Performance Metrics for Parallel Systems

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

ABSTRACT: One frequently needs to compare the performance of two or more parallel computers; but how should this be done? The most straightforward way to do this would be to rely upon a suite of benchmarks. Unfortunately, there are many problems and limitations with this approach. Therefore, one is frequently forced to rely upon a combination of approaches. Unfortunately, many of the alternative approaches are all to frequently based on excessively simplistic approximations and extrapolations. This paper will discuss some of these issues so that the problems they can cause may be avoided in the future.

KEYWORDS: Supercomputer, high performance computing, performance metrics, parallel programming.

1 Introduction

It is frequently necessary to compare the performance of two or more parallel computers. While in theory this might seem like a simple enough proposition, the reality is that it is far from being simple. Let us assume that the most logical approach is to rely upon a suite of benchmarks based on the expected work load. How should the benchmarks be selected? We already said that they were to be based on the expected work load, but what does this mean? Some sites have very steady work loads that slowly evolve over time; however many sites will see significant variations in the work load from month to month. Additionally, most sites have a collection of jobs that they would like to run, but which either cannot be run on their current systems, or at best can only be run as a demo run in dedicated mode.

Continuing to look at this problem in greater depth, two additional problems come to mind:

(1) The benchmark suite has to be runable on the available benchmarkinghardware. In other words, while most vendors have one or more systems that they use for running benchmarks, in general, these systems are not maxed-out configurations. Furthermore, there is competition for these systems, so in general, there will be some reluctance on the part of the vendors to run benchmark suites that take more than a few days to run. This point becomes even more important if the suite needs to be run more than once (e.g., to judge the effectiveness of tuning the code and/or the hardware configuration). This may make it particularly difficult to benchmark extremely large, and presumably highly scalable, jobs to see if the largest configurations perform as promised.

(2) What should the benchmark code look like? If one supplies source code and rules concerning what can be done to it, that very action can adversely impact the performance of the code on some systems. For example:

- Vector optimized code will, in general, perform suboptimally on **RISC**-based platforms.
- Code that has already been parallelizedmight have been done so based on certain assumptions about the computer's architecture. For example, it might have been written in CM-Fortran and targeted at the CM-5 made by Thinking Machines. Such a program can in theory be transformed into **HPF**. In theory, such a program is highly portable. The reality can be quite different, and even when the program is "portable," this says nothing about its performance or cost effectiveness.
- Even if message-passing code that has been run on an early RISC-based system (e.g., the Intel Paragon) is supplied, the tuning of the code may be suboptimal. For example, it might have assumed that 1–10 **MFLOPS** were good. Given a peak-per-processor interprocessor communication rate of 100 **MB**/s, this is likely to result in a favorable ratio between

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.

Note: Definitions for boldface text can be found in the Glossary.

communication and computation. However, if one switches to a processor where the expected performance is in the range of 50–200 MFLOPS, then it is unlikely that the ratio between communication and computation will still be as favorable. In particular, the latency of interprocessor communication is likely to be much more important.

The Numerical Aerodynamic Simulation (NAS) group at **NASA** Ames Research Center came up with an alternative approach. Version 1 of the NAS Parallel Benchmarks (NPB1) was based on paper and pencil descriptions of the algorithms. It was up to each vendor to write their own code from scratch and to optimize in almost any way they saw fit. The only requirements were that they could not change the algorithm and that the results had to pass certain tests. While this approach has the decided advantage of allowing the vendors to show off their wares in the most favorable light possible, it turned out that in many cases this light was much too favorable.

One final note on this topic is that so far we have been requiring the vendor to use a specific algorithm. The problem is that not only is vector optimized code suboptimal for RISC- based architectures but it may also be difficult to parallelize. In particular, it may have a limited amount of available parallelism. For this and other reasons, there may be a strong benefit to using a combination of Loop-Level Parallelism and Shared Memory **SMPs** when trying to parallelize such a code. On the other hand, if one sticks to algorithms that are known to perform well on current **MPPs**, one may be accepting a significant decrease in the algorithmic efficiency and, therefore, the delivered performance. Such a hit may be unavoidable if one is committed to using MPPs; however, these considerations demonstrate why it can be easy to underestimate the value of a Shared Memory SMP such as the SGI Origin 2000.

The rest of the paper will discuss some commonly used metrics that are frequently used to supplement, or even replace, benchmarking as the basis for procurement decisions.

2 Reviewing the Metrics

When discussing metrics for any form of High Performance Computing with the user community, four metrics are commonly mentioned:*

(1) Does the machine have enough (usable) memory to run my job?

(2) How long will it take for my job to run?

(3) What is the overall turnaround time for the job?

(4) Can I use my favorite algorithm on the new computer?

The first metric is, in general, fairly simple to evaluate. Either the machine has enough memory, or it doesn't (and one would have to use an out-of-coresolver). In reality, things can be a bit more complicated since some combinations of computer architecture and software will result in some or all of the data being replicated up to N times, where N is the number of processors being used. Additionally, some architectures will favor the use of additional arrays (possibly for scratch purposes, or alternatively to hold the transpose of another array), which can also increase the memory requirements. Even so, it is, in general, fairly simple to predict if a job will fit in the memory of a certain machine.†

The second metric is the one that seems to be most frequently discussed by the software developers, although it is this author's experience that most users are most interested in the third metric. Sections 3 and 4 will discuss these metrics in greater detail.

The author is presenting a separate paper devoted to the fourth metric at this conference [2]. Therefore, this metric will not be discussed in great detail. The important point of the fourth metric is that some algorithms are easier to port to some system architectures than others. In particular, some of the most efficient serial algorithms are relatively easy to port to vector-based and/or Shared Memory SMPs. At the same time, they can be much more difficult to port to traditional Distributed Memory MPPs. As such, the choice of hardware can in some cases limit the choice of algorithm and in so doing negatively impact the observed results for the second and third metric.

3 How Long Will It Take for My Job to Run?

While the second metric discusses the run time for a complete job, rarely will one see presentations based on run time for the complete job. While there are several reasons for this, the most common is that it takes too much computer time to obtain fully converged solutions. This is especially the case if one wishes to use this metric to measure performance as a function of the number of processors used.

---

* "Programmers should always look for the best available algorithm (parallel or not) before trying to scale up the 'wrong' algorithm. For folks other than computer scientists, time to solution is more important than linear speedup [1]."

† It is important to keep in mind that on a distributed memory architecture, the memory requirement will frequently set the lower bound on the number of processors that can be used for a certain job. Historically, distributed memory architectures were configured with limited amounts of memory per processor. As a result, one was sometimes forced to use more processors than were optimal for a particular job, just to have enough memory to run that job at all. While this is in general less of an issue today than in the past (i.e., some distributed memory systems are now equipped with 1 GB of memory per processor), the issue has not gone away entirely (e.g., as late as 1996, the default configuration for the Cray T3E was 64 MB per processor) [3][4].

For simulations that have the concept of a *Time Step* (e.g., runs involving **CFD**), a common solution to this problem is to assume that it takes a fixed number of time steps to solve the problem (unfortunately, this is only an approximation whose validity is sometimes left unresolved) [1]. Given this solution, one can then stick to comparing the performance based on calculations involving a relatively small number of time steps. The down side to this is that *start up* and *termination* costs will now skew the results. The solution to this problem is of course to use some method to remove these costs from consideration (the assumption being that they will not be excessively large when running the complete problem).

If the simplifications were to stop at this point, we would, in general, still be on fairly solid ground. Unfortunately, that is rarely the case. Instead, several different approaches are frequently applied to simplifying this metric even further, and it is highly questionable if any of these approaches can be justified except by the claim of expediency. Let us briefly look at some of these simplifications and see where the problems are.

3.1 Ideal Speedup

Most textbooks on parallel programming talk about the *ideal speedup* as being linear speedup. They then go on to talk about how the performance will deviate from linear speedup as the result of Amdahl's Law, the costs associated with interprocessor communication, etc. They may also talk about *scaled speedup* [5]. However, while for obvious reasons it is desirable to have linear speedup as the ideal speedup, some approaches to parallelization can result in distinctly nonlinear speedup for the ideal speedup (see the next paragraph for an example).

If one has a code that is vectorizable but difficult to parallelize using message-passing code (e.g., many Implicit CFD codes fall into this category), then the most appropriate way to parallelize the code might be to use loop-level parallelism (e.g., OPENMP). However, if one assumes that the number of processors being used is within a factor of 10 of the available parallelism (an assumption that will in general be true when working with 3-D problems and 32 or more processors), then the ideal speedup will be a stairstep, rather than a straight line. Furthermore, if different loops are parallelized in different directions, and each direction has a different number of iterations associated with it, then the ideal speedup will consistently fall well below the line for linear speedup.

The basis of this claim has to do with the number of iterations being a small-to-moderate-sized integer. To see this, let us assume that we have only one loop to parallelize and that the available number of processors equals the available parallelism. Let us create a plot of performance vs. the number of processors used. As we increase the number of processors being used, we will reach the point where all but one or two of the processors have two iterations assigned to them, with the remaining processor(s) having three iterations assigned to them. Add another processor, and all of the processors have, at most, two iterations assigned to them, and the speed takes a noticeable jump. Add additional processors, and nothing happens. As a result of adding the additional processors, more and more of the processors only have one iteration assigned to them, but as long as even one processor has two iterations assigned to it, the speed will remain unchanged (in terms of the ideal speedup, on some real systems, the speed may actually drop). Finally, when the number of processors equals the available parallelism, one will get a significant jump in performance.

This effect can be clearly seen in the theoretical numbers in Figure 1 and Table 1. Furthermore, if one looks at actual results that we have for the Implicit CFD code F3D, one can see that at least on a well-designed system such as the SGI Origin 2000, this effect really does exist.

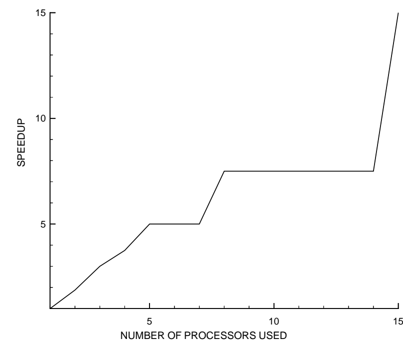

Figure 1. Predicted speedup for a loop with 15 units of parallelism.

Table 1. 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

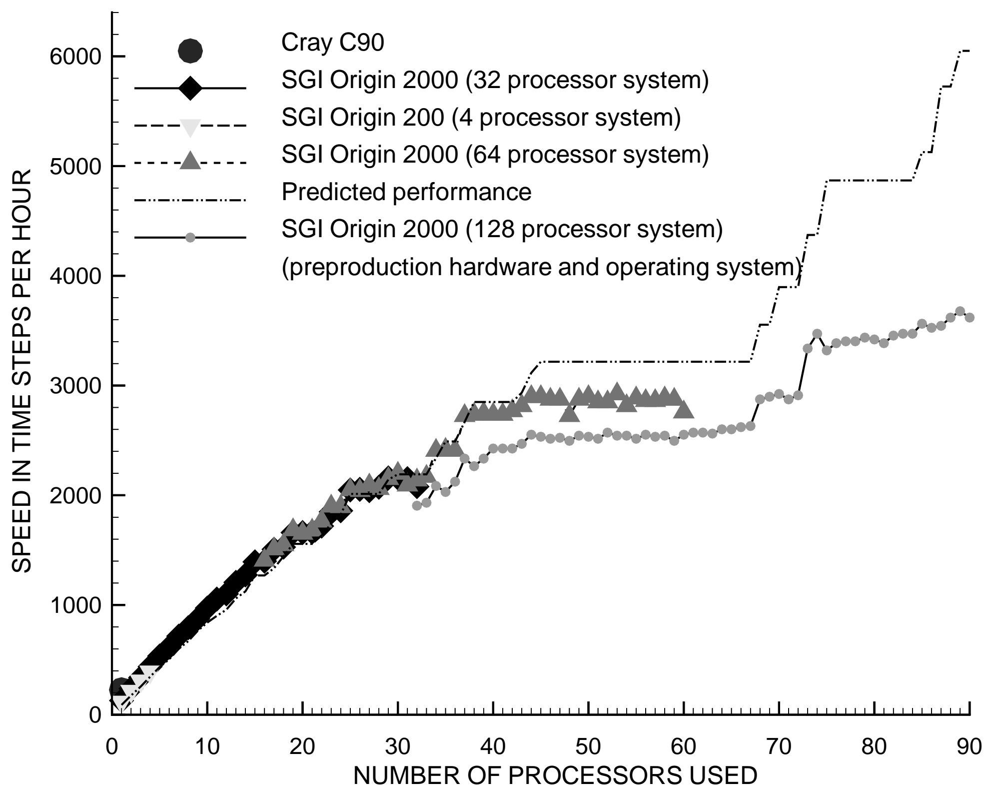

Figure 2. Performance results for the one-million-grid-point data set.

From this discussion, one can see that assumptions regarding what constitutes the ideal speedup can vary depending on the approach to parallelization. Additionally, since the use of loop-level parallelism is primarily restricted to Shared Memory SMPs (as well as the now obsolete SIMD systems), the predicted speedup can, at least, indirectly depend on the choice of system architecture.

3.2 Linear Speedup

Many people seem to be of the impression that the compilers are producing excellent serial code. Therefore, all that they really need to worry about is getting good linear speedup (or, at least, linear scaled speedup). A few people go even further and assume that any good code will get linear speedup/scaled speedup and that the code will achieve approximately the same percentage of peak on all RISC (or even RISC/**CISC**) based systems. Based on this train of thought, one is left with the inescapable conclusion that unless the vendor's pricing is way out of line, one should buy the system with the fastest processor possible (based purely on peak speed). In general, these conclusions are supported by the published Linpack numbers.

The problem is that Linpack is an easily tuned code that, in general, gives a very high level of performance on almost all HPC systems currently in production. If one compares the Linpack numbers to those of other HPC benchmarks (e.g., the NAS Parallel Benchmarks), one will, in general, see a poor level of correlation between the two sets of numbers. There are three main reasons for this:

(1) Given poorly tuned code, few if any compilers will produce a highly efficient executable. In general, poorly tuned code will result in performance levels that range from poor to fair. The problem is compounded by the design decisions that all vendors make in designing their systems. In some cases, these decisions can make it nearly impossible to produce highly tuned code for some/many algorithms. Therefore, the actual performance can depend dramatically (in some cases by a factor of 10 or more) on the skill of the programmer and the choices made by the system designers.

(2) While some computational kernels (e.g., Linpack) are easily made to *live* out of cache, for many other programs, that is not the case (especially if they are still written in a manner which favors vector processors). In that case, the speed of the program is likely to correlate strongly with how rapidly data can be moved between memory and the processor. Unfortunately, for some

systems, this may limit the program's performance to only a few percent of peak (based on MFLOPS) [6].

(3) As the processors get faster, and assuming that one is able to maintain the performance as a constant percentage of peak, then it becomes increasingly more difficult to obtain linear speedup without increasing the amount of work assigned to each processor. The problem here is that if one doubles the processor speed, while decreasing the number of processors used by a factor of two, it is far from clear that the system with the fastest processors is the best system to buy. In particular, since systems with faster processors almost always cost more per Peak MFLOPS, the system with the slower processors might actually be more cost effective and just as fast (given the appropriate number of processors).

Therefore, we have seen that basing decisions purely on linear speedup (or even worse, the assumption of linear speedup) can fail to properly predict the performance of a code-machine combination for the second metric.

3.3 Maximum Speed

Is the fastest machine the best machine for the job? If one is a *Grand Challenge* user, or alternatively a software developer primarily interested in pushing the boundaries for really large jobs, then faster is certainly better. Once again, this can lead one to the conclusion that one should only buy/use highly scalable systems with very fast processors and lots of memory (although how much memory per processor can still be debated). However, such a decision can have several penalties associated with it:

- The new machine might not meet the needs of all of the users (e.g., those interested in running parametric studies involving huge numbers of serial jobs).
- Bigger machines tend to be harder and therefore more expensive to build. In general, the more scalable the design is, the greater the cost per delivered (usable) MFLOPS is likely to be.
- Many users may not need to use huge numbers of processors. Alternatively, the users may only be using large numbers of processors to satisfy their memory requirements. Therefore, while many sites are likely to find that huge highly scalable systems can produce excellent results for the second metric, they will produce far from optimal results for the third metric.

However, at many sites, the majority of the cycles are consumed by jobs using a small-to-moderate percentage of the total available resources. Therefore, for the users at these sites, once the system size has reached a certain point, there is little or no benefit (there may even be a large penalty) from increasing the size of an individual system any further. In such a case, it is reasonable to do a cost/benefit analysis comparing a single large system to multiple, moderate-sized systems (this is especially true when dealing with the metacenter concept, where some centers may go one way, while others are free to go the other way, thereby ensuring that the needs of Grand Challenge users continue to be addressed). This thought will be discussed further in section 4.

4 What Is the Overall Turnaround Time for the Job?

The overall turnaround time is primarily the sum of the following values:

- Time spent waiting to start running (on many systems, this can be as long as a week or longer).
- The minimum amount of time required for the program to execute (this is what one would see on a dedicated system).
- Any degradation to the run time resulting from not running on a dedicated system (e.g., resulting from extra context switches and/or time spent sitting at spin locks).
- Time spent waiting for the operating system to schedule a job's time slice when running in a time-sharing environment.

From this, one can see that the three things that matter are:

(1) Minimizing the time spent waiting for a job to start running.

(2) Ensuring that the minimum run time will more than meet the needs of most users.

(3) Minimizing/eliminating any sources of degradation to the run time that result from running in a multiuser environment.

In theory then, if the nonGrand Challenge jobs collectively need X amount of delivered **GFLOPS**, with any one job requiring a minimum of Y MFLOPS to meet its performance requirements, then one can buy any system/combination of systems that delivers X GFLOPS in quantas of Y MFLOPS. One can then go out and buy the cheapest system that meets these specifications.

The reality is frequently a bit different than that. First, one frequently has a fixed amount of money to spend, so instead of buying the cheapest system, one will normally buy the largest complex of systems one can afford. The second point is that flexibility in scheduling and use will frequently translate into a cost savings that will allow one to run more jobs. In other words, if different jobs use different numbers of processors, then it is probably best from a scheduling standpoint if the smallest systems being considered are at least 2-3 * Y MFLOPS in size.

Another consideration is, How efficiently is the memory being used? On shared memory systems, some jobs will be able to share a single pool of memory M MB in size. Running the same job on a distributed memory architecture might then require replicating the data structures in the local memory of every processor (this is a common practice for chemistry codes). As a result, when using N processors, one must now buy N * M MB of memory. This can make it much cheaper to run these jobs on a shared memory system, such as those made by SGI and SUN. Unfortunately, it is not always clear if this savings will still be present when running on less main stream architectures, such as the KSR1 or even the Convex Exemplar with its CTI cache.

Even when running a standard message-passing job, different users may wish to see different amounts of memory per processor. If one equips all of the processors only with the largest amount of memory possible, the throughput might suffer due to the decreased number of processors in the system. Therefore, in a throughput oriented environment, it is probably best to equip different groups of processors with different amounts of memory. Exactly how this should be accomplished (especially when dealing in a metacenter type of environment) can be quite complicated (in part, because politics will almost always be a factor in the final decision). While this discussion has centered around memory vs. processors, there can be other resources (e.g., disk drives, I/O controllers, and even software licenses) that can also be an important part of the picture.

5 Conclusion

We have seen that there can be a number of factors that decide which metric is most appropriate in each circumstance. While many users will want to only report linear speedup, that is frequently a metric of questionable value. In particular, while one might be tempted to say that his or her metric has nothing to do with the procurement decisions that this paper keeps stressing, the truth is that more often than not they will have everything to do with those decisions. If the decision makers only see glowing reports, their conclusions may be unintentionally biased. If they see mostly glowing reports and only a limited number of reports that stress the thornier issues, it can be all too easy to ignore those issues. Therefore, any user who writes reports based solely on oversimplified metrics runs the very real risk of leading management, and possibly even the entire industry, down a path of uncertain value (at best).

6 References

[1] Dowd, K., and C. Severance. *High Performance Computing: RISC Architectures, Optimization & Benchmarks 2nd Edition*, Sebastopol, CA: O'Reily & Associates, Inc., 1998.

[2] Pressel, D. M., W. B. Sturek, J. Sahu, and K. R. Heavey. "How Moderate-Sized RISC-Based SMPs Can Outperform Much Larger Distributed Memory MPPs." To be published in the conference proceedings for the 1999 Advanced Simulation Technologies Conference, San Diego, CA, sponsored by The Society for Computer Simulation International (SCS), 11–15 April 1999.

[3] Saini, S. (editor). NAS Parallel Benchmarks. NPB 1 Data.
Electronically published at
http/science.nas.nasa.gov/Software
/NPB/NPB1Results/index.html, 17 November 1996.

[4] Fox, G. C., R. D. Williams, and P. C. Messina. *Parallel Computing Works!* San Francisco, CA: Morgan Kaufmann Publishers, Inc., 1994.

[5] 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.

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

Glossary

CFD - Computational Fluid Dynamics

CISC - Complicated Instruction Set Computer, an approach to processor design that assumes that the best way to get good performance out of a system is to provide instructions that are designed to implement key constructs (e.g., loops) from **high level languages**.

GLOPS - Billion Floating Point Operations per Second
High Level Languages - Computer languages that are designed to be relatively easy for the programmer to read and write. Examples of this type of language are Fortran, Cobol, C, etc.

HPF - High Performance Fortran

Low Level Languages - Computer languages that are designed to reflect the actual instruction set of a particular computer. In general, the lowest level language is known as Machine Code. Just slightly above machine code is a family of languages collectively known as Assembly Code.

MB - Million Bytes

MFLOPS - Million Floating Point Operations per Second

MPP - Massively Parallel Processor

NASA - National Aeronautics and Space Administration

RISC - Reduced Instruction Set Computer, an approach to processor design that argues that the best way to get good performance out of a system is to eliminate the Micro Code that CISC systems use to implement most of their instructions. Instead, all of the instructions will be directly implemented in hardware. This places obvious limits on the complexity of the instruction set, which is why the complexity had to be *reduced*.

SMP - Symmetric Multiprocessor