency; and a model of the architecture including the internode communication (network topology and protocol reflecting horizontal memory structure) and the on-node computation (effective performance parameters including vertical memory structure); one can formulate optimal concurrency and optimal execution time estimates, on per-iteration basis or overall (by taking into account any granularity-dependent convergence rate).

For three-dimensional simulation computation costs (per iteration) assume and idealized cubical domain with:

- $n$ grid points in each direction, for total work $N = \mathcal{O}(n^3)$
- $p$ processors in each direction, for total processors $P = \mathcal{O}(p^3)$
- execution time per iteration, $An^3/p^3$ (where $A$ 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), $Bn^2/p^2$
- global reductions at a cost of $C \log p$ or $Cp^{1/d}$ (where $C$ includes synchronization frequency as well as other topology-independent factors)
- $A$, $B$, $C$ are all expressed in the same dimensionless units, for instance, multiples of the scalar floating point multiply-add.

For the tree-based global reductions with a logarithmic cost, we have a total wall-clock time per iteration of

$$T(n,p) = A\frac{n^3}{p^3} + B\frac{n^2}{p^2} + C\log p.$$

For optimal $p$, $\frac{\partial T}{\partial p} = 0$, or $-3A\frac{n^3}{p^4} - 2B\frac{n^2}{p^3} + \frac{C}{p} = 0$, or (with $\theta \equiv \frac{32 \cdot B^3}{243 \cdot A^2 C}$),

$$p_{opt} = \left(\frac{3A}{2C}\right)^{1/3} \left( \left[1 + (1 - \sqrt{\theta})\right]^{1/3} + \left[1 - (1 - \sqrt{\theta})\right]^{1/3} \right) \cdot n.$$

This implies that the number of processors along each dimension, $p$, can grow with $n$ without any "speeddown" effect. The optimal running time is

$$T(n, p_{\text{opt}}(n)) = \frac{A}{\rho^3} + \frac{B}{\rho^2} + C\log(\rho n),$$

where $\rho = \left(\frac{3A}{2C}\right)^{1/3} \left( \left[1 + (1 - \sqrt{\theta})\right]^{1/3} + \left[1 - (1 - \sqrt{\theta})\right]^{1/3} \right)$. In limit of global reduction costs dominating nearest-neighbor costs, $B/C \to 0$, leading to

$$p_{opt} = (3A/C)^{1/3} \cdot n,$$

and

$$T(n, p_{\text{opt}}(n)) = C \left[ \log n + \frac{1}{3} \log \frac{A}{C} + const. \right].$$

We observe the direct proportionality of execution time to synchronization cost times frequency, $C$. This analysis is on a per iteration basis; fuller analysis would multiply this cost by an iteration count estimate that generally depends upon $n$ and $p$; see [7]. It shows that an arbitrary factor of performance can be gained by following processor number with increasing problem sizes. Many multiple-scale applications of high-end PDE simulation (e.g., direct Navier-Stokes at high Reynolds numbers) can absorb all conceivable boosts in discrete problem size thus made available, yielding more and more science along the way.

The analysis above is for a memory-scaled problem; however, even a fixed-size PDE problem can exhibit excellent scalability over reasonable ranges of $P$, as shown in Fig. 2 from [4].

## 4 Source #2: More Efficient Use of Faster Processors

Current low efficiencies of sparse codes can be improved if regularity of reference is exploited through memory-assist features. PDEs have a simple, periodic workingset structure that permits effective use of prefetch/dispatch directives. They also have lots of slackness (process concurrency in excess of hardware concurrency). Combined with processor-in-memory (PIM) technology for efficient memory gather/scatter to/from densely used cache-block transfers, and also with multithreading for latency that cannot be amortized by sufficiently large block transfers, PDEs can approach full utilization of processor cycles. However, high bandwidth is critical, since many PDE algorithms do only $\mathcal{O}(N)$ work for $\mathcal{O}(N)$ gridpoints' worth of loads and stores.

Through these technologies, one to two orders of magnitude can be gained by first catching up to today's clocks, and then by following the clocks into the few-GHz range.

### 4.1 PDE Workingsets

The workingset is a time-honored notion in the analysis of memory system performance [2]. Parallel PDE computations have a smallest, a largest, and a spectrum of intermediate workingsets. The smallest is the set of unknowns, geometry

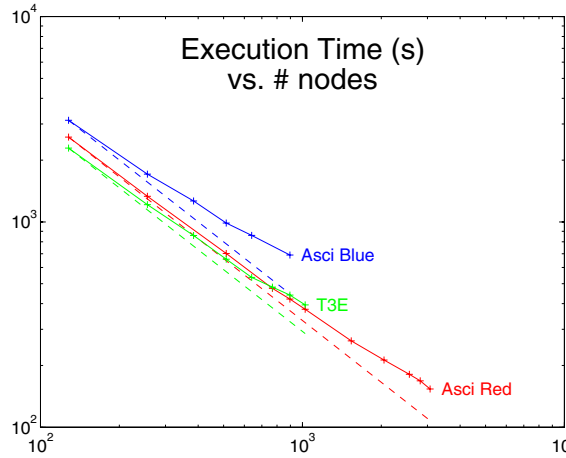


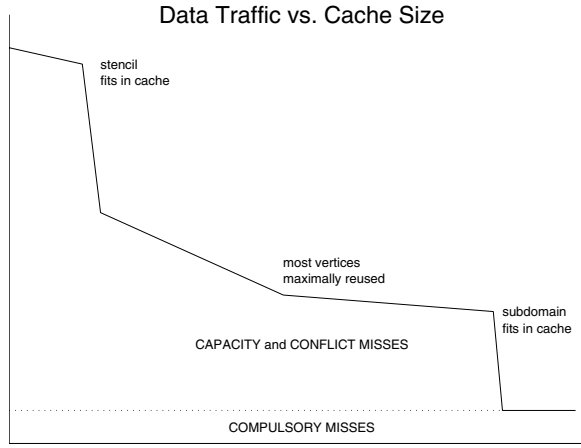

**Fig. 2.** Log-log plot of execution time vs. processor number for a full Newton-Krylov-Schwarz solution of an incompressible Euler code on two ASCI machines and a large T3E, up to at least 768 nodes each, and up to 3072 nodes of ASCI Red.

data, and coefficients at a multicomponent stencil. Its size is $N_s \cdot (N_c^2 + N_c + N_a)$ (relatively sharp). The largest is the set of unknowns, geometry data, and coefficients in an entire subdomain. Its size is $(N_x/P) \cdot (N_c^2 + N_c + N_a)$ (also relatively sharp) The intermediate workingsets are the data in neighborhood collections of gridpoints/cells that are reused within neighboring stencils.

As successive workingsets "drop" into a level of memory, capacity misses, (and with effort conflict misses) disappear, leaving only the one-time compulsory misses (see Fig. 3).

Architectural and coding strategies can be based on workingset structure. There is no performance value in any memory level with capacity larger than what is required to store all of the data associated with a subdomain. There is little performance value in memory levels smaller than the subdomain but larger than required to permit full reuse of most data within each subdomain subtraversal (middle knee, Fig. 3). After providing an L1 cache large enough for smallest workingset (associated with a stencil), and multiple independent stencils up to desired level of multithreading, all additional resources should be invested in a large L2 cache. The L2 cache should be of write-back type and its population under *user* control (e.g., prefetch/dispatch directives), since it is easy to determine when a data element is fully used within a given mesh sweep. Since this information has persistence across many sweeps it is worth determining and exploiting it. Tables describing grid connectivity are built (after each grid rebalancing) and stored in PIM. This meta-data is used to pack/unpack densely-used cache lines during subdomain traversal.

**Fig. 3.** Thought experiment for cache traffic for PDEs, as function of the size of the cache, from small (large traffic) to large (compulsory miss traffic only), showing knees corresponding to critical workingsets.




The left panel of Fig. 4 shows a set of twenty vertices in a 2D airfoil grid in the lower left portion of the domain whose working data are supposed to fit in cache simultaneously. In the right panel, the window has shifted in such a way that a majority of the points left behind (all but those along the upper boundary) are fully read (multiple times) and written (once) for this sweep. This is an unstructured analog of compiler "tiling" a regular multidimensional loop traversal. This corresponds to the knee marked "most vertices maximally reused" in Fig 3.

To illustrate the effects of reuse of a different type in a simple experiment that does not require changing cache sizes or monitoring memory traffic, we provide in Table 2 some data off three different machines for incompressible and compressible Euler simulation, from [6]. The unknowns in the compressible case are organized into $5 \times 5$ blocks, whereas those in the compressible case are organized into $4 \times 4$ blocks, by the nature of the physics. Data are intensively reused within a block, especially in the preconditioning phase of the algorithm. This boosts the overall performance by 7–10%.

The cost of greater per-processor efficiency arranged in these ways is the programming complexity of managing data traversals, the space to store gather/scatter tables in PIM, and the time to rebuild these tables when the mesh or physics changes dynamically.

**Fig. 4.** An unstructured analogy to the compiler optimization of "tiling" for a block of twenty vertices (*courtesy of D. Mavriplis*).

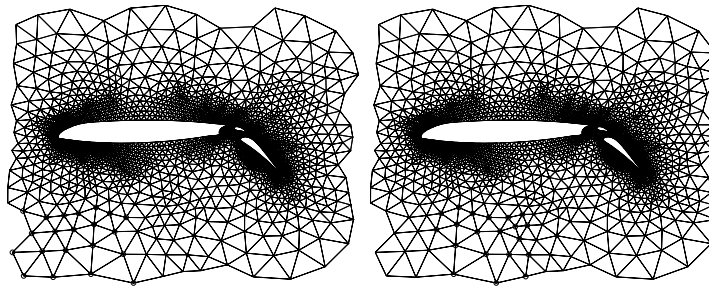


**Table 2.** Mflop/s per processor for highly L1- and register-optimized unstructured grid Euler flow code. Per-processor utilization is only 8% to 27% of peak. Slightly higher figure for compressible flow reflects larger number of components coupled densely at a gridpoint.

| | Origin 2000 | | SP | | T3E-900 | |
|---|---|---|---|---|---|---|
| Processor | R10000 | | P2SC (4-card) | | Alpha 21164 | |
| Instr. Issue | 2 | | 4 | | 2 | |
| Clock (MHz) | 250 | | 120 | | 450 | |
| Peak Mflop/s | 500 | | 480 | | 900 | |
| Application | Incomp. | Comp. | Incomp. | Comp. | Incomp. | Comp. |
| Actual Mflop/s | 126 | 137 | 117 | 124 | 75 | 82 |
| Pct. of Peak | 25.2 | 27.4 | 24.4 | 25.8 | 8.3 | 9.1 |

## 5 Source #3: More Architecture-Friendly Algorithms

Algorithmic practice needs to catch up to architectural demands. Several "one-time" gains remain that could improve data locality or reduce synchronization frequency, while maintaining required concurrency and slackness. "One-time" refers to improvements by small constant factors, nothing that scales in $N$ or $P$. Complexities are already near their information-theoretic lower bounds for some problems, and we reject increases in flop rates that derive from *less* efficient algorithms, as defined by parallel execution time. These remaining algorithmic performance improvements may cost extra memory or they may exploit shortcuts of numerical stability that occasionally backfire, making performance modeling less predictable However, perhaps as much as an order of magnitude of performance remains here.

Raw performance improvements from algorithms include data structure reorderings that improve locality, such as interlacing of all related grid-based data structures and ordering gridpoints and grid edges for L1/L2 reuse. Dis-

cretizations that improve locality include such choices as higher-order methods (which lead to denser couplings between degrees of freedom than lower-order methods) and vertex-centering (which, for the same tetrahedral grid, leads to denser blockrows than cell-centering, since there are many more than four nearest neighbors). Temporal reorderings that improve locality include block vector algorithms (these reuse cached matrix blocks; the vectors in the block are independent) and multi-step vector algorithms (these reuse cached vector blocks; the vectors have sequential dependence).

Temporal reorderings may also reduce the synchronization penalty but usually at a threat to stability. Synchronization *frequency* may be reduced by deferred orthogonalization and pivoting and speculative step selection. Synchronization *range* may be reduced by replacing a tightly coupled global process (e.g., Newton) with loosely coupled sets of tightly coupled local processes (e.g., Schwarz).

Precision reductions make bandwidth seem larger. Lower precision representation in memory of preconditioner matrix coefficients or other poorly known data causes no harm to algorithmic convergence rates, as along as the data are expanded to full precision before arithmetic is done on them, after they are in the CPU.

As an illustration of the effects of spatial reordering, we show in Table 3 from [4] the Mflop/s per processor for processors in five families based on three versions of an unstructured Euler code: the original F77 vector version, a version that has been interlaced so that all data defined at a gridpoint is stored near-contiguously, and after a vertex reordering designed to maximally reuse edge-based data. Reordering yields a factor of 2.6 (Pentium II) up to 7.5 (P2SC) on this the unstructured grid Euler flow code.

**Table 3.** Improvements from spatial reordering: uniprocessor Mflop/s, with and without optimizations.

| Processor | clock | Interlacing, Edge Reord. | Interlacing (only) | Original |
|---|---|---|---|---|
| R10000 | 250 | 126 | 74 | 26 |
| P2SC (2-card) | 120 | 97 | 43 | 13 |
| Alpha 21164 | 600 | 91 | 44 | 33 |
| Pentium II (Linux) | 400 | 84 | 48 | 32 |
| Pentium II (NT) | 400 | 78 | 57 | 30 |
| Ultra II | 300 | 75 | 42 | 18 |
| Alpha 21164 | 450 | 75 | 38 | 14 |
| 604e | 332 | 66 | 34 | 15 |
| Pentium Pro | 200 | 42 | 31 | 16 |

## 6 Source #4: Algorithms Delivering More "Science per Flop"

Some algorithmic improvements do not improve flop rate, but lead to the same scientific end in reduced time or at lower hardware cost. They achieve this by requiring less memory and fewer operations than other methods, usually through some form of adaptivity. Such adaptive programs are more complicated and less thread-uniform than those they improve upon in quality/cost ratio. It is desirable that petaflop/s machines be "general purpose" enough to run the "best" algorithms. This is not daunting conceptually, but it puts an enormous premium on dynamic load balancing. An order of magnitude or more in execution time can be gained here for many problems.

Adaptivity in PDEs is usually through of in terms of spatial discretization. Discretization type and order are varied to attain the required accuracy in approximating the continuum everywhere without over-resolving in smooth, easily approximated regions. Fidelity-based adaptivity changes the continuous formulation to accommodate required phenomena everywhere without enriching in regions where nothing happens. A classical aerodynamics example is a full potential model in the farfield coupled to a boundary layer near no-slip surfaces. Stiffness-based adaptivity changes the solution algorithm to provide more powerful, robust techniques in regions of space-time where the discrete problem is linearly or nonlinearly "stiff," without extra work in nonstiff, locally well-conditioned regions

Metrics and procedures for effective adaptivity strategies are well developed for some discretization techniques, such as method-of-lines to stiff initial-boundary value problems in ordinary differential equations and differential algebraic systems and finite element analysis for elliptic boundary value problems. It is fairly wide open otherwise. Multi-model methods are used in *ad hoc* ways in numerous commercially important engineering codes, e.g., Boeing's TRANAIR code [8]. Polyalgorithmic solvers have been demonstrated in principle, but rarely in the "hostile" environment of high-performance multiprocessing. Sophisticated software approaches (e.g., object-oriented programming) make advanced adaptivity easier to manage. Advanced adaptivity may require management of hierarchical levels of synchronization — within a region or between regions. User-specification of hierarchical priorities of different threads may be required — so that critical-path computations can be given priority, while subordinate computations fill unpredictable idle cycles with other subsequently useful work.

To illustrate the opportunity for localized algorithmic adaptivity, consider the steady-state shock simulation described in Fig. 5 from [1]. During the period between iterations 3 and 15 when the shock is moving slowly into position, only problems in local subdomains near the shock need be solved to high accuracy. To bring the entire domain into adjustment by solving large, ill-conditioned linear algebra problems for every minor movement of the shock on each Newton iteration is wasteful of resources. An algorithm that adapts to nonlinear stiffness would seek to converge the shock location before solving the rest of the subdomain with high resolution or high algebraic accuracy.

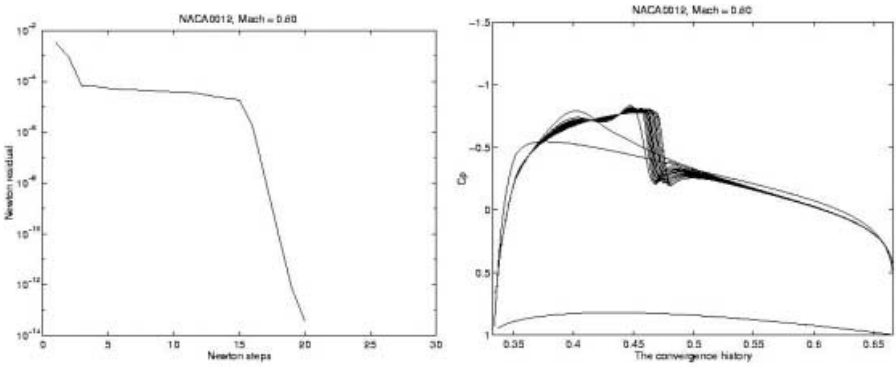



**Fig. 5.** Transonic full potential flow over NACA airfoil, showing (left) residual norm at each of 20 Newton iterations, with a plateau between iterations 3 and 15, and (right) shock developing and creeping down wing until "locking" into location at iteration 15, while the rest of flow field is "held hostage" to this slowly converging local feature.

## 7 Summary of Performance Improvements

We conclude by summarizing the types of performance improvements that we have described and illustrated on problems that can be solved on today's terascale computers or smaller. In reverse order, together with the possible performance factors available, they are:

- Algorithms that deliver more "science per flop"
  - possibly large problem-dependent factor, through adaptivity (but we won't count this towards rate improvement)
- Algorithmic variants that are more architecture-friendly
  - expect **half** an order of magnitude, through improved locality and relaxed synchronization
- More efficient use of processor cycles, and faster processor/memory
  - expect **one-and-a-half** orders of magnitude, through memory-assist language features, PIM, and multithreading
- Expanded number of processors
  - expect **two** orders of magnitude, through dynamic balancing and extreme care in implementation

Extreme concurrency — one hundred thousand heavyweight processes and a further factor of lightweight threads — are necessary to fully exploit this aggressive agenda. We therefore emphatically mention that PDEs do not arise individually from within a vacuum. Computational engineering is not about individual large-scale analyses, done fast and well-resolved and "thrown over the wall." Both results of an analysis and their sensitivities are desired. Often multiple operation points for a system to be simulated are known *a priori*, rather

than sequentially. The sensitivities may be fed back into an optimization process constrained by the PDE analysis. Full PDE analyses may also be inner iterations in a multidisciplinary computation. In such contexts, "petaflop/s" may mean 1,000 analyses running somewhat asynchronously with respect to each other, each at 1 Tflop/s. This is clearly a less daunting challenge and one that has better synchronization properties for exploiting such resources as "The Grid" than one analysis running at 1 Pflop/s.

As is historically the case, the high end of scientific computing will drive technology improvements across the entire information technology spectrum. This is ultimately the most compelling reason for pushing on through the next four orders of magnitude.

## Acknowledgements

The author would like to thank his direct collaborators on computational examples reproduced in this chapter from earlier published work: Kyle Anderson, Satish Balay, Xiao-Chuan Cai, Bill Gropp, Dinesh Kaushik, Lois McInnes, and Barry Smith. Ideas and inspiration for various in various sections of this article have come from discussions with Shahid Bokhari, Rob Falgout, Paul Fischer, Kyle Gallivan, Liz Jessup, Dimitri Mavriplis, Alex Pothen, John Salmon, Linda Stals, Bob Voigt, David Young, and Paul Woodward. Computer resources have been provided by DOE (Argonne, Lawrence Livermore, NERSC, and Sandia), and SGI-Cray.

## References

1. X.-C. Cai, W. D. Gropp, D. E. Keyes, R. G. Melvin and D. P. Young, Parallel Newton-Krylov-Schwarz Algorithms for the Transonic Full Potential Equation, SIAM J. Scientific Computing, **19**:246–265, 1998.
2. P. Denning, The Working Set Model for Program Behavior, Commun. of the ACM, **11**:323–333, 1968.
3. J. Dongarra, H.-W. Meuer, and E. Stohmaier, Top 500 Supercomputer Sites, `http://www.netlib.org/benchmark/top500.html`, June 2000.
4. W. D. Gropp, D. K. Kaushik, D. E. Keyes and B. F. Smith, Achieving High Sustained Performance in on Unstructured Mesh CFD Application, Proc. of Supercomputing'99 (CD-ROM), IEEE, Los Alamitos, 1999.
5. J. L. Gustafson, Re-evaluating Amdahl's Law, Commun. of the ACM **31**:532–533, 1988.
6. D. K. Kaushik, D. E. Keyes and B. F. Smith, NKS Methods for Compressible and Incompressible Flows on Unstructured Grids, Proc. of the 11th Intl. Conf. on Domain Decomposition Methods, C.-H. Lai, et al., eds., pp. 501–508, Domain Decomposition Press, Bergen, 1999.
7. D. E. Keyes, How Scalable is Domain Decomposition in Practice?, Proc. of the 11th Intl. Conf. on Domain Decomposition Methods, C.-H. Lai, et al., eds., pp. 282–293, Domain Decomposition Press, Bergen, 1999.

8. D. P. Young, R. G. Melvin, M. B. Bieterman, F. T. Johnson, S. S. Samant and J. E. Bussoletti, A Locally Refined Rectangular Grid Finite Element Method: Application to Computational Fluid Dynamics and Computational Physics, J. Computational Physics **92**:1–66, 1991.