Twelve Ways to Fool the Masses When Giving Performance Results on Parallel Computers

David H. Bailey

June 11, 1991
Ref: Supercomputing Review, Aug. 1991, pg. 54--55

Abstract

Many of us in the field of highly parallel scientific computing recognize that it is often quite difficult to match the run time performance of the best conventional supercomputers. This humorous article outlines twelve ways commonly used in scientific papers and presentations to artificially boost performance rates and to present these results in the “best possible light” compared to other systems.

The author is with the Numerical Aerodynamic Simulation (NAS) Systems Division at NASA Ames Research Center, Moffett Field, CA 94035.

Many of us in the field of highly parallel scientific computing recognize that it is often quite difficult to match the run time performance of the best conventional supercomputers. But since lay persons usually don't appreciate these difficulties and therefore don't understand when we quote mediocre performance results, it is often necessary for us to adopt some advanced techniques in order to deflect attention from possibly unfavorable facts. Here are some of the most effective methods, as observed from recent scientific papers and technical presentations:

1. Quote only 32-bit performance results, not 64-bit results.

We all know that it is hard to obtain impressive performance using 64-bit floating point arithmetic. Some research systems do not even have 64-bit hardware. Thus always quote 32-bit results, and avoid mentioning this fact if at all possible. Better still, compare your 32-bit results with 64-bit results on other systems. 32-bit arithmetic may or may not be appropriate for your application, but the audience doesn't need to be bothered with such details.

2. Present performance figures for an inner kernel, and then represent these figures as the performance of the entire application.

It is quite difficult to obtain high performance on a complete large-scale scientific application, timed from beginning of execution through completion. There is often a great deal of data movement and initialization that depresses overall performance rates. A good solution to this dilemma is to present results for an inner kernel of an application, which can be souped up with artificial tricks. Then imply in your presentation that these rates are equivalent to the overall performance of the entire application.

3. Quietly employ assembly code and other low-level language constructs.

It is often hard to obtain good performance from straightforward Fortran or C code that employs the usual parallel programming constructs, due to compiler weaknesses on many highly parallel computer systems. Thus you should feel free to employ assembly-coded computation kernels, customized communication routines and other low-level code in your parallel implementation. Don't mention such usage, though, since it might alarm the audience to learn that assembly-level coding is necessary to obtain respectable performance.

4. Scale up the problem size with the number of processors, but omit any mention of this fact.

Graphs of performance rates versus the number of processors have a nasty habit of trailing off. This problem can easily be remedied by plotting the performance rates for problems whose sizes scale up with the number of processors. The important point is to omit any mention of this scaling in your plots and tables. Clearly disclosing this fact might raise questions about the efficiency of your implementation.

5. Quote performance results projected to a full system.

Few labs can afford a full-scale parallel computer --- such systems cost millions of dollars. Unfortunately, the performance of a code on a scaled down system is often not very impressive. There is a straightforward solution to this dilemma --- project your performance results linearly to a full system, and quote the projected results, without justifying the linear scaling. Be very careful not to mention this projection, however, since it could seriously undermine your performance claims for the audience to realize that you did not actually obtain your results on real full-scale hardware.

6. Compare your results against scalar, unoptimized code on Crays.

It really impresses the audience when you can state that your code runs several times faster than a Cray, currently the world's dominant supercomputer. Unfortunately, with a little tuning many applications run quite fast on Crays. Therefore you must be careful not to do any tuning on the Cray code. Do not insert vectorization directives, and if you find any, remove them. In extreme cases it may be necessary to disable all vectorization with a command line flag. Also, Crays often run much slower with bank conflicts, so be sure that your Cray code accesses data with large, power-of-two strides whenever possible. It is also important to avoid multitasking and autotasking on Crays --- imply in your paper that the one processor Cray performance rates you are comparing against represent the full potential of a $25 million Cray system.

7. When direct run time comparisons are required, compare with an old code on an obsolete system.

Direct run time comparisons can be quite embarrassing, especially if your parallel code runs significantly slower than an implementation on a conventional system. If you are challenged to provide such figures, compare your results with the performance of an obsolete code running on obsolete hardware with an obsolete compiler. For example, you can state that your parallel performance is “100 times faster than a VAX 11/780”. A related technique is to compare your results with results on another less capable parallel system or minisupercomputer. Keep in mind the bumper sticker “We may be slow, but we’re ahead of you.”

8. If MFLOPS rates must be quoted, base the operation count on the parallel implementation, not on the best sequential implementation.

We know that MFLOPS rates of a parallel codes are often not very impressive. Fortunately, there are some tricks that can make these figures more respectable. The most effective scheme is to compute the operation count based on an inflated parallel implementation. Parallel implementations often perform far more floating point operations than the best sequential implementation. Often millions of operations are masked out or merely repeated in each processor. Millions more can be included simply by inserting a few dummy loops that do nothing. Including these operations in the count will greatly increase the resulting MFLOPS rate and make your code look like a real winner.

9. Quote performance in terms of processor utilization, parallel speedups or MFLOPS per dollar.

As mentioned above, run time or even MFLOPS comparisons of codes on parallel systems with equivalent codes on conventional supercomputers are often not favorable. Thus whenever possible, use other performance measures. One of the best is “processor utilization” figures. It sounds great when you can claim that all processors are busy nearly 100% of the time, even if what they are actually busy with is synchronization and communication overhead. Another useful statistic is “parallel speedup” --- you can claim “fully linear” speedup simply by making sure that the single processor version runs sufficiently slowly. For example, make sure that the single processor version includes synchronization and communication overhead, even though this code is not necessary when running on only one processor. A third statistic that many in the field have found useful is “MFLOPS per dollar”. Be sure not to use “sustained MFLOPS per dollar”, i.e. actual delivered computational throughput per dollar, since these figures are often not favorable to new computer systems.

10. Mutilate the algorithm used in the parallel implementation to match the architecture.

Everyone is aware that algorithmic changes are often necessary when we port applications to parallel computers. Thus in your parallel implementation, it is essential that you select algorithms which exhibit high MFLOPS performance rates, without regard to fundamental efficiency. Unfortunately, such algorithmic changes often result in a code that requires far more time to complete the solution. For example, explicit linear system solvers for partial differential equation applications typically run at rather high MFLOPS rates on parallel computers, although they in many cases converge much slower than implicit or multigrid methods. For this reason you must be careful to downplay your changes to the algorithm, because otherwise the audience might wonder why you employed such an inappropriate solution technique.

11. Measure parallel run times on a dedicated system, but measure conventional run times in a busy environment.

There are a number of ways to further boost the performance of your parallel code relative to the conventional code. One way is to make many runs on both systems, and then publish the best time for the parallel system and the worst time for the conventional system. Another is to time your parallel computer code on a dedicated system and time your conventional code in a normal loaded environment. After all, your conventional supercomputer is very busy, and it is hard to arrange dedicated time. If anyone in the audience asks why the parallel system is freely available for dedicated runs, but the conventional system isn't, change the subject.

12. If all else fails, show pretty pictures and animated videos, and don't talk about performance.

It sometimes happens that the audience starts to ask all sorts of embarrassing questions. These people simply have no respect for the authorities of our field. If you are so unfortunate as to be the object of such disrespect, there is always a way out --- simply conclude your technical presentation and roll the videotape. Audiences love razzle-dazzle color graphics, and this material often helps deflect attention from the substantive technical issues.

Acknowledgments

The author wishes to acknowledge helpful contributions and comments by the following persons: R. Bailey, E. Barszcz, R. Fatoohi, P. Frederickson, J. McGraw, J. Riganati, R. Schreiber, H. Simon, V. Venkatakrishnan, S. Weeratunga, J. Winget and M. Zosel.