HOW MODERATE-SIZED RISC-BASED SMPs CAN OUTPERFORM MUCH LARGER DISTRIBUTED MEMORY MPPs

D. M. Pressel
Corporate Information and Computing Center
U.S. Army Research Laboratory
Aberdeen Proving Ground, Maryland 21005-5067
Email: dmpresse@arl.mil

Walter B. Sturek
Corporate Information and Computing Center
U.S. Army Research Laboratory
Aberdeen Proving Ground, Maryland 21005-5067
Email: sturek@arlmil

J. Sahu
Weapons and Materials Research Directorate
U.S. Army Research Laboratory
Aberdeen Proving Ground, Maryland 21005-5066
Email: sahu@arl.mil

K. R. Heavey
Weapons and Materials Research Directorate
U.S. Army Research Laboratory
Aberdeen Proving Ground, Maryland 21005-5066
Email: heavey@arl.mil

ABSTRACT: *Historically, comparison of computer systems was based primarily on theoretical peak performance. Today, based on delivered levels of performance, comparisons are frequently used. This of course raises a whole host of questions about how to do this. However, even this approach has a fundamental problem. It assumes that all FLOPS are of equal value. As long as one is only using either vector* ***or*** *large distributed memory MIMD MPPs, this is probably a reasonable assumption. However, when comparing the algorithms of choice used on these two classes of platforms, one frequently finds a significant difference in the number of FLOPS required to obtain a solution with the desired level of precision. While troubling, this dichotomy has been largely unavoidable. Recent advances involving moderate-sized RISC-based SMPs have allowed us to solve this problem. The net result is that for some problems a 128 processor Origin 2000 can outperform much larger MPPs.*

KEYWORDS: Supercomputer, high performance computing, parallel processor, computational fluid dynamics

1 Introduction

For a given job, one can define the *Delivered Performance* such that

*Delivered Performance = Theoretical Peak Performance * Total Efficiency*

where

*Total Efficiency = Algorithmic Efficiency * Serial Efficiency * Parallel Efficiency.*

Traditionally, many researchers using parallel computers have ignored the question of Algorithmic Efficiency and/or Serial Efficiency, preferring to stress Parallel Efficiency. A few people have even gone so far as to assume that all jobs on all machines have similar levels of efficiency, and therefore, all one needs to know is the Theoretical Peak Performance for the machines in question.

Note: This work was made possible through a grant of computer time by the Department of Defense (DOD) High Performance Computing Modernization Program. Additionally, it was funded as part of the Common High Performance Computing Software Support Initiative administered by the DOD High Performance Computing Modernization Program.

A direct consequence of this attitude has been the reaction by many users of vector computers who point out that while the parallel computers may be delivering higher levels of floating point operations per second, the vector computers will frequently have better wall clock time. Even when one takes into account the relative costs of the machines, an important consideration in throughput oriented environments, the vector machines will frequently fare much better than the raw numbers might indicate. Based on these observations, it is clear that one needs to consider Algorithmic Efficiency and Serial Efficiency as well as Parallel Efficiency when evaluating projects that use parallel computers.

When one compares the architectures of Cray vector computers (e.g., the C90), traditional MPPs (e.g., the Cray T3E or the IBM SP), and **RISC**[*]-based **SMPs** (e.g., the SGI Origin 2000 or the SUN HPC 10000), one finds significant differences in the design principles on which these systems are based. The Cray vector computers have vector processors, a low latency very high bandwidth memory system, and make very few assumptions about data locality or data reuse. In fact, the very nature of their design tends to discourage attempts to tune for data locality or reuse.

Traditional MPPs have an intermediate level of memory latency and bandwidth. Most of these systems now use RISC

[*] Definitions for boldface text can be found in the glossary.

processors with moderate-sized caches and some design features that facilitate streaming data into and out of the processor. Experiments reported by the NAS group at NASA Ames and an analysis performed by David O'Neal and John Urbanic at the Pittsburgh Supercomputing Center indicate that the memory system limitations on these systems result in a lower level of efficiency than with comparable codes running on the Cray vector machines (Saini 1996, 1997; O'Neal and Urbanic 1997).

RISC-based SMPs tend to have longer memory latencies and somewhat lower memory bandwidth than the MPPs. In general, they also have no special features designed to facilitate the streaming of data into and out of the processor. On the other hand, they are usually equipped with at least 1 MB of cache per processor. Codes that have been tuned to take advantage of this cache can in many cases reduce the rate of cache misses, that miss all the way back to main memory, to less than 1%. As a result, for some codes, it is possible to achieve serial levels of performance that actually exceed the performance achieved on MPPs (both in terms of absolute single processor performance and in terms of the percentage of peak). However, the serial performance for codes that were only tuned to run on an MPP may fall short of what is normally seen on an MPP (Sahu et al. 1997, 1998).

From this discussion, it should be clear that different classes of machines are likely to deliver different percentages of peak performance. Furthermore, the delivered level of performance is likely to strongly depend on the quality of the tuning (this includes the vector machines, since there are a lot of tricks to producing good vectorizable code). Finally, the ability to deliver well-tuned code will frequently depend on the design of the hardware itself.

For the rest of the discussion, it will be assumed that the codes are well tuned for serial performance and that the serial performance achievable with the MIPS R10K, HP PA-8000, Alpha 21164 (as configured for the T3E), and the IBM P2SC are all comparable. This means that in terms of efficiency, the MIPS R10K and the IBM P2SC have a significant lead over the other two chips (this agrees with published results as well as information that the author has received in private briefings). It is also assumed that the achievable serial efficiency of the MIPS R10K and the IBM P2SC approaches and, in some cases, matches that seen on Cray vector machines.[*]

The remainder of this paper will focus on parallel efficiency and algorithmic efficiency, with most of the emphasis being on the latter. The assumption here is that in many cases one can produce parallel efficiencies that are close to 100%.[†]

This leaves the question of what the algorithmic efficiency is. While it is hard to identify what one means by an algorithmic efficiency of 100%, it is generally easy to define the relative algorithmic efficiency of two approaches by comparing the number of floating point operations required to achieve a solution.[‡]

The important point here is that many parallelized programs have a significantly lower algorithmic efficiency than do the programs normally run on serial/vector processors. Evidence will be given that it is in some cases now possible to avoid this degradation. The down side is that the approaches that lead to this conclusion are in many cases not highly scalable. In some cases, they may work on a range of SMPs and MPPs, while in other cases, they will only work on SMPs. However, even with these limitations, it is possible that these techniques will allow the performance of a 64–128 processor job to equal or exceed the performance of a job using a 512 processor MPP written using traditional techniques.

2 Delivered Performance

If one asks most users of computer equipment what job characteristics they consider to be important, the general replies seem to center on two themes:

(1) The accuracy of the results.[§]

(2) The time to completion.[**]

While, in general, one assumes that these two themes are independent, in reality, that has rarely been the case. In a resource-constrained environment, it is easy to see how the sophistication of the calculations that one can reasonably hope to carry out will be limited. What may be less obvious is that there are other ways in which the search for a rapid time to completion can adversely affect the accuracy of the results.

A common problem, which comes from the field of Computational Fluid Dynamics (CFD), is that, frequently, the most efficient serial or vectorizable algorithm uses what is

---

[*] To a first approximation, the preceding assumptions seem to be nearly independent of the processor speeds in question. This probably means that each of these designs is strongly limited by the performance of the memory system and/or other system components.

[†] In situations where this is not likely to be the case, some discussion will be made of what the predicted behavior is.

[‡] Ideally, this comparison should be made on a single machine, even if the different algorithms are not normally all used on the same machine. This way, one is measuring differences in the algorithms, not in the compilers. However, even here, it is important that if operations are normally performed N times, where N is the number of processors, then if one is comparing using three different algorithms, one to be run on a vector machine with 16 processors, one on an SMP with 100 processors, and one on an MPP with 500 processors, then the operation counts should reflect the intended usage.

[§] In general, this is a relative concept and refers to the accuracy conforming to the expectations for a particular run.

[**] This may either apply to the time required to run a single job or the time required to run a complete set of jobs. In the former case, one is usually limited to using some or all of a single computer. In the latter case, one is usually limited to using some or all of the resources at one or a small number of computer centers.

known as an Implicit approach to solving the Navier-Stokes equations. Unfortunately, such an approach has generally resisted attempts to parallelize it. An example of this problem can be found in the F3D code out of NASA Ames Research Center, which uses a *block tridiagonal solver* based on Gaussian Elimination. Due to dependencies in this part of the code, even though this solver handles three-dimensional problems, each of the sweeps through the data can only be parallelized in a single direction. In the past, three methods have been used to get around this problem:

(1) In theory, one can replace Gaussian Elimination with a more parallelizable algorithm such as cyclic reduction. Unfortunately, this approach can itself result in two major problems:

- It increases the operation count by a factor of $LOG_2(N)$, where N is the number of processors being used. Clearly, this has the effect of decreasing the *algorithmic efficiency*.*
- Since this algorithm requires the use of a large number of relatively small messages, it was much better suited for use on **SIMD** (Flynn 1972) machines than for use with today's **MIMD** (Flynn 1972) machines.

(2) One can use an entirely different algorithm, such as one of the Explicit algorithms, which are known to be highly parallelizable. Unfortunately, the use of these algorithms will, in general, substantially increase the operation count required to obtain a solution. In other words, once again, the *algorithmic efficiency* suffers.

(3) Alternatively, one can use **domain decomposition** as the basis for parallelization. Unfortunately, this approach can severely compromise the convergence behavior of the algorithm. A number of approaches have been suggested to deal with this problem, but in all cases, the *algorithmic efficiency* will to some degree suffer† (Wang and Tafti 1997; Singh, Uthup, and Ravishanker, date unknown).

The key point here is that:

*Delivered Performance = Theoretical Peak Performance * Total Efficiency,*

where

*Total Efficiency = Algorithmic Efficiency * Serial Efficiency * Parallel Efficiency.*

* Using 512 processors will increase the operation count for this part of the solver by a factor of 9.

† Many of these approaches will also limit the available parallelism and/or adversely affect the parallel efficiency.

Therefore, any changes that result in a decrease in the *Algorithmic Efficiency* will directly affect the *Delivered Performance*, even though the performance as measured by **MFLOPS** might be quite high.‡ Using this unit, *Delivered Performance* is inversely proportional to the *Time to Completion* for a job (assuming that the processors are dedicated to this job).

Figure 1 shows an example of this for a fixed-size problem when one attempts to scale to large numbers of processors. Two things that are important to note here are:

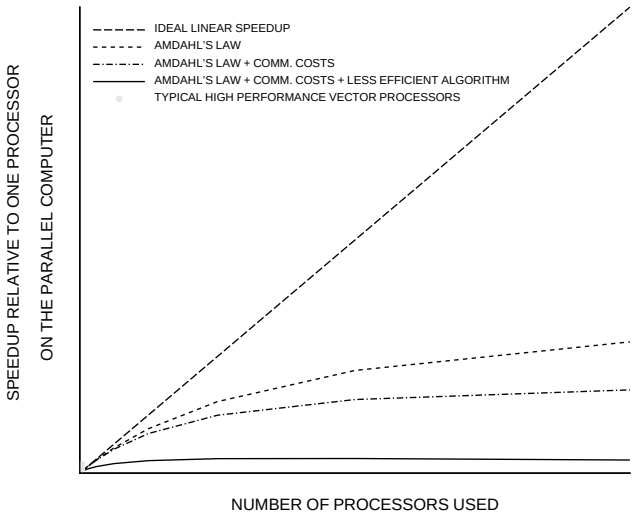

Figure 1. Predicted speedup from the parallelization of a problem with a fixed problem size.

(1) For sufficiently large numbers of processors, the combined effects of Amdahl's Law and the costs of interprocessor communication will limit the maximum achievable level of performance. Therefore, for all but the largest problem sizes, and given enough processors, the parallel efficiency may be far less than 100%.

(2) The effect of going to less efficient algorithms in an attempt to improve the parallelizability of the code can virtually eliminate the perceived benefits of having a highly parallelizable code.

If one applies Gustafson's (1988) concept of *scaled speedup*, one can overcome some if not all of the limiting effects attributable to Amdahl's Law and interprocessor communication. However, this concept will have little impact on the loss of algorithmic efficiency. Therefore, the basic premise behind Figure 1 (and this paper) remains intact.

3 Loop-Level Parallelism

‡ The units for *Delivered Performance* are Useful *MFLOPS*.

It turns out that there is an alternative way in which one can parallelize Implicit CFD codes, which does not result in a reduction of their *Algorithmic Efficiency*. This approach is based on parallelizing the individual loops and is therefore referred to as Loop-Level Parallelism. Of course, if this method is so great, then one might wonder why it was not the method of choice all along. The following are some of the reasons for this:

- Loop-Level Parallelism in general is based on the same parallelism used to produce vector code. Therefore if the program is to run in parallel on a vector computer such as the Cray C90, it will be difficult to produce a code that exhibits both good vector performance and good parallel performance at the same time.

- While in theory it is possible to implement Loop-Level Parallelism using some form of message-passing code, the result can be a huge number of calls to the message-passing library (either to implement matrix transpose operations and/or to manually implement some form of coherency protocol). By comparison, when Loop-Level Parallelism is implemented on a shared memory system, it is not uncommon to leave the loops in the boundary condition routines unparallelized (in general, these loops may represent 80% or more of the loops in the program, but less than 1% of the total work). This makes it both painful to implement Loop-Level Parallelism using message-passing code and, in general, results in code that is very inefficient.

- Traditionally, there have been two types of shared memory platforms. The first type is based on a small number of vector processors. This tends to make the system very expensive, while limiting one's ability to show good speedup. As a result, many codes run on vector processors were never parallelized. The second type of system was based on inexpensive mass-market microprocessors. Unfortunately, until recently, the aggregate peak speed of systems based on this design was generally much less than the peak speed of one processor on a state-of-the-art vector machine from Cray Research.

Therefore, until recently, none of the machines commonly used for High Performance Computing were well suited for use with Loop-Level Parallelism. It was not until the advent of the SGI Power Challenge that one could make a clear case for investigating this approach. Even then, enough people equated Loop-Level Parallelism with Automatic Parallelization (a concept that doesn't work very well) that they failed to properly appreciate the potential for this approach (Theys, Braun, and Siegel 1998). In fact, even now there are only a few systems (e.g., the SGI Origin 2000) for which a compelling case can be made (in some cases, the bottleneck is the hardware, while in other cases, limitations in the operating system and/or the compilers are at fault).

Table 1 shows the potential benefit of using Loop-Level Parallelism in conjunction with a well-designed shared memory system. Table 2 shows the actual speedup that was achieved for different problem sizes when using Loop-Level Parallelism with an SGI Origin 2000 to run the F3D code for a common test case.

Table 1. The No. of Processors Required to Achieve a Specified Level of Delivered Performance Using Traditional Techniques

Speedup Relative to One Processor	Minimum No. of Processors Required When Using	
	Domain Decomposition	Cyclic Reduction
16	64	108
32	181	256
48	333	418
64	512	589
80	716	767

4 Speedup

Up until now, this discussion has assumed that one can easily achieve linear speedup. In reality, this is frequently not the case. Therefore, let us consider what is likely to be the case when using both the traditional approaches to parallelization and loop level parallelism. Based on the numbers in Table 1, it is clear that when using traditional approaches, one will likely need a large number of processors. However, for fixed-size problems, Amdahl's Law predicts that there is enough serial code remaining that one will asymptotically approach a maximum level of performance when using large numbers of processors. The traditional counter argument has been to use the concept of *scaled speedup* (Gustafson 1988). With this concept, the available parallelism and the available work are assumed to scale linearly with the problem size. Therefore, as the problem size gets bigger, one can use additional processors while keeping the run time constant. This concept also assumes that the amount of work associated with the serial code grows very slowly, if at all, and can therefore be ignored.

A common rule of thumb when parallelizing programs on distributed memory MPPs is that one should use the smallest number of processors possible, with the amount of memory per processor usually being the limiting factor. Most modern MPPs are now equipped with between 64 MB and 1 GB of memory per processor, with somewhere around 10–20 MB of memory per processor reserved for use by the operating system. Based on these

Table 2. The Speedup Achieved When Using the F3D Code Parallelized Using Loop-Level Parallelism on SGI Origin 2000's

No. of Processors Used	Grid Size (Millions of Grid Points)	Speedup Relative to One Processor
23	1.00	16.8
23	3.00	16.3
21	6.00	16.1
20	12.0	16.4

20	23.8	17.0
21	59.4	18.1
21	124.0	17.6
85	1.0	32.4
81	3.0	32.2
55	6.0	32.1
46	12.0	33.0
45	23.8	32.1
41	59.4	33.1
41	124.0	32.8
114	12.0	48.1
87	23.8	49.8
61	59.4	48.1
61	124.0	48.6
117	59.4	66.9
88	124.0	65.7
116	124.0	81.4

Note: Except for the largest test case, runs using fewer than 64 processors were run on either 32 or 64 processor Origin 2000's. Due to the memory requirements of the largest test case, all runs were made on a 128 processor Origin 2000. For all of the remaining cases, runs were made on a preproduction 128 processor Origin 2000.

numbers, Table 3 shows how many processors one would normally expect to use for the test cases mentioned in Table 2.

Table 3. The No. of Processors That One Would Normally Use When Using an MPP and Traditional Techniques to Process the Test Cases

Grid Size (Millions of Grid Points)	Recommended No. of Processors
1.0	1–10
3.0	2–30
6.0	3–60
12.0	6–120
23.8	12–240
59.4	30–600
124.0	62–1,240

There is no guarantee that one will actually get good scalability all the way to the upper bounds listed in Table 3. Rather, the upper bound is based on the impossibility of running the job if there is not enough memory. However, the rule of thumb indicates that it is questionable if one will see linear speedup when using even larger numbers of processors.

When comparing Tables 1 and 3, it becomes apparent that there are some problems. The smaller test cases are unlikely to produce speedups much in excess of a factor of 16. While in theory the larger problem sizes will fare better, there is a second, less obvious problem. Very few of the currently installed MPPs are configured with 512 or more processors. Therefore, in many cases, one will find it difficult, if not impossible, to use enough processors to get speedups of 64 or greater.

Turning our attention to programs parallelized using Loop-Level Parallelism, the following question comes up: What kinds of speedup is one likely to see from these programs? The answer here is a bit complicated. In general, the available parallelism will be a function of the smallest of the grid dimensions. Therefore, the available parallelism will, at best, scale as the cube root of the size of each zone. A direct result of this is that it no longer makes sense to talk about scaled speedup. Instead, one is back at the problem of obtaining speedup for a fixed problem size.

The second problem is that when using Loop-Level Parallelism, the available parallelism is frequently within an order of magnitude of the number of processors being used. Since there are an integer number of iterations in a loop, the predicted speedup is no longer linear, but rather is a stairstep.

Figure 2 and Table 4 show an example of this. This also means that, for smaller problems, one may run out of parallelism in some/or all of the loops prior to using all of the processors in the machine.

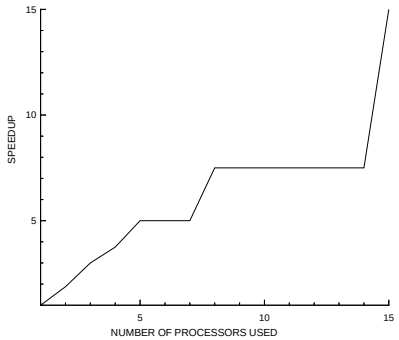

Figure 2. Predicted speedup for a loop with 15 units of parallelism.

Table 4. Predicted speedup for a loop with 15 units of parallelism

No. of Processors	Maximum Units of Parallelism Assigned to a Single Processor	Predicted Speedup
1	15	1.000
2	8	1.875
3	5	3.000
4	4	3.750
5–7	3	5.000
8–14	2	7.500
15	1	15.000

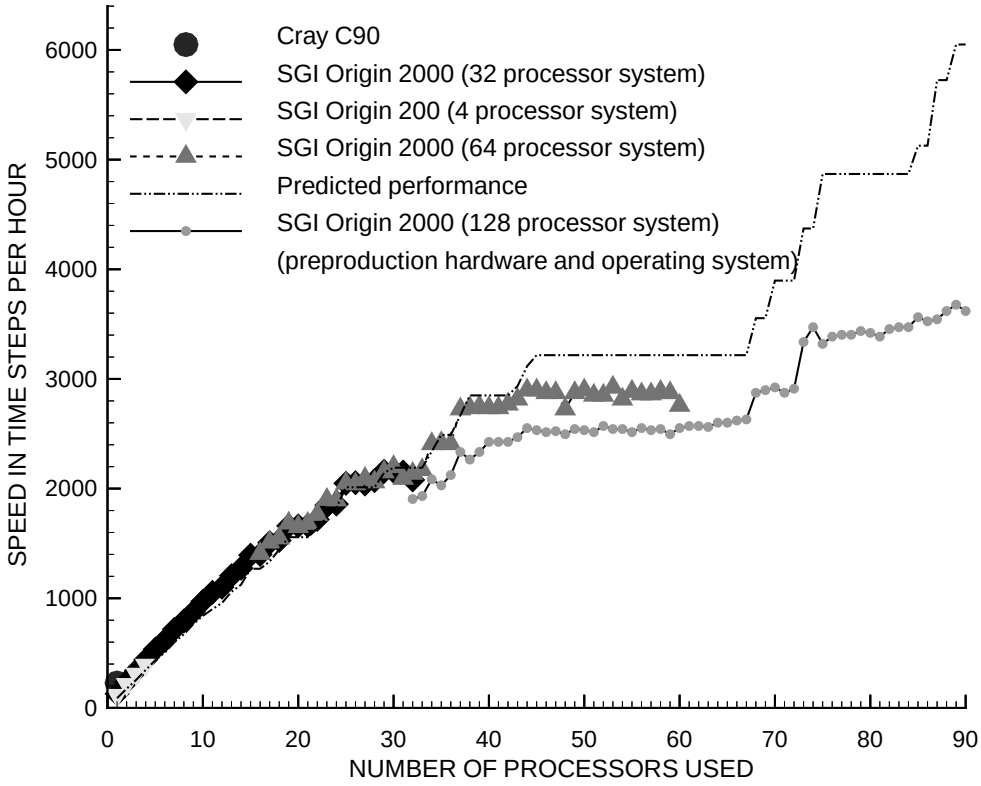

Figure 3. Performance results for the one-million-grid-point data set.

An additional complication with Loop-Level Parallelism is that since many of the loops will be doing very little work, the overhead associated with parallelizing them may be so great as to result in Parallel Slowdown! This situation is especially common in the boundary condition routines. While in theory this problem should be less severe when dealing with larger problem sizes, the reality of the situation is that the code will normally be tuned for the smaller problem sizes. While in many cases it should be possible to reduce the amount of CPU time spent on serial code to 1% or less of the total CPU time, this is enough for Amdahl's Law to be a problem when using more than about 50 processors. The combination of the stairstepping with Amdahl's law explains why the smaller test cases show limited speedup in Table 2.*

Figures 3 and 4 show all of these effects in a real problem. Figure 3 is for a relatively small problem (less than 500 MB of memory), while Figure 4 is for a relatively large problem (over 20 GB of memory). Our calculations indicate that the primary reason for the difference between the predicted and measured levels of performance in these curves is Amdahl's law.†

The combination of Loop-Level Parallelism and RISC-based SMPs has been shown to be a promising approach to parallelizing a class of highly efficient algorithms that had previously resisted attempts at parallelization. Additionally, evidence has been presented that demonstrates that, in general, the resulting code achieves a much higher level of delivered performance than traditional techniques might be expected to deliver. While it is not practical to look at all possible approaches in detail and to determine what their effect is on the Total Efficiency in all cases, it seems likely that the benefits of using our approach are real.

---

* A third reason for the limited speedup is that the average memory latency on the 128 processor Origin 2000 is slightly longer than on the 32 and 64 processor systems. This has the effect of decreasing the serial efficiency from 30 to 40% to about 25–30% on the 128 processor system.

† The predicted curve is based on the assumption that one can achieve the same percentage of peak performance for a single processor job on both the Cray C90 and on RISC-based machines such as the SGI Origin 2000. This does not mean that one will achieve this without any work. Rather, it is assumed that a significant effort at tuning the code was made on both platforms. Taking into account only the available level of parallelism (the stairstepping effect), this expected level of performance is then extraploted out for multiprocessor runs. As such, the predicted level of performance will, in general, equal or exceed the observed level of performance and serve as an excellent reference point for determing how well the system is performing.

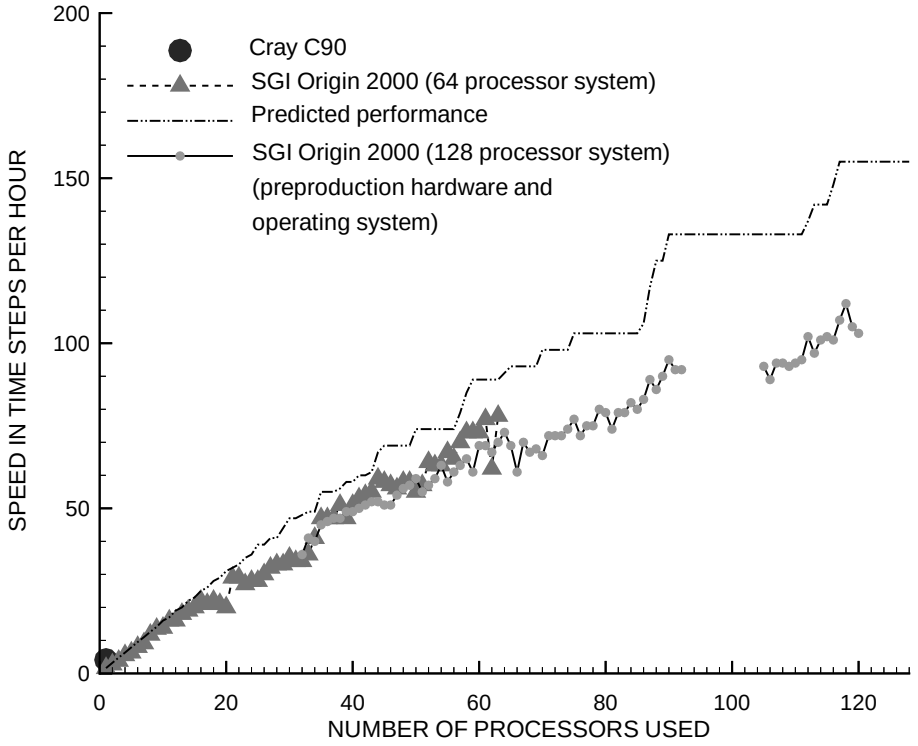

Figure 4. Performance results for the 59-million-grid-point data set.

5 Conclusions

An additional consideration is the availability of the hardware. SGI and SUN have both been quite successful at selling moderate-sized RISC-based SMPs. While IBM and Cray (a subsidiary of SGI) have sold a significant number of MPPs, very few of them had 512 or more processors. Therefore, even when in theory the performance of a large MPP using traditional methods should exceed our results, it is far from certain that one will actually be able to obtain access to enough processors in a single machine at one time.

6 References

Flynn, M. J. Some Computer Organizations and Their Effectiveness. IEEE Transactions Computers, C-21 948-60, 1972.

Gustafson, J. L. Reevaluating Amdahl's Law. Communications of the ACM, vol. 31, no. 5, pp. 532–533, The Association for Computing Machinery, Inc., May 1988.

O'Neal, D., and J. Urbanic. On Performance and Efficiency: Cray Architectures. Parallel Applications Group Pittsburgh Supercomputing Center, Electronically published at http://www.psc.edu/~oneal/eff/eff.html, August 1997.

Sahu, J., D. M. Pressel, K. R. Heavey, and C. J. Nietubicz. Parallel Application of a Navier-Stokes Solver for Projectile Aerodynamics. Published in Parallel Computational Fluid Dynamics, Recent Developments and Advances Using Parallel Computers. Proceedings of the Parallel CFD'97 Conference Manchester, U.K., 19–21 May 1997. Edited by D. R. Emerson, J. Periaux, A. Ecer, N. Satofuka, and P. Fox, Amsterdam: Elsevier, 1998.

Sahu, J., D. M. Pressel, K. R. Heavey, and C. J. Nietubicz. Parallel Application of a Navier-Stokes Solver for Projectile Aerodynamics. To be published in the proceedings of the 1998 Army Science Conference.

Saini, S. (ed.). NAS Parallel Benchmarks, NPB 1 Data. Electronically published at http://Science.nas.nasa.gov/Software/NPB/NPB1Results/index.html, 17 November 1996.

Saini, S. (ed). NAS Parallel Benchmarks, NPB 2 Data. Electronically published at http://Science.nas.nasa.gov/Software/NPB/NPB2Results/index.html, 17 November 1997.

Singh, K. P., Biju Uthup, and Laxmi Ravishanker. Parallelization of Euler and N-S Code on 32 Node Parallel Super Computer PACE+. Presented at the ADA/DRDO-DERA Workshop on CFD, date unknown.

Theys, M. D., T. D. Braun, and H. J. Siegel. Widespread Acceptance of General-Purpose, Large-Scale Parallel Machines: Fact, Future, or Fantasy? IEEE Concurrency Parallel,

Distributed and Mobile Computing, published by the IEEE Computer Society, January–March 1998 issue.

Wang, G., and D. K. Tafti. Performance Enhancement on Microprocessors With Hierarchical Memory Systems for Solving Large Sparse Linear Systems. Submitted to the International Journal of Supercomputing Applications, February 1997.

GLOSSARY

RISC: Reduced Instruction Set Computer.

SIMD: Single Instruction Multiple Data - A class of parallel computers as defined in Flynn's taxonomy.

SMP: Symmetric Multiprocessor - A term normally only applied to shared memory systems using hardware memory coherency protocols.

MIMD: Multiple Instruction Multiple Data - A class of parallel computers as defined in Flynn's taxonomy.

Domain decomposition: The process of splitting a small number of zones (some of which are assumed to be large) into a moderate to large number of zones (generally all of which are fairly small in size).

MFLOPS: Million Floating Point Operations Per Second.

MPP: Massively Parallel Processor.