Latency, bandwidth, and concurrent issue limitations in high-performance CFD

W.D. Gropp [a], D.K. Kaushik [a,*], D.E. Keyes [b], B.F. Smith [a]

[a] *Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, USA*
[b] *Mathematics and Statistics Department, Old Dominion University, Norfolk, VA 23529, USA*

Abstract

To achieve high performance, a parallel algorithm needs to effectively utilize the memory subsystem and minimize the communication volume and the number of network transactions. These issues gain further importance on modern architectures, where the peak CPU performance is increasing much more rapidly than the memory or network performance. In this paper, we present some performance enhancing techniques that were employed on an unstructured mesh implicit solver. Our experimental results show that this solver adapts reasonably well to the high memory and network latencies.

Keywords: Latency tolerance; Memory hierarchy; Memory bandwidth; Cache misses; Hybrid programming model; High-performance computing

1. Introduction and motivation

Many of the 'Grand Challenges' of computational science are formulated as partial differential equations (PDEs). PDE solvers typically perform at a computational rate well below other scientific simulations (e.g. with dense linear algebra or N-body kernels) on modern architectures with deep memory hierarchies. The primary reason for this relatively poor performance is good algorithmic efficiency in the traditional sense: low work to data size ratio, relative to clock/bandwidth ratios in contemporary microprocessors. High memory and network latencies and imbalances in superscalar architecture also play a role.

In a typical PDE computation, four basic groups of tasks can be identified, based on the criteria of arithmetic concurrency, communication patterns, and the ratio of operation complexity to data size within the task. These four distinct groups, present in most implicit codes, are vertex-based loops, edge-based loops, recurrences, and global reductions. Each of these groups of tasks stresses a different subsystem of contemporary high-performance computers. After tuning, linear algebraic recurrences run close to the aggregate memory-bandwidth limit on performance,

flux computation loops over edges are bounded either by memory bandwidth or instruction scheduling, and parallel efficiency is bounded primarily by slight load imbalances at synchronization points [2,3].

In this paper, we present some strategies that have been effective in tolerating the latencies arising from the hierarchical memory system (Section 2) and network (Section 3). We also compare the different programming models in Section 4 from a performance standpoint. Our demonstration code, PETSc-FUN3D, solves the Euler and Navier–Stokes equations of fluid flow in incompressible and compressible forms with second-order flux-limited characteristics-based convection schemes and Galerkin-type diffusion on unstructured meshes. The solution algorithm employed in PETSc-FUN3D is pseudo-transient Newton–Krylov–Schwarz (ψNKS) [5] with block-incomplete factorization on each subdomain of the Schwarz preconditioner and with varying degrees of overlap.

2. Adapting to the high memory latency

Since the gap between memory and CPU speeds is ever widening [6], it is crucial to maximally utilize the data brought into the levels of memory hierarchy that are close to the CPU. The data structures for primary (e.g. momenta

---

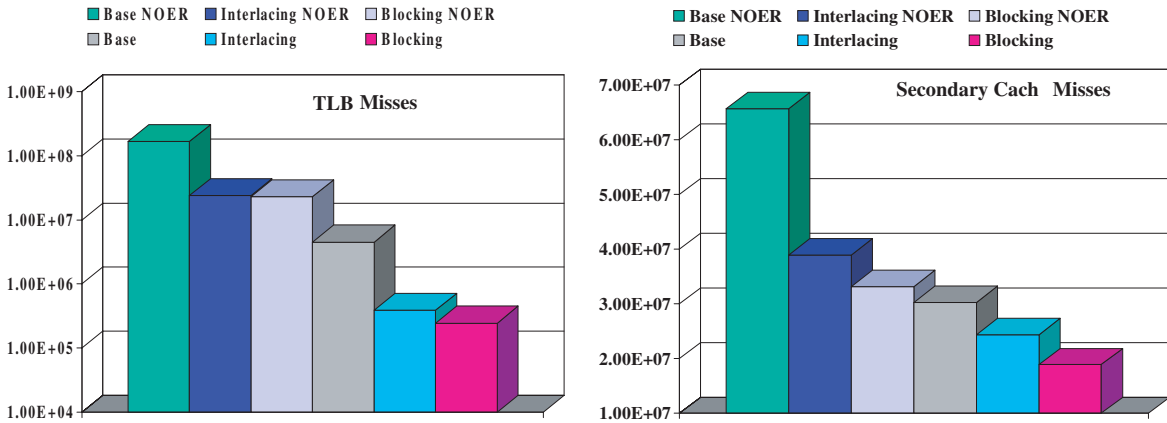*Computational Fluid and Solid Mechanics*
K.J. Bathe (Editor)

Fig. 1. TLB misses (log scale) and secondary cache misses (linear scale) on one processor of an Origin2000 for a 22,677-vertex case, showing dramatic improvements in data locality due to data ordering (grid edge reordering and field variable interlacing) and blocking techniques. ('NOER' denotes *no* edge ordering; otherwise edges are reordered by default.)

and pressure) and auxiliary (e.g. geometry and constitutive parameter) fields must be adapted to hierarchical memory. Three simple techniques have proved very useful in improving the performance of the FUN3D code, which was originally tuned for vector machines. We have used *interlacing* (creating spatial locality for the data items needed successively in time), *structural blocking* for a multicomponent system of PDEs (cutting the number of integer loads significantly, and enhancing reuse of data items in registers), and *vertex and edge reorderings* (increasing the level of temporal and spatial locality in cache). These techniques are discussed in detail in [3].

Fig. 1 shows the effectiveness of these techniques on one processor of the SGI Origin2000. We observe that the edge reordering reduces the TLB misses by two orders of magnitude, while secondary cache misses are reduced by a factor of 3.5.

Another aspect of memory hierarchy that attains importance in the computation of PDEs is the large gap between the required and the available memory bandwidths [2]. Since linear algebraic kernels run close to the available memory bandwidth, we store elements of the preconditioner for the Jacobian matrix in single-precision to improve the performance of the sparse triangular matrix solution phase. In our 'matrix-free' implementation, the Jacobian itself is never explicitly needed; see [5]. All computation with the preconditioner is still done in full (double) precision. The performance advantages are shown in Table 1, where the single-precision storage version runs at almost twice the rate of the double-precision storage version, clearly identifying memory bandwidth as the bottleneck. The number of time steps needed to converge is not affected, since the preconditioner is already very approximate by design.

3. Tolerating the network limitations

Domain-decomposed parallelism for PDEs is a natural means of overcoming Amdahl's law in the limit of fixed problem size per processor. Computational work on each evaluation of the conservation residuals scales as the volume of the (equal-size) subdomains, whereas communication overhead scales only as the surface. This ratio is fixed when problem size and processors are scaled in proportion, leaving only global reduction operations over all processors as an impediment to perfect performance scaling.

When the load is perfectly balanced (easily achieved for static meshes) and local communication is not an issue because the network is scalable, the optimal number of processors is related to the network diameter. For logarithmic networks, like a hypercube, the optimal number of processors, P, grows directly in proportion to the problem size, N. For a d-dimensional torus network, $P \propto N^{d/d+1}$. The

Table 1

Execution times (in seconds) on a 250-MHz Origin2000 for a fixed-size 357,900-vertex case with single- or double-precision storage of the preconditioner matrix

Number of processors	Computational phase			
	Linear solve		Overall	
	Double	Single	Double	Single
16	223	136	746	657
32	117	67	373	331
64	60	34	205	181
120	31	16	122	106

The results suggest that the linear solver time is bottlenecked by memory bandwidth. This conclusion is supported by analytical estimates in [2].

Table 2
Scalability bottlenecks on ASCI Red for a fixed-size 2.8M-vertex mesh

Number of processors	Its	Time (seconds)	Speedup	Efficiency		
				$\eta_{overall}$	η_{alg}	η_{impl}
128	22	2039	1.00	1.00	1.00	1.00
256	24	1144	1.78	0.89	0.92	0.97
512	26	638	3.20	0.80	0.85	0.94
1024	29	362	5.63	0.70	0.76	0.93
2048	32	208	9.78	0.61	0.69	0.89
3072	34	159	12.81	0.53	0.65	0.82

Number of processors	Percent times for			Scatter scalability	
	Global reductions	Implicit synchronizations	Ghost point scatters	Total data sent per iteration (GB)	Application level effective bandwidth per node (MB/s)
128	5	4	3	3.6	6.9
256	3	6	4	5.0	7.5
512	3	7	5	7.1	6.0
1024	3	10	6	9.4	7.5
2048	3	11	8	11.7	5.7
3072	5	14	10	14.2	4.6

The preconditioner used in these results is block Jacobi with ILU(1) in each subdomain. We observe that the principal nonscaling factor is the implicit synchronization.

proportionality constant is a ratio of work per subdomain to the product of synchronization frequency and internode communication latency.

In Table 2, we present a closer look at the relative cost of computation for PETSc-FUN3D for a *fixed-size* problem of 2.8 million vertices on the ASCI Red machine, from 128 to 3072 nodes. The intent here is to identify the factors that retard the scalability. The overall parallel efficiency (denoted by $\eta_{overall}$) is broken into two components: η_{alg} measures the degradation in the parallel efficiency due to the increased iteration count of this (non-coarse-grid-enhanced) ψNKS algorithm as the number of subdomains increases, while η_{impl} measures the degradation coming from all other nonscalable factors, such as global reductions, load imbalance (implicit synchronizations), and hardware limitations.

From Table 2, we observe that the buffer-to-buffer time for global reductions for these runs is relatively small and does not grow on this excellent network. The primary factors responsible for the increased overhead of communication are the implicit synchronizations and the ghost point updates (interprocessor data scatters).

The increase in the percentage of time (3–10%) for the scatters results more from algorithmic issues than from hardware/software limitations. With an increase in the number of subdomains, the percentage of grid point data that must be communicated also rises. For example, the total amount of nearest neighbor data that must be communicated per iteration for 128 subdomains is 3.6 gigabytes, while for 3072 subdomains it is 14.2 gigabytes. Although

more network wires are available when more processors are employed, scatter time increases. When problem size and processor count are scaled together, we expect scatter time to occupy a fixed percentage of the total time and load imbalance to be reduced at high granularity.

4. Choosing the right programming model

The performance results above are based on subdomain parallelism using the Message Passing Interface (MPI) [4]. With the availability of large-scale SMP clusters, different software models for parallel programming require a fresh assessment. For machines with physically distributed memory, MPI is a natural and successful software model. For machines with distributed shared memory and nonuniform memory access, both MPI and OpenMP have been used with respectable parallel scalability. For clusters with two or more SMPs on a single node, the mixed software model of threads within a node (OpenMP being a special case of threads because of the potential for highly efficient handling of the threads and memory by the compiler) and MPI between the nodes appears natural. Several researchers (e.g. [1,7]) have used this mixed model with reasonable success.

We investigate the mixed model by employing OpenMP in the flux calculation phase only. This phase takes over 60% of the execution time on ASCI Red and is an ideal candidate for shared-memory parallelism because it does not suffer from the memory bandwidth bottleneck. In

Table 3
Execution time (in seconds) on the 333 MHz Pentium Pro ASCI Red machine for function evaluations only for a 2.8M-vertex case, comparing the performance of the hybrid (MPI/OpenMP) and the distributed memory (MPI alone) programming models

Nodes	MPI/OpenMP threads per node		MPI processes per node	
	1	2	1	2
256	483	261	456	258
2560	76	39	72	45
3072	66	33	62	40

Table 3, we compare the performance of this phase when the work is divided by using two OpenMP threads per node with the performance when the work is divided using two independent MPI processes per node. There is no communication in this phase. Both processors work with the same amount of memory available on a node; in the OpenMP case, it is shared between the two threads, while in the case of MPI it is divided into two address spaces.

The hybrid MPI/OpenMP programming model appears to be a more efficient way to employ shared memory than that of heavyweight subdomain-based processes (MPI alone), especially when the number of nodes is large. The MPI model works with a larger number of subdomains (equal to the number of MPI processors), resulting in slower rate of convergence. The hybrid model works with fewer chunkier subdomains (equal to the number of nodes), resulting in faster convergence rate and shorter execution time, despite the fact that there is some redundant work when the data from the two threads is combined because of the lack of a vector-reduce operation in the OpenMP standard (version 1) itself.

5. Conclusions and future directions

Unstructured implicit CFD solvers are amenable to scalable implementation, but careful tuning is needed to obtain the best product of per processor efficiency and parallel efficiency. The principal nonscaling factor is implicit synchronization, not the communication itself.

For solution algorithms for systems modeled by PDEs on contemporary high-end architecture, critical research directions are: (1) less synchronous algorithms; (2) memory latency tolerant algorithms (e.g. algorithms that can reuse matrix); and (3) hybrid programming models. To influence future architectures while adapting to current ones, we recommend adoption of new benchmarks featuring implicit methods on unstructured grids, such as the application featured here.

References

[1] Bova SW, Breshears CP, Cuicchi CE, Demirbilek Z, Gabb HA. Dual-level parallel analysis of harbor wave response using MPI and OpenMP. Int J High Perform Comput Appl 2000;14:49–64.
[2] Gropp WD, Kaushik DK, Keyes DE, Smith BF. Toward realistic performance bounds for implicit CFD codes. In: Proceedings of Parallel CFD '99. Amsterdam: Elsevier, 1999, pp. 233–240.
[3] Gropp WD, Kaushik DK, Keyes DE, Smith BF. Performance Modeling and Tuning of an Unstructured Mesh CFD Application. Proceedings of SC2000, Los Alamitos, IEEE Computer Society, 2000.
[4] Gropp WD, Lusk E, Skjellum A. Using MPI: Portable Parallel Programming with the Message Passing Interface, 2nd edn. Cambridge, MA: MIT Press, 1999.
[5] Gropp WD, McInnes LC, Tidriri MD, Keyes DE. Globalized Newton–Krylov–Schwarz algorithms and software for parallel implicit CFD. Int J High Perfor Comput Appl 2000;14:102–136.
[6] Hennessy JL, Patterson DA. Computer Architecture: A Quantitative Approach. Morgan Kaufmann: 1996.
[7] Mavriplis DJ. Parallel Unstructured Mesh Analysis of High-Lift Configurations. AIAA 2000;923.