

# The Next Four Orders of Magnitude in Performance for Parallel CFD

D. E. Keyes<sup>a\*</sup>

<sup>a</sup>Mathematics & Statistics Department, Old Dominion University, Norfolk, VA 23529, ISCR, Lawrence Livermore National Laboratory, Livermore, CA 94551, and ICASE, NASA Langley Research Center, Hampton, VA 23681, [keyes@icase.edu](mailto:keyes@icase.edu).

While some simulations 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. We briefly review the algorithmic structure of typical PDE-based CFD codes that is responsible for this situation and consider possible architectural and algorithmic sources for performance improvement towards the achievement of the remaining four orders of magnitude required to reach 1 Petaflop/s.

## 1. INTRODUCTION

A 1 Teraflop/s computer today could be built with as a few as 1,000 processors of 1 Gflop/s (peak) each, but due to inefficiencies within the processors, is more practically, but still optimistically, characterized as about 4,000 processors of 250 Mflop/s (delivered) each. To get to 1 Petaflop/s, we could go wider (1,000,000 processors of 1 Gflop/s each) or mainly deeper (10,000 processors of 100 Gflop/s each), with similar or larger safety factors to accommodate for processor inefficiency. From the point of view of PDE simulations on Eulerian grids, either extreme of a 1 Petaflop/s machine should be interesting.

Many of the “Grand Challenges” of HPCC, ASCI, and SSI are formulated as PDEs (possibly among alternative formulations); however, PDE simulations have struggled to hold their own among recent Bell Prize submissions, as they require a balance among architectural components that is not necessarily met in a machine designed to “max out” on the standard LINPACK benchmark. Until recently, CFD has successfully competed against applications with more intensive data reuse only on special-purpose machines (vector or SIMD) in statically discretized, explicit formulations.

PDEs come in many varieties and complexities, but though their mathematical properties differ greatly, their computational implementations are surprisingly similar, whether of evolution (e.g., time hyperbolic, time parabolic) or equilibrium (e.g, elliptic, spatially hyperbolic or parabolic) type. Memory estimates, in words,  $M \approx N_x \cdot (N_c + N_a + N_c^2 \cdot N_s)$ , and work estimates, in flops,  $W \approx N_x \cdot N_t \cdot (N_c + N_a + N_c^2 \cdot (N_c + N_s))$  hold for many types of grid-based CFD simulations, where  $N_x$  is the number of spatial grid points,  $N_t$  is the

---

\*Supported in part by the NSF under grant ECS-9527169, by NASA under contracts NAS1-19480 and NAS1-97046, by Argonne National Laboratory under contract 982232402, and by Lawrence Livermore National Laboratory under subcontract B347882.

number of temporal or iterative steps,  $N_c$  is components per point,  $N_a$  represents auxiliary storage per point, and  $N_s$  is the number of grid points in the discretization stencil. The last terms in the expressions for  $M$  and  $W$  are for implicit methods, for the storage and application of a preconditioner. Semi-implicit operator-split and explicit methods allow some reductions.

Whether for explicit or implicit methods, resource scaling for PDEs is similar: for 3D problems,  $\text{Work} \propto (\text{Memory})^{4/3}$ . In equilibrium problems, work scales with problem size times the number of iteration steps. For reasonably tolerable implicit methods the latter is proportional to the resolution in a single spatial dimension. For evolutionary problems, work scales with problem size times the number of time steps, and CFL-type arguments place the latter on order of the resolution in a single spatial dimension. While the exponent  $4/3$  is difficult to bring down for general CFD problems, 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. Four tasks are typical in grid-based PDE solvers: (1) vertex-based loops (resp., cell-based, for cell-centered storage) for state vector and auxiliary vector updates; (2) edge-based “stencil op” loops (resp., dual edge-based) for residual evaluation, approximate Jacobian evaluation, and Jacobian-vector product; (3) sparse, narrow-band recurrences, as in approximate factorization and back substitution; and (4) vector inner products and norms for orthogonalization/conjugation and convergence and stability checks.

Let  $N = N_x \cdot N_c$  be the number of unknowns and  $P$  the number of processors. Then, for explicit solvers of the generic form

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

concurrency is pointwise,  $\mathcal{O}(N)$ ; the communication-to-computation ratio is surface-to-volume,  $\mathcal{O}\left(\left(\frac{N}{P}\right)^{-1/3}\right)$ ; the communication range is nearest-neighbor, except for time-step computation; and the synchronization frequency is once per time-step,  $\mathcal{O}\left(\left(\frac{N}{P}\right)^{-1}\right)$ . For domain-decomposed implicit solvers of the form

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

concurrency is either pointwise,  $\mathcal{O}(N)$ , or subdomainwise,  $\mathcal{O}(P)$ ; the comm.-to-comp. ratio still *mainly* surface-to-volume,  $\mathcal{O}\left(\left(\frac{N}{P}\right)^{-1/3}\right)$ ; and the communication is still *mainly* nearest-neighbor. However, convergence checking, orthogonalization/conjugation steps, and hierarchically coarsened problems add nonlocal communication. The synchronization frequency is *higher* – often more than once per grid-sweep, up to Krylov subspace dimension,  $\mathcal{O}\left(K\left(\frac{N}{P}\right)^{-1}\right)$ . Storage per point is *also higher*, by factor of  $\mathcal{O}(K)$ . Load balance issues are the same as for explicit — routine if the grid is static, and highly challenging in the dynamic adaptive case. In this chapter, we assume grids are quasi-static.

## 2. SOURCE #1: EXPANDED NUMBER OF PROCESSORS

A simple bulk-synchronous scaling argument suggests that continued expansion of the number of processors is feasible as long as the architecture provides a global reduction

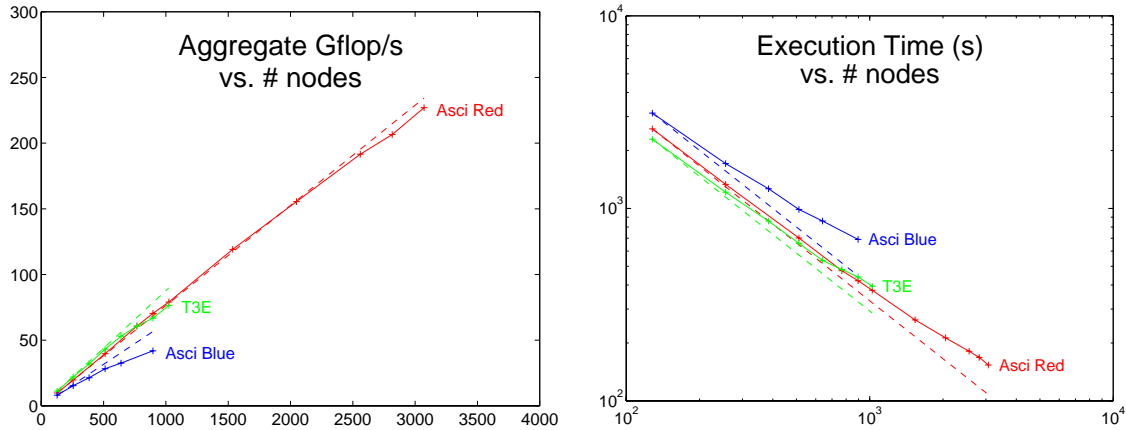


Figure 1. Flop rates (left) and execution time reduction (right) for an Euler flow problem on three machines. Dashed lines show the ideal for each performance curve, based on a left endpoint of 128 processors.

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.

As popularized with the 1986 Karp Prize entry of Benner, Gustafson & Montry, Amdahl’s law can be defeated if serial (or bounded concurrency) sections make up a decreasing fraction of total work as problem size and processor count scale — true for most explicit or iterative implicit PDE solvers. A simple, back-of-envelope parallel complexity analysis [4] shows that processors can be increased as fast, or almost as fast, as problem size, assuming load is perfectly balanced. An important caveat relative to Beowulf-type systems is that the processor network must also be scalable (in protocols as well as in hardware). Therefore, *all* remaining four orders of magnitude could be met by hardware expansion. However, this does *not* mean that fixed-size applications of today would run 10<sup>4</sup> times faster; these Gustafson-type analyses are for problems that are correspondingly larger.

As encouraging evidence that even fixed-size CFD problems scale well into thousands of processors, we reproduce from [1], in Fig. 1, flop/s scaling and execution time curves for Euler flow over an ONERA M6 Wing, on a tetrahedral grid of 2.8 million vertices, run on up to 1024 processors of a 600 MHz T3E, 768 processors of IBM’s ASCI Blue Pacific, and 3072 dual-processor nodes of Intel’s ASCI Red. (The execution rate scales better than the execution time for this fixed-size problem, since as the subdomains get smaller with finer parallel granularity more of the work is redundant, on ghost regions.)

### 3. SOURCE #2: MORE EFFICIENT USE OF FASTER PROCESSORS

Looking internal to a processor, we argue that there are only two intermediate levels of the memory hierarchy that are essential to a typical domain-decomposed PDE simu-

Table 1

Flop/s rate and percent utilization, as a function of dense point-block size, which varies in “Incomp.” and “Comp.” formulations.

	Origin 2000		SP		T3E-900	
Processor	R10000		P2SC (4-card)		Alpha 21164	
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

lation, 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 schedule is periodic and predictable. 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.

Improvement of the low efficiencies of most current sparse codes through regularity of reference is an active area of research that yields strong dividends for PDEs. PDEs have a simple, periodic workingset structure that permits effective use of prefetch/dispatch directives, and they have a luxurious amount of “slackness” (potential process concurrency in excess of hardware concurrency). Combined with intelligent processors-in-memory (PIM) features to do gather/scatter cache transfers and multithreading for latency that cannot be amortized by sufficiently large block transfers, PDEs can approach full utilization of processor cycles. An important architectural caveat is that high bandwidth is critical to support these other advanced features, since PDE algorithms do only  $\mathcal{O}(N)$  work for  $\mathcal{O}(N)$  gridpoints worth of loads and stores.

One to two orders of magnitude can be gained by catching up to the clock, through such advanced features, and by following the clock into the few-GHz range. Even without PIM, multithreading, and bandwidth (in words per second) equal to the processor clock rate times the superscalarity, one can see the advantage in blocking in the comparisons in Table 1. For the same Euler flow system considered above, the problem was run incompressibly (with  $4 \times 4$  blocks at each point) and compressibly (with  $5 \times 5$  blocks at each point). On three different architectures, this modest improvement in reuse of cached data leads to a corresponding improvement in efficiency.

We briefly consider the workingsets that are relevant to PDE solvers. The smallest consists of the unknowns, geometry data, and coefficients at a single multicomponent stencil, of size  $N_s \cdot (N_c^2 + N_c + N_a)$ . The largest consists of the unknowns, geometry

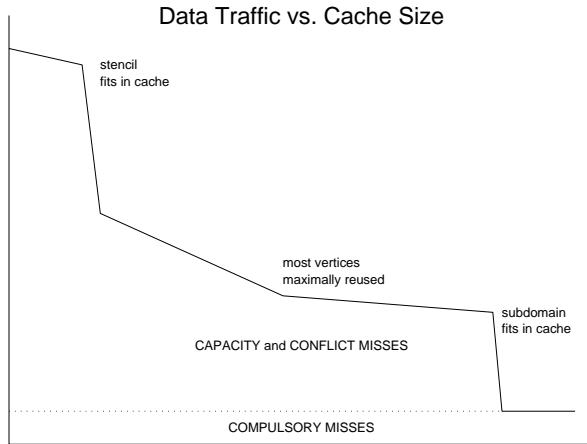


Figure 2. Idealized model of cache traffic for fixed computation as cache size increases, showing two extreme knees and one gradual “knee.”

data, and coefficients in an entire subdomain, of size  $(N_x/P) \cdot (N_c^2 + N_c + N_a)$ . Most practical caches will be sized in between these two. The critical workingset to consider in relation to cache size is the intermediate one of the data in neighborhood collection of gridpoints/cells that is reused when the group of corresponding neighboring stencils is updated. As successive workingsets “drop” into a level of memory, capacity (and with effort conflict) misses disappear, leaving only compulsory misses, as sketched in the illustration of memory traffic generated from a fixed computation versus varying cache size in Fig. 2.

There is no performance value in memory levels larger than subdomain, and little performance value in memory levels smaller than subdomain but larger than required to permit full reuse of most data within each subdomain subtraversal (middle knee, Fig. 2). The natural strategy based on this simple workingset structure is therefore, after providing an L1 cache large enough for smallest workingset (and multiple independent copies up to desired level of multithreading, if applicable), all additional resources should be invested in large L2. Furthermore, L2 should be of write-back type and its population should be under user-assist with prefetch/dispatch directives. Tables describing grid connectivity should be built (within each quasi-static grid phase) and stored in PIM — used to pack/unpack dense-use cache lines during subdomain traversal.

The costs of this greater per-processor efficiency are the programming complexity of managing the subdomain traversal, the space to store the gather/scatter tables in PIM, the time to (re)build the gather/scatter tables, and the memory bandwidth commensurate with peak rates of all processors. Unfortunately, current shared-memory machines have disappointing memory bandwidth for PDEs; the extra processors beyond the first sharing a memory port are often not useful.

Table 2

Experiments in grid edge reordering and data structure interlacing on various uniprocessors for the Euler flow problem. In Mflop/s. Final column shows relative speedups.

Processor	clock (MHz)	Interlacing, Edge Reord.	Interlacing (only)	Original	Speedup
P2SC (2-card)	120	97	43	13	7.5
R10000	250	126	74	26	4.8
604e	332	66	34	15	4.4
Ultra II	300	75	42	18	4.2
Alpha 21164	600	91	44	33	2.8
Pentium II (Linux)	400	84	48	32	2.6

#### 4. SOURCE #3: MORE ARCHITECTURE-FRIENDLY ALGORITHMS

Besides the two just considered 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 this section, 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.

Algorithmic practice needs to catch up to architectural demands, and several “one-time” gains remain to be contributed 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 information-theoretic lower bounds for some CFD solvers, and we reject increases in flop rates that derive from *less* efficient algorithms, as defined by parallel execution time. A caveat here is that the remaining algorithmic performance improvements may cost extra space or may bank on stability shortcuts that occasionally backfire, making performance modeling less predictable. Perhaps an order of magnitude of performance remains here.

Raw performance improvement from algorithms include: (1) spatial reorderings that improve locality, such as interlacing of all related grid-based data structures and ordering gridpoints and grid edges for L1/L2 reuse; (2) discretizations that improve locality, such as higher-order methods (which lead to larger denser blocks at each point than lower-order methods) and vertex-centering (which, for the same tetrahedral grid, leads to denser blockrows than cell-centering); (3) temporal reorderings that improve locality, such as block vector algorithms (these reuse cached matrix blocks; vectors in block are independent), and multi-step vector algorithms (these reuse cached vector blocks; vectors have sequential dependence); (4) temporal reorderings that reduce synchronization penalty, such as less stable algorithmic choices that reduce synchronization *frequency* (deferred orthogonalization, speculative step selection) and less global methods that reduce synchronization *range* by replacing a tightly coupled global process (e.g., Newton) with loosely coupled sets of tightly coupled local processes (e.g., Schwarz); and (5) precision

reductions that make memory bandwidth seem larger, such as lower precision representation of preconditioner matrix coefficients or poorly known coefficients (arithmetic is still performed on full precision extensions).

Table 2 (from data in [1]) shows some experimental improvements from spatial reordering on the same unstructured-grid Euler flow problem described earlier.

## 5. SOURCE #4: ALGORITHMS PACKING MORE “SCIENCE PER FLOP”

It can be argued that this last 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 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 equivalent to less powerful but more versatile platforms. The danger of such a pyrrhic victory is real.

Some algorithmic improvements do not improve flop rate, but lead to the same scientific end in the same time at lower hardware cost (less memory, lower operation complexity). A caveat here is that such adaptive programs are more complicated and less thread-uniform than those they improve upon in quality/cost ratio. They are not daunting, conceptually, but they put an enormous premium on dynamic load balancing. An order of magnitude or more can be gained here for many problems.

Some examples of adaptive opportunities are: (1) spatial discretization-based adaptivity, in which discretization type and order are varied to attain required approximation to the continuum everywhere without over-resolving in smooth, easily approximated regions; (2) fidelity-based adaptivity, in which the continuous formulation is varied to accommodate physical complexity without enriching physically simple regions; and (3) “stiffness”-based adaptivity, in which the solution algorithm is changed to provide more powerful, robust techniques in regions of space-time where discrete problem is linearly or nonlinearly stiff, without extra work in nonstiff, locally well-conditioned regions.

What are the status and prospects for such advanced adaptivity? Appropriate metrics to govern the adaptivity and procedures to exploit them are already well developed for some discretization techniques, including method-of-lines ODE solutions to stiff IBVPs and DAEs, and FEA for elliptic BVPs. This field is fairly wide open for other types of numerical analyses. Fidelity-based multi-model methods have been used in *ad hoc* ways in numerous commercially important engineering codes, e.g., Boeing TRANAIR [5]. Polyalgorithmic solvers have been demonstrated in principle e.g., [3], but rarely in the “hostile” environment of high-performance multiprocessing. These advanced adaptive approaches demand sophisticated software approaches, such as object-oriented programming. Management of hierarchical levels of synchronization (within a region and between regions) is also required. User-specification of hierarchical priorities of different threads would also

be desirable — so that critical-path computations can be given priority, while subordinate computations fill up unpredictable idle cycles with other subsequently useful work.

An experimental example of new opportunities for localized algorithmic adaptivity is described surrounding Figs. 5 and 6 in [2]. For transonic full potential flow over a NACA airfoil, solved with Newton’s method, excellent progress in residual reduction is made for the first few steps and the last few steps. In between, a shock develops and creeps downwing until it “locks” into its final location, while the rest of flow field is “held hostage” to this slowly converging local feature, whose stabilization completely dominates execution time. Resources should be allocated differently before and after shock location stabilizes.

## 6. SUMMARY

To recap in reverse order, the performance improvement possibilities that suggest that petaflop/s is within reach for PDEs are: (1) algorithms that deliver more “science per flop” possibly large problem-dependent factor, through adaptivity (though we won’t count this towards rate improvement); (2) algorithmic variants that are more architecture-friendly, which we expect to contribute *half* an order of magnitude, through improved locality and relaxed synchronization; (3) more efficient use of processor cycles, and faster processor/memory from which we expect *one-and-a-half* orders of magnitude, through memory-assist language features, PIM, and multithreading; and (4) an expanded number of processors to which we look for the remaining *two* orders of magnitude. The latter will depend upon more research in dynamic balancing and extreme care in implementation.

## 7. ACKNOWLEDGMENTS

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. Computer resources were provided by DOE (Argonne, LLNL, NERSC, Sandia), and SGI-Cray.

## REFERENCES

1. W. K. Anderson, W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith. Achieving high sustained performance in an unstructured mesh CFD application. In *Proceedings of SC’99 (CDROM)*, November 1999.
2. 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. Sci. Comput.*, 19:246–265, 1998.
3. A. Ern, V. Giovangigli, D. E. Keyes, and M. D. Smooke. Towards polyalgorithmic linear system solvers for nonlinear elliptic systems. *SIAM J. Sci. Comput.*, 15:681–703, 1994.
4. D. E. Keyes. How scalable is domain decomposition in practice? In C.-H. Lai *et al.*, editor, *Proceedings of the 11th International Conference on Domain Decomposition Methods*, pages 286–297. Domain Decomposition Press, Bergen, 1999.
5. 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.Comp. Phys.*, 92:1–66, 1991.