Toward Realistic Performance Bounds for Implicit CFD Codes

W. D. Gropp,[a*] D. K. Kaushik,[b†] D. E. Keyes,[c‡] and B. F. Smith[d*]

[a]Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, gropp@mcs.anl.gov.

[b]Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439 and Computer Science Department, Old Dominion University, Norfolk, VA 23529, kaushik@cs.odu.edu.

[c]Mathematics and 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.

[d]Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, bsmith@mcs.anl.gov.

The performance of scientific computing applications often achieves only a small fraction of peak performance [7,17]. In this paper, we discuss two causes of performance problems—insufficient memory bandwidth and a suboptimal instruction mix—in the context of a complete, parallel, unstructured mesh implicit CFD code. The results show that the performance of our code and of similar implicit codes is limited by the memory bandwidth of RISC-based processor nodes to as little as 10% of peak performance for some critical computational kernels. Limits on the number of basic operations that can be performed in a single clock cycle also limit the performance of "cache-friendly" parts of the code.

1. Introduction and Motivation

Traditionally, numerical analysts have evaluated the performance of algorithms by counting the number of floating-point operations. It is well known that this is not a good estimate of performance on modern computers; for example, the performance advantage of the level-2 and level-3 BLAS over the level-1 BLAS for operations that involve the same number of floating-point operations is due to better use of memory, particularly the reuse of fast memory [5,6]. Paradoxically, however, the success of the level-2 and level-3 BLAS at reaching near-peak levels of performance has obscured the difficulties

*Supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

†Supported by Argonne National Laboratory under contract 983572401.

‡Supported in part by the National Science Foundation under grant ECS-9527169, by NASA under contracts NAS1-19480 and NAS1-97046, and by Argonne National Laboratory under contract 982232402.

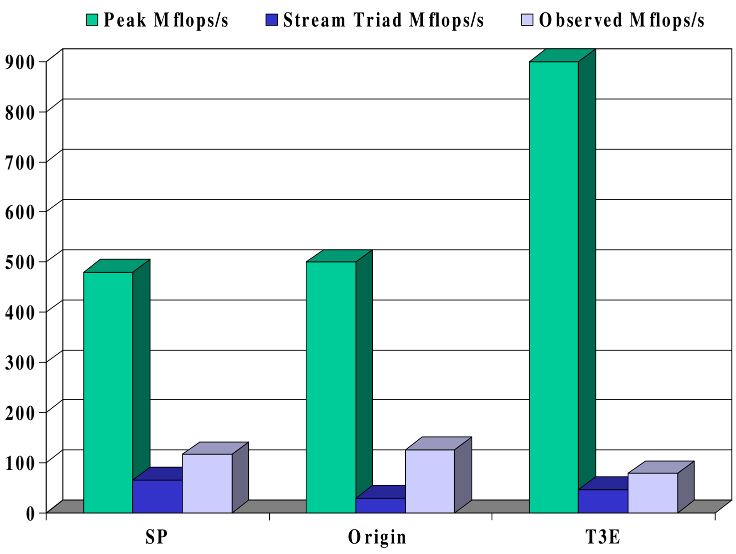

Figure 1. Sequential performance of PETSc-FUN3D for a small grid of 22,677 vertices (with 4 unknowns per vertex) run on an 120 MHz IBM SP, 250 MHz SGI Origin2000, and 450 MHz SGI/Cray T3E.

faced by many other numerical algorithms. On the algorithmic side, tremendous strides have been made; many algorithms now require only a few floating-point operations per mesh point. On the hardware side, however, memory system performance is improving at a rate that is much slower than that of processor performance [8,11]. The result is a mismatch in capabilities: algorithm design has minimized the work per data item, but hardware design depends on executing an increasing large number of operations per data item.

The importance of memory bandwidth to the overall performance is suggested by the performance results shown in Figure 1. These show the single-processor performance for our code, PETSc-FUN3D [2,9], which was originally written by W. K. Anderson of NASA Langley [1]. The performance of PETSc-FUN3D is compared with the peak performance and the results of the STREAM benchmark [11], which measures achievable performance for memory-bandwidth limited computations. The figure shows that the STREAM results are a much better indicator of performance than are the peak numbers. To illustrate the performance limitations caused by insufficient available memory bandwidth, we discuss in Section 2 a sparse matrix-vector multiply algorithm, a critical operation in many iterative methods used in implicit CFD codes.

Even for computations that are not memory intensive, computational rates often fall far short of peak performance. This is true for the flux computation in our code, even when the code has been well tuned for cache-based architectures [10]. We show in Section 3 that instruction scheduling is a major source of the performance shortfall in the flux computation step.

This paper focuses on the per-processor performance of compute nodes used in parallel

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

Figure 2. General form of sparse matrix-vector product algorithm: storage format is AIJ or compressed row storage; the matrix has `m` rows and `nz` non-zero elements and gets multiplied with `N` vectors; the comments at the end of each line show the assembly level instructions the current statement generates, where `AT` is address translation, `Br` is branch, `Iop` is integer operation, `Fop` is floating-point operation, `Of` is offset calculation, `LD` is load, and `St` is store.

computers. Our experiments have shown that PETSc-FUN3D has good scalability [2]. In fact, since good per-processor performance reduces the fraction of time spent in computation as opposed to communication, achieving the best per-processor performance is a critical prerequisite to demonstrating uninflated parallel performance [3].

2. Performance Analysis of the Sparse Matrix-Vector Product

The sparse matrix-vector product is an important part of many iterative solvers used in scientific computing. While a detailed performance modeling of this operation can be complex, particularly when data reference patterns are included [14–16], a simplified analysis can still yield upper bounds on the achievable performance of this operation. To illustrate the effect of memory system performance, we consider a generalized sparse matrix-vector multiply that multiplies a matrix by N vectors. This code, along with operation counts, is shown in Figure 2.

2.1. Estimating the Memory Bandwidth Bound

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

For the algorithm presented in Figure 2, the matrix is stored in compressed row storage format (similar to PETSc's AIJ format [4]). For each iteration of the inner loop in Figure 2, we transfer one integer (`ja` array) and $N+1$ doubles (one matrix element and N vector elements), and we do N floating-point multiply-add (`fmadd`) operations or $2N$ flops. Finally, we store the N output vector elements. We get the following estimate of the data volume:

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

This gives us an estimate of the bandwidth required by the processor to do $2nz\ N$ flops at the peak speed:

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

Alternatively, given the memory performance, we can predict the maximum achievable performance, as follows:

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

where M_{BW} is measured in Mflops/sec and BW stands for the available memory bandwidth in Mbytes/s, as measured by the STREAM [11] benchmark. (The raw bandwidth based on memory bus frequency and width is not a suitable choice because it cannot be sustained in any application; at the same time, some applications can achieve higher bandwidth than that measured by STREAM.

In Table 1, we show the memory bandwidth required for peak performance and the achievable performance for a matrix in AIJ format with 90,708 rows and 5,047,120 nonzero entries on an SGI Origin2000 (unless otherwise mentioned, this matrix is used in all subsequent computations). The matrix is a typical Jacobian from a PETSc-FUN3D application (incompressible version) with four unknowns per vertex. The same table also shows the memory bandwidth requirement for the block storage format (BAIJ) for this matrix with a block size of four; in this format, the `ja` array is smaller by a factor of the block size. We observe that the blocking helps significantly by cutting down on the memory bandwidth requirement. Having more than one vector also requires less memory bandwidth and boosts the performance: we can multiply four vectors in about 1.5 times the time needed to multiply one vector.

2.2. Estimating the Operation Issue Limitation

To analyze this performance bound, we assume that all the data items are in primary cache (equivalent to assuming infinite memory bandwidth). Referring to the sparse matrix-vector algorithm in Figure 2, we obtain the following composition of the workload for each iteration of the inner loop:

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

Table 1
Effect of memory bandwidth on the performance of sparse matrix-vector product on SGI Origin2000 (250 MHz R10000 processor). The STREAM benchmark memory bandwidth [11] is 358 MB/s; this value is used to calculate the ideal Mflops/s. The achieved values of memory bandwidth and Mflops/s are measured by using hardware counters on this machine. Our experiments show that we can multiply four vectors in 1.5 times the time needed to multiply one vector.

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

- $N + 2$ loads and stores

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

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

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

2.3. Performance Comparison

In Figure 3, we compare three performance bounds: the peak performance based on the clock frequency and the maximum number of floating-point operations per cycle, the performance predicted from the memory bandwidth limitation in Equation 1, and the performance based on operation issue limitation in Equation 2. For the sparse matrix-vector multiply, the memory-bandwidth limit on performance clearly is a good approximation. The greatest differences between the performance observed and predicted by memory bandwidth are on the systems with the smallest caches (IBM SP and T3E), where our assumption that there are no conflict misses is likely to be invalid.

3. Flux Calculation

A complete CFD application has many computational kernels. Some of these, like the sparse matrix-vector product analyzed in Section 2, are limited in performance by the available memory bandwidth. Other parts of the code may not be limited by memory bandwidth but may still perform significantly below peak performance. In this section, we consider such a kernel in the PETSc-FUN3D code.

The flux calculation is a major part of our unstructured mesh solver and accounts for over 50% of the overall execution time. Since PETSc-FUN3D is a vertex-centered code,

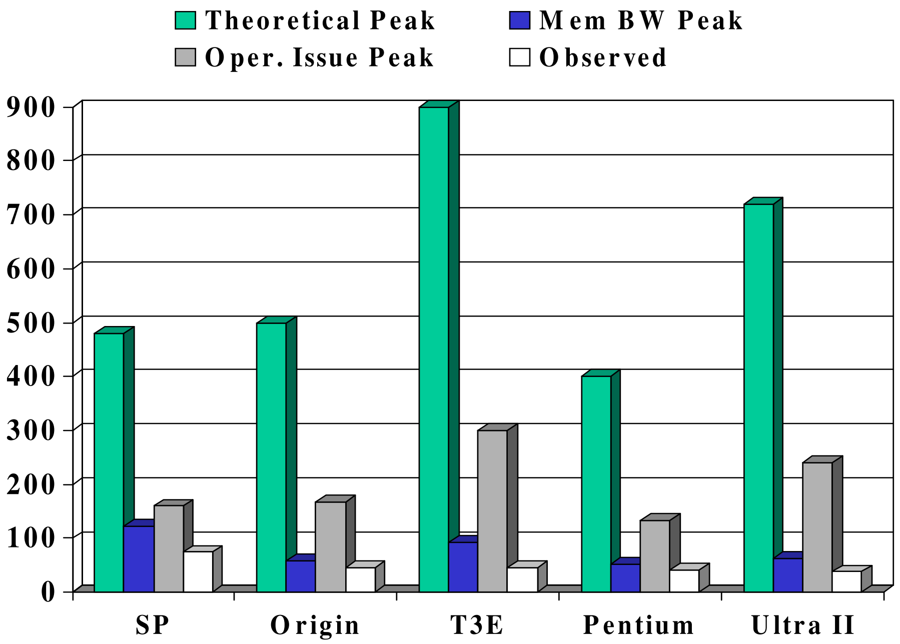

Figure 3. Three performance bounds for sparse matrix-vector product. The bounds based on memory bandwidth and instruction scheduling are much closer to the observed performance than the theoretical peak of the processor. Only one vector ($N = 1$) is considered here. The matrix size has `m` = 90,708 rows and `nz` = 5,047,120 nonzero entries. The processors are 120 MHz IBM SP (P2SC "thin", 128 KB L1), 250 MHz Origin2000 (R10000, 32 KB L1, and 4 MB L2), 450 MHz T3E (DEC Alpha 21164, 8 KB L1, 96 KB unified L2), 400 MHz Pentium II (running Windows NT 4.0, 16 KB L1, and 512 KB L2), and 360 MHz SUN Ultra II (4 MB external cache); memory bandwidth values are taken from the STREAM benchmark Web site.

the flow variables are stored at nodes. When a pass is made over an edge, the flow variables from the node-based arrays are read, many floating-point operations are done, and residual values at each node of the edge are updated. An analysis of the code suggests that, because of the large number of floating-point operations, memory bandwidth is not a limiting factor. Measurements on our Origin2000 support this conclusion; only 57 MB/s are needed. However, the measured floating-point performance is 209 Mflops/sec out of a peak of 500 Mflops/s. While this is good, it is substantially under the performance that can be achieved with dense matrix-matrix operations. To understand where the limit on the performance of this part of the code comes from, we take a close look at the assembly code for the flux calculation function. This examination yields the the following mix of the workload for each iteration of the loop over edges:

- 519 total instructions
- 111 integer operations

- 250 floating-point instructions of which there are 55 are `fmadd` instructions, for 305 flops

- 155 memory references

If all operations could be scheduled optimally—say, one floating-point instruction, one integer instruction, and one memory reference per cycle—this code would take 250 instructions and achieve 305 Mflops/s. However, dependencies between these instructions, as well as complexities in scheduling the instructions [12,13], make it very difficult for the programmer to determine the number of cycles that this code would take to execute. Fortunately, many compilers provide this information as comments in the assembly code. For example, on the Origin2000, when the code is compiled with cache optimizations turned off (consistent with our assumption that data items are in primary cache for the purpose of estimating this bound), the compiler estimates that the above work can be completed in about 325 cycles. This estimate leads to a performance bound of 235 Mflops/s (47% of the peak on 250 MHz processor). We actually measure 209 Mflops/s using hardware counters. This result shows that the performance in this phase of the computation is actually restricted by the instruction scheduling limitation. We are working on an analytical model for this phase of computation.

4. Conclusions

We have shown that a relatively simple analysis can identify bounds on the performance of critical components in an implicit CFD code. Because of the widening gap between CPU and memory performance, those parts of the application whose performance is bounded by the available memory bandwidth are doomed to achieve a declining fraction of peak performance. Because these are bounds on the performance, improvements in compilers cannot help. For these parts of the code, we are investigating alternative algorithms, data structures, and implementation strategies. One possibility, suggested by the analysis in Section 2, is to use algorithms that can use multiple vectors instead of a single vector with each sparse-matrix multiply.

For another part of our code, the limitation is less fundamental and is related to the mix of floating-point and non-floating-point instructions. Analyzing this code is more difficult; we relied on information provided by the compiler to discover the instruction mix and estimates on the number of cycles that are required for each edge of the unstructured mesh. Improving the performance of this part of the code may require new data structures (to reduce non-floating-point work) and algorithms (to change the balance of floating-point to other instructions).

5. Acknowledgments

We thank Kyle Anderson of the NASA Langley Research Center for providing FUN3D. Satish Balay, and Lois McInnes of Argonne National Laboratory co-developed the PETSc software employed in this paper. Computer time was supplied by DOE (through Argonne and NERSC).

REFERENCES

1. W. K. Anderson and D. L. Bonhaus. An implicit upwind algorithm for computing turbulent flows on unstructured grids. *Computers and Fluids*, 23:1–21, 1994.
2. W. K. Anderson, W. D. Gropp, D. K. Kaushik D. E. Keyes, and B. F. Smith. Achieving high sustained performance in an unstructured mesh CFD application. Technical Report ANL/MCS-P776-0899, MCS Division, Argonne National Laboratory, `http://www.mcs.anl.gov/petsc-fun3d/papers.html`, August 1999.
3. D. F. Bailey. How to fool the masses when reporting results on parallel computers. *Supercomputing Review*, pages 54–55, 1991.
4. S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. The Portable, Extensible Toolkit for Scientific Computing (PETSc) ver. 22. `http://www.mcs.anl.gov/petsc/petsc.html`, 1998.
5. J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of Fortran basic linear algebra subprograms: Model implementation and test programs. *ACM Transactions on Mathematical Software*, 14:18–32, 1988.
6. J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16:1–28, 1988.
7. W. D. Gropp. Performance driven programming models. In *Massively Parallel Programming Models (MPPM-97)*, pages 61–67. IEEE Computer Society Press, 1997.
8. J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 1996.
9. D. K. Kaushik, D. E. Keyes, and B. F. Smith. On the interaction of architecture and algorithm in the domain-based parallelization of an unstructured grid incompressible flow code. In J. Mandel et al., editor, *Proceedings of Proceedings of the 10th International Conference on Domain Decomposition Methods*, pages 311–319. Wiley, 1997.
10. D. K. Kaushik, D. E. Keyes, and B. F. Smith. Newton-Krylov-Schwarz methods for aerodynamic problems: Compressible and incompressible flows on unstructured grids. In C.-H. Lai et al., editor, *Proceedings of the 11th International Conference on Domain Decomposition Methods*. Domain Decomposition Press, Bergen, 1999.
11. J. D. McCalpin. STREAM: Sustainable memory bandwidth in high performance computers. Technical report, University of Virginia, 1995. `http://www.cs.virginia.edu/stream`.
12. MIPS Technologies, Inc., `http://techpubs.sgi.com/library/manuals/2000/007-2490-001/pdf/007-2490-001.pdf`. *MIPS R10000 Microprocessor User's Manual*, January 1997.
13. Silicon Graphics, Inc, `http://techpubs.sgi.com/library/manuals/3000/007-3430-002/pdf/007-3430-002.pdf`. *Origin 2000 and Onyx2 Performance and Tuning Optimization Guide*, 1998. Document Number 007-3430-002.
14. O. Temam and W. Jalby. Characterizing the behavior of sparse algorithms on caches. In *Proceedings of Supercomputing 92*, pages 578–587. IEEE Computer Society Press, 1992.
15. S. Toledo. Improving the memory-system performance of sparse-matrix vector multiplication. *IBM J. Res. and Dev.*, 41:711–725, 1997.
16. J. White and P. Sadayappan. On improving the performance of sparse matrix-vector multiplication. In *Proceedings of the 4th International Conference on High Performance Computing (HiPC '97)*, pages 578–587. IEEE Computer Society, 1997.
17. W. A. Wulf and A. A. McKee. Hitting the wall: Implications of the obvious. Technical Report CS-94-48, University of Virginia, Dept. of Computer Science, December 1994.