Guidelines for Presenting Parallel Performance

Lawrence A. Crowl

Department of Computer Science
Oregon State University
Corvallis, Oregon 97331-3202

Technical Report 92-80-07

December 1992

Abstract

People use parallel computers for improved program performance, so effective presentation of parallel performance is important. This paper gives advice on how to effectively present parallel performance in a manner that reduces distortion over common practice. The paper concentrates primarily on visual aspects of presentation, and secondarily on methodological aspects.

Computing Reviews Categories and Subject Descriptors: C.4 [**Performance of Systems**] — *measurement techniques, performance attributes* D.1.3 [**Programming Techniques**]: Concurrent Programming

Keywords: parallel performance, performance graphs, benchmarking

1 Introduction

People use parallel computers to improve the performance of programs, that is, to make them solve a single problem in less time or to solve more problems in a fixed time. Because performance is so important to parallel programming, people devote substantial effort to measuring, presenting, and comparing the performance of parallel systems. Unfortunately, many current presentations of performance are not very effective. This paper gives a set of guidelines to clearly and effectively present parallel performance. By knowing what writers should be doing, readers will be better able to interpret and critically evaluate a writer's presentation.

In presenting and comparing parallel performance, your overall goal should be to communicate the maximum amount of information, in the least amount of space and time, while minimizing any distortions. Three general guidelines for achieving this are:

Present information, not just data. Simply publishing a list of all data points collected would communicate all the data available, but would communicate information poorly. Your audience rarely needs to see the raw data. Reserve the raw data for an appendix, or better yet, anonymous FTP.

Effectively use space and time. On the printed page, two things limit your ability to communicate, the space available and the time the reader must spend to interpret your presentation. For only a few data points, simply listing the data points may minimize the space and time. For many data points, a carefully crafted graph or other visualization may minimize the space and time.

Minimize distortion. Many things can distort parallel performance, either intentionally or accidentally. While some distortion appears to be inevitable, choosing the right measures of parallel performance, presenting them well on paper, and making fair comparisons will produce minimum distortion.

These guidelines are in conflict, so you must choose the balance among information, space, time, and distortion.

This paper is organized as follows: section 2 discusses what to measure to obtain accurate performance; section 3 discusses what information to present; and section 4 discusses how to present that information on paper. Since comparing performance yields new insights, section 5 discusses what information should be compared; and section 6 discusses how to present that comparison on paper. Section 7 summarizes this paper. Throughout the paper, I use examples of actual parallel program executions. Those program executions are derived from the open literature are referenced.

2 What to Measure

Before presenting parallel performance, you must first measure the performance. There are many factors that go into determining the performance of a parallel program — the algorithm, the programming system, the machine, the problem, the execution time, etc. (See section 5 for further discussion.) You can control all of them directly except time. So, measure time.

Measuring time seems pretty obvious. What is not so obvious is what kind of time to measure. Should you measure single process CPU time, total CPU time, or elapsed (wall clock) time?

single process CPU time: Unless the operating system guarantees that all processes receive exactly the same CPU time, some processes will use less CPU than others. If the measured process happens to be one that consumes less CPU time, the measured computation fails to accurately reflect the amount of work done by the unmeasured processes. You can see this effect in the coarse-grained executions of Figure 11b.

total process CPU time: By measuring the CPU time of all processes, you can ensure that you measure all the work done. However, for many parallel programs, the time spent waiting for locks, paging, or I/O can be a substantial fraction of the total elapsed time. Total process CPU time fails to capture waiting time.

elapsed (wall clock) time: People use parallel computers to obtain better elapsed time, and not better CPU time. (CPU time is a mechanism for cost accounting in shared environments.) Measuring elapsed time will account not only for the computational work, but also for any waiting for locks, paging, and I/O. Much of this waiting may be inherent in the program, and hence is an integral part of parallel performance. (A consequence of using elapsed time as a measure is that performance of the machine should be measured on a dedicated machine, which may be unrealistic for some systems.) Measuring elapsed execution time captures both computation and waiting.

Figure 1 (derived from [Deshpande and Schultz, 1992].) shows one example of significant difference between CPU time and elapsed time. While CPU time may be a tempting and convenient measure, the bottom line in parallel program performance is throughput, and elapsed time is the only strong measure of parallel performance.

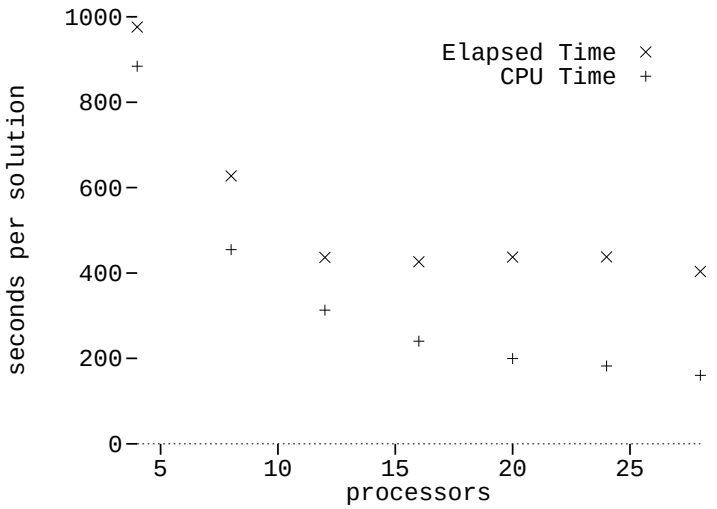

Figure 1: Elapsed Time versus CPU Time

Measure Statistics

You may be tempted to measure the execution time of your program once and report that number. Resist the temptation. Parallel programs must often deal with random events in the system; memory accesses may be reordered, operating system daemons may or may not execute, etc. Because of these random events, parallel program execution time is not deterministic; it varies around some mean execution time. To ensure that the measured execution time of a program is not some statistical fluke, you must measure execution time over several program runs and determine the mean and variance of the program's execution time. A high variance may indicate problems with the program, such as high contention for locks. Without statistical measures of the program's execution time, you don't know if the time is a distortion of reality.

Particularly prevalent are distortions based on cache effects. When making several execution runs of a parallel algorithm, particularly within a single program, the first few runs within a single program may be significantly faster than subsequent runs. The could occur because of any of the many caches that modern systems uses, such as instruction caches, main memory caches, virtual memory caches, file system caches, *etc.* If you find that the cache affects your timings, you can adopt one of two strategies.

transient cache: Most programs will yield the same answer each time the program runs, so running the program more than once is pointless. In these cases, measuring multiple execution times requires that any caches be flushed so that you accurately measure the transient cache startup behavior. In addition to the memory cache, you must also worry about address translation caches, page migration and replication caches, file system caches, etc. Measuring transient caches is the preferred method for accounting for the cache.

stable cache: Sometimes the time necessary to collect transient cache execution times exceeds the time available. In this case, you should measure execution time after the cache is stable, which may involve leaving the first few executions out of statistical measures, such as average execution time. You should document clearly that you are measuring execution times for a stable cache, and that real applications should not expect the same level of performance.

Measuring elapsed (wall clock) time statistically yields fewer opportunities for distorted performance results.

3 What to Present

Once you have a statistical measure of the elapsed time, you must decide what to present to the reader. Simply present the elapsed time, and avoid presenting speedup where possible.

Speedup is a common way to present parallel performance, but it has several problems, which arise from the fact that speedup is a derivative measure and there are many choices on how to derive speedup. In a survey of parallel programmers Barr and Hickman [1991] found no clear consensus on how to derive speedup. Each time you present speedup, you force your reader to investigate how you derived speedup, and how to interpret your results.

kind of speedup: Are you using normal speedup (fixed problem size), scaled speedup (problem size grows with the number of processors [Gustafson, 1988; Gustafson *et al.*, 1988]), or memory-bound speedup (problem size grows until memory limited [Sun and Ni, 1992])? For scaled speedup, does the problem size grow linearly with the number of processors, or does the resulting complexity grow with with the number of processors? If scaling with complexity, are you scaling with worst-case complexity or average-case complexity? For some problems, the complexity is not even known, which makes scaling with complexity problematic.

sequential time base: What is the base time used to calculate speedup — the parallel program on one processor, an equivalent sequential program, or the best known sequential program? Does your machine sacrifice sequential performance for parallel scalability? (Most do.) If so, the sequential time will be artificially high, yielding overestimates on the speedup. In some cases, the problem is too large to execute on a single processor and you must guess at a base time, which is ripe for distortion.

resources available: The concept of speedup implies that only the number of processors changes. However, in any real machine, more than just the number of processors changes. For example, as you use more processors, you're likely to get some combination of a larger cache, more physical memory, and more contention from system daemons. Because of growing resources, some programs get superlinear speedup! This causes no end of argument, [Faber *et al.*, 1986; Parkinson, 1986; Faber *et al.*, 1987; Janßen, 1987]. Speedup implicitly ignores growing resources.

Speedup is highly subject to misinterpretation. On the other hand, if you present elapsed time; you avoid making derivative interpretations; you avoid making your reader check your derivation; and you avoid making claims about your base case. In short, presenting elapsed time does not distort your results.

4 How to Present

How you present information on paper can have a significant effect on how well you communicate. Many factors are involved in mapping information to paper and balancing them requires care and taste. Tufte [1983; 1990] gives an excellent perspective on how to present information on paper, and I recommend his books to any author presenting quantitative information.[1]

4.1 Tables, Bar Charts, and Graphs

The amount of data you wish to present has a strong influence on the best way to present it. If you have only a few data points, it is best to just list the data points in a table, as in the table of Figure 2, which appeared in [Deshpande and Schultz, 1992]. It takes little space and is easy to interpret. For more than three or four data points it is best to use either a bar chart or graph. Bar charts work best when the independent variable is not quantitative, such as the type of machine or

[1] Many graphical plotting packages do not follow Tufte's recommendations, in which case you will have to balance your effort with the quality of the presentation.

the type of application. Graphs work best when the independent variable is quantitative, such as the number of processors. For example, the graph of Figure 2 presents information more effectively than the corresponding table.

Processors	Sequent Symmetry	Encore Multimax
1	14417.9	15338.0
2	7058.0	7566.1
4	3566.2	3807.2
6	2412.8	2584.5
8	1799.4	1934.9
10	1471.9	1577.9
12	1227.4	1320.5
14	1069.9	1137.9
16	944.5	999.1
18	872.4	917.8

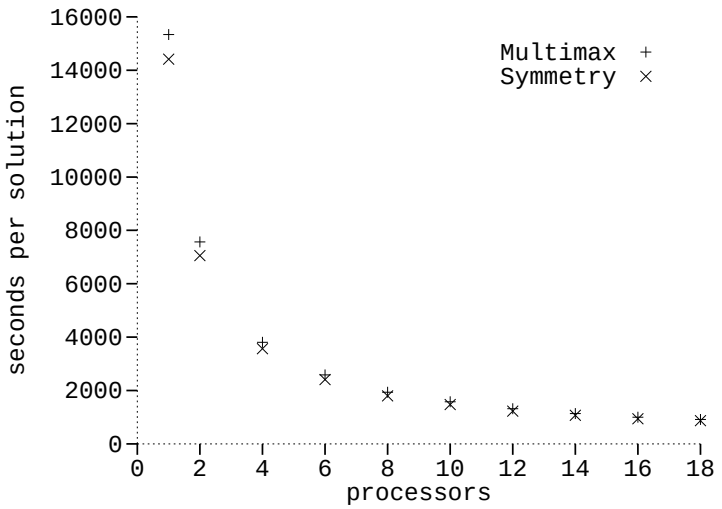

Figure 2: Table versus Graph

Since most of what I have to say about bar charts applies equally well to graphs, I will concentrate on graphs. In addition, I will concentrate on the number of processors, p, as the independent variable and execution time, $t(p)$, as the dependent variable.

4.2 Plot Functions

When designing a graph, you must choose a function that plots data onto paper. The simplest is to just use a linear function. For example, each unit in the independent variable may correspond to one centimeter horizontally on paper and each unit in the dependent variable may correspond to two centimeters vertically on paper.

Since I can't know how large your units are, or how much paper you have, I will concentrate on the general type of the function, not on its specific coefficients. I will discuss the properties of four function types, linear time (width proportional to the number of processors and height proportional to time), linear speed (width proportional to the number of processors and height proportional to the inverse of time), log-log time (width proportional to the logarithm of the number of processors and height proportional to the logarithm of time), and log-log speed (width proportional to the logarithm of the number of processors and height proportional to the logarithm of the inverse of time). The linear-speed plot is visually similar to speedup plots, and can serve as a substitute.

Qualitative Changes Are Obvious

The first property we must ask of a plot function is that qualitative changes be easy to see. For example, Figure 3 (derived from [Crowl, 1991].) shows the same parallel performanceunder each

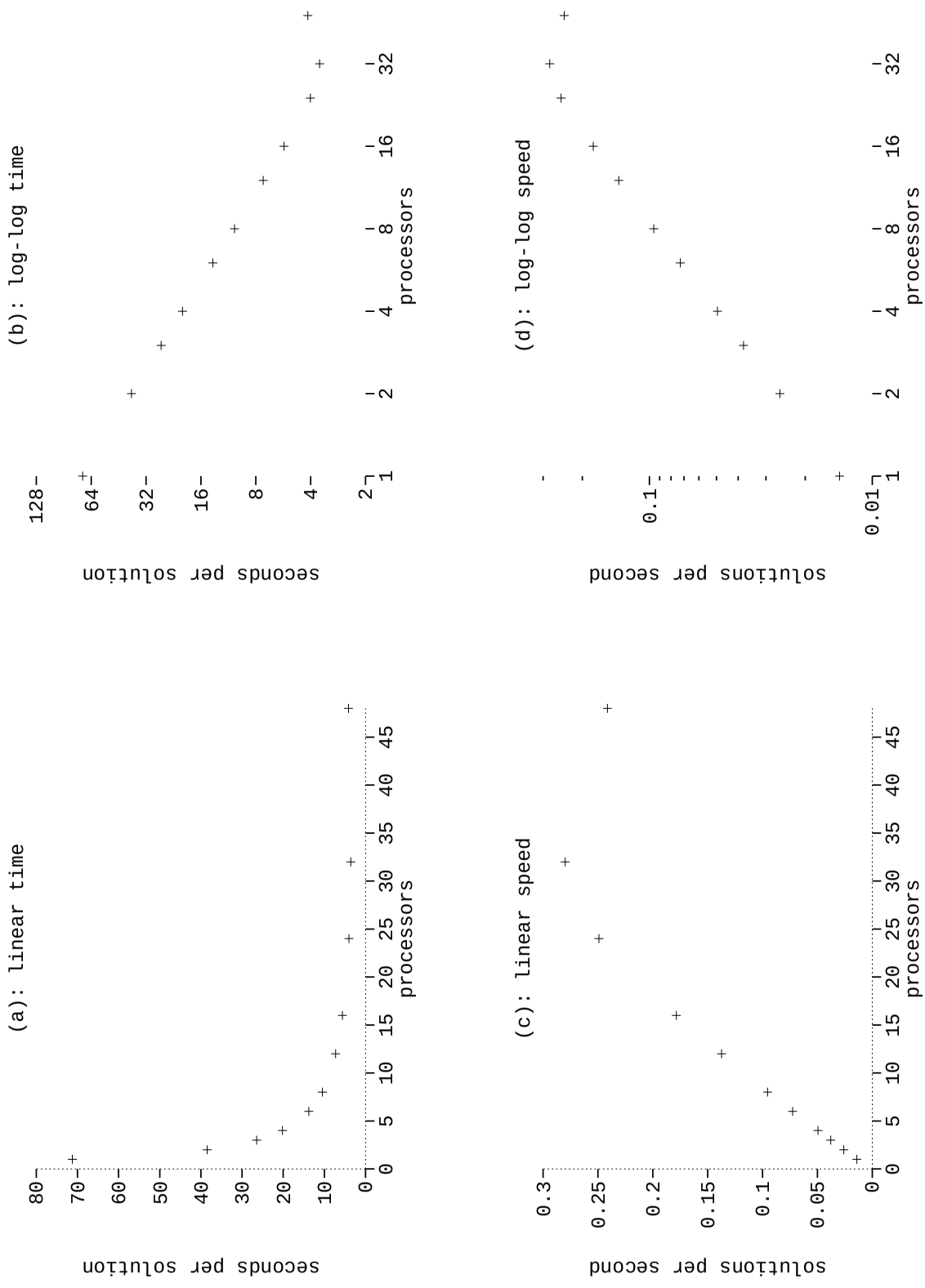

Figure 3: Showing Qualitative Changes

of the four plot functions. In looking for qualitative changes, we should be able to see where performance is improving, leveling off, and getting worse. These changes are easy to see in the linear speed, log-log time, and log-log speed, but difficult to see in the linear time plot. This is because relatively large changes in performance at high numbers of processors receive very little visual space in the linear plot.

Linear Performance is Linear on Paper

People are good at recognizing straight lines, but fairly poor at recognizing specific curves, such as $1/x$. So, the plot function should make linear performance look straight on paper. Linear plots don't have this property, which means you should avoid linear plots. For example, in Figure 4 (also derived from [Crowl, 1991]) it is clear that performance is non-linear in most plots, but it is almost impossible to see that performance is non-linear in the linear-time plot. Similarly, in Figure 5 the near-linear performance in all but the linear-time plot is clear.

Absolute Performance is Linear on Paper

When the absolute number of processors and absolute performance are important to your point, you want the number of processor and the resulting performance to be proportional on paper. For example, each additional processor might change the horizontal position of the point on paper by one centimeter. The only plot function that achieves this property for both processors and performance is the is the linear speed plot function.

There is a danger with linear speed plots. If the origin is not at $(0, 0)$, marginally better programs can look much better. For example, Figure 6a visually overestimates the loss of performance past eight processors. Figure 6b does not. This same problem also appears in bar charts that do not have a zero origin, or which "tear out" part of the bar. "Tear outs" occur in bar graphs that have a zero origin, but then remove the middle part of the bar to reduce space. So, always use zero based origins.

Relative Performance is Linear on Paper

When the relative change in processors and performance are an important to your point, you want relative changes in processors and performance to be proportional on paper. For example, each 10% increase in processors might change the paper position by one centimeter. The linear-time and linear-speed plot functions do not satisfy this property, but the log-log-time and log-log-speed plots do. With log-log plots, the relative changes are given equal visual space, which means we see the relative performance for both small and large numbers of processors easily. Linear speed plots tend not to show relative performance well for small numbers of processors; and linear time plots tend not to show relative performance well for large numbers of processors.

Performance is Scale Invariant on Paper

The log-log plot functions have another useful property — scale invariance. This means that doubling each value on the curve does not change the shape of the curve. The curve may change

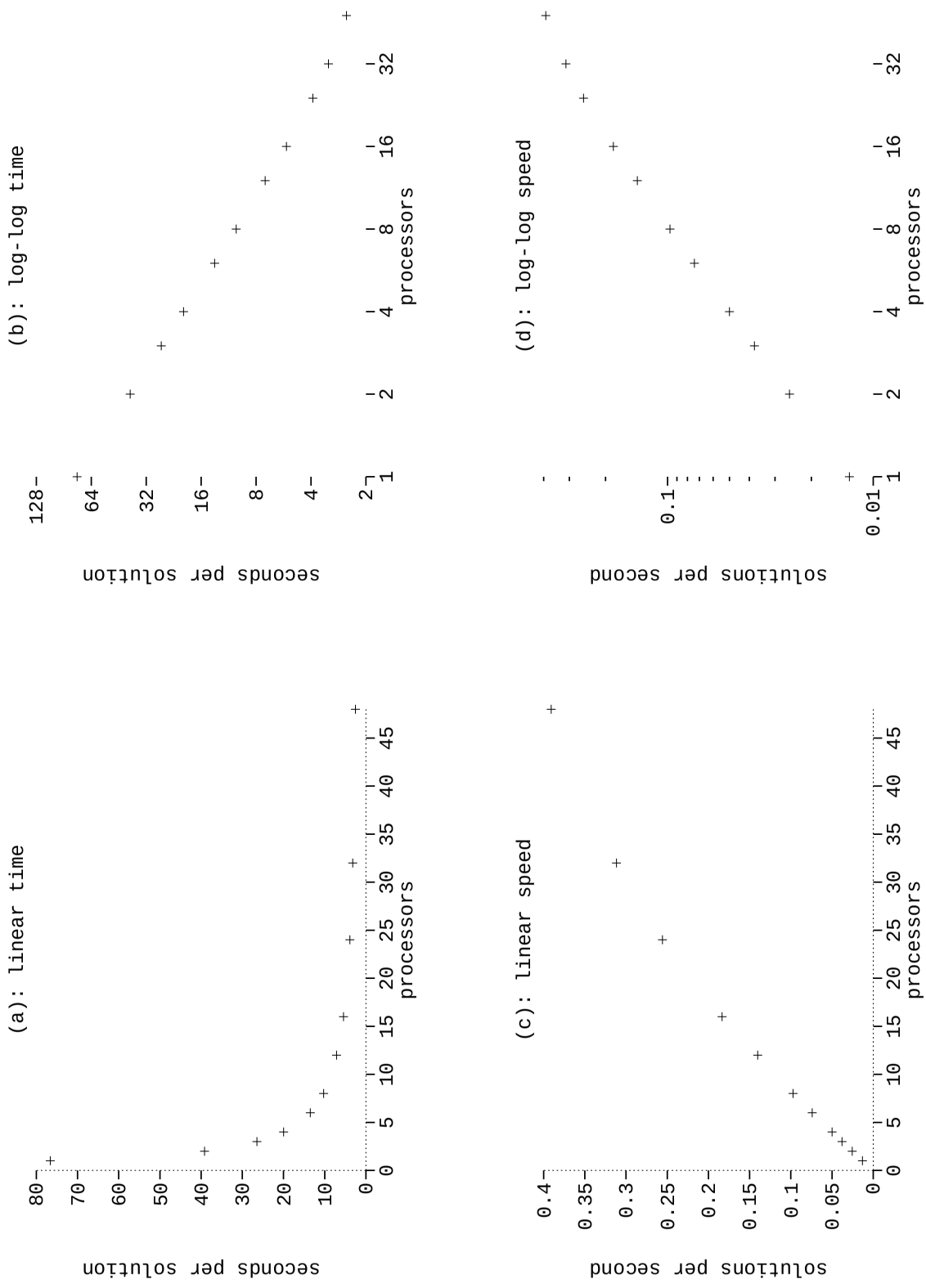

Figure 4: Showing Non-Linear Performance

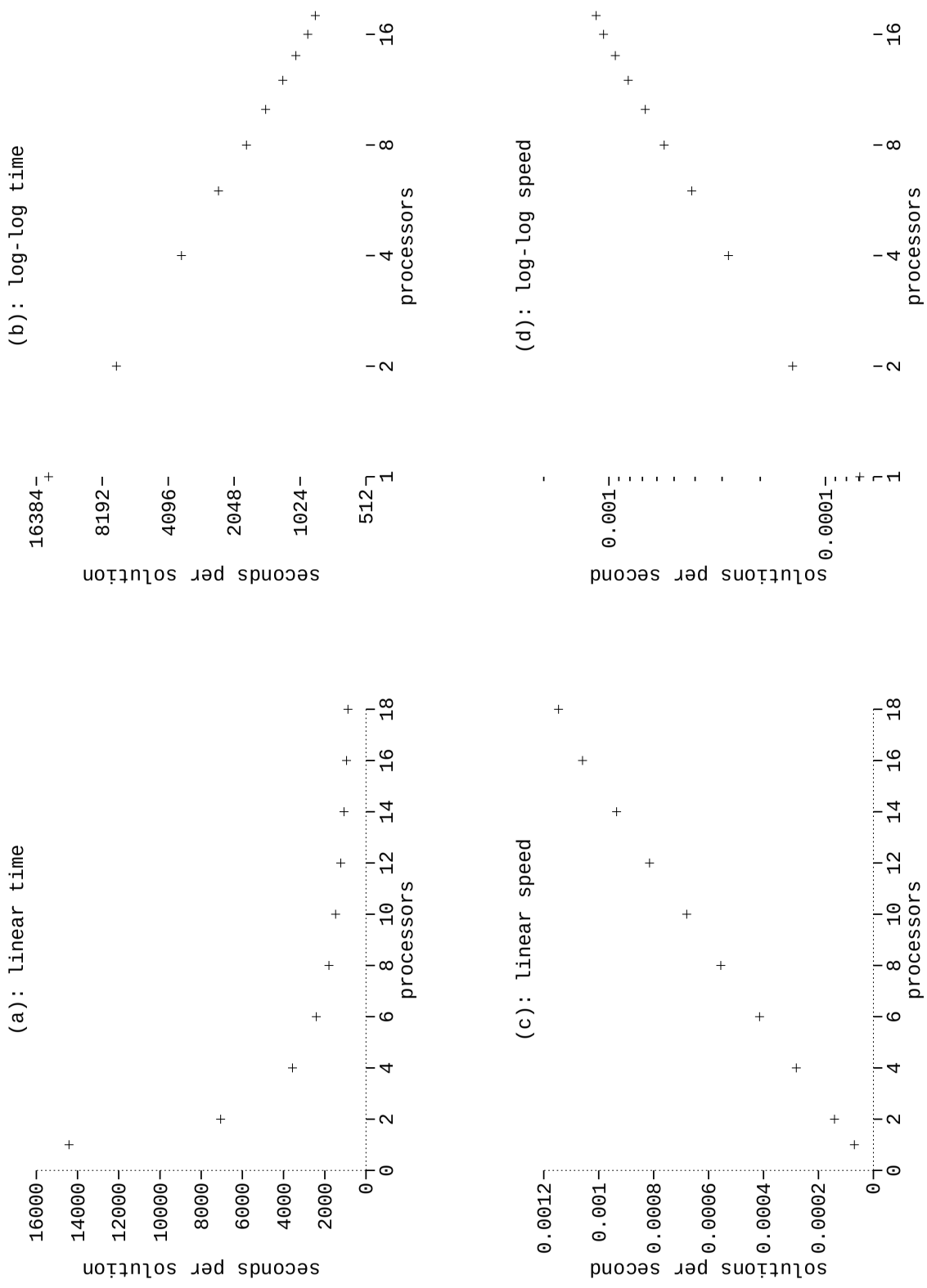

Figure 5: Showing Near-Linear Performance

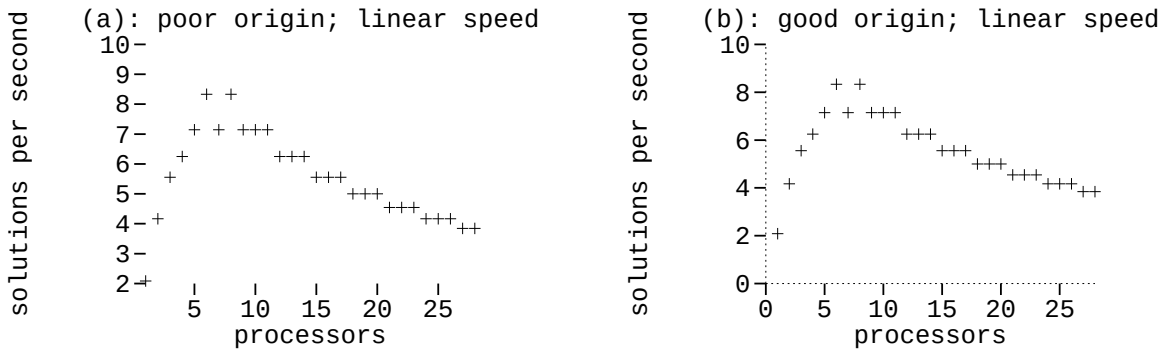

Figure 6: Zero and Non-Zero Origins

position, but it does not change shape. Linear plots do not have this property. Returning to the example from [Deshpande and Schultz, 1992], in Figure 7a, the Symmetry and Multimax appear to be getting closer together in performance, while in Figure 7c they appear to be getting further apart. Figures 7b and 7d shows the Multimax and Symmetry a constant distance apart at each processor, which shows that program performance differences are relatively constant across all numbers of processors. Since log-log plots are scale invariant, the shapes of the curves are independent of processor technology. This property is important when we compare performance on different machines (section 6).

Summary of Properties

The linear-time plot function fails to meet any of the properties mentioned above,

- qualitative changes are obvious,
- linear performance is linear on paper,
- absolute performance is linear on paper,
- relative performance is linear on paper, and
- performance is scale invariant on paper;

so you should avoid linear-time plots. The linear-speed plot shows absolute performance well, but does shows relative performance poorly and is not scale invariant. You should use linear-speed plots when absolute performance is important. The log-log plots show absolute performance poorly, but show relative performance well and are scale invariant. The log-log-speed plot is consistent with the linear-speed plot in that bigger is better; but log-log-time plots will have an opposite slope from linear-speed plots in the ideal case, which will cause less confusion when changing plot functions. For this reason, you should use the log-log-time plot function when absolute performance is not particularly important.

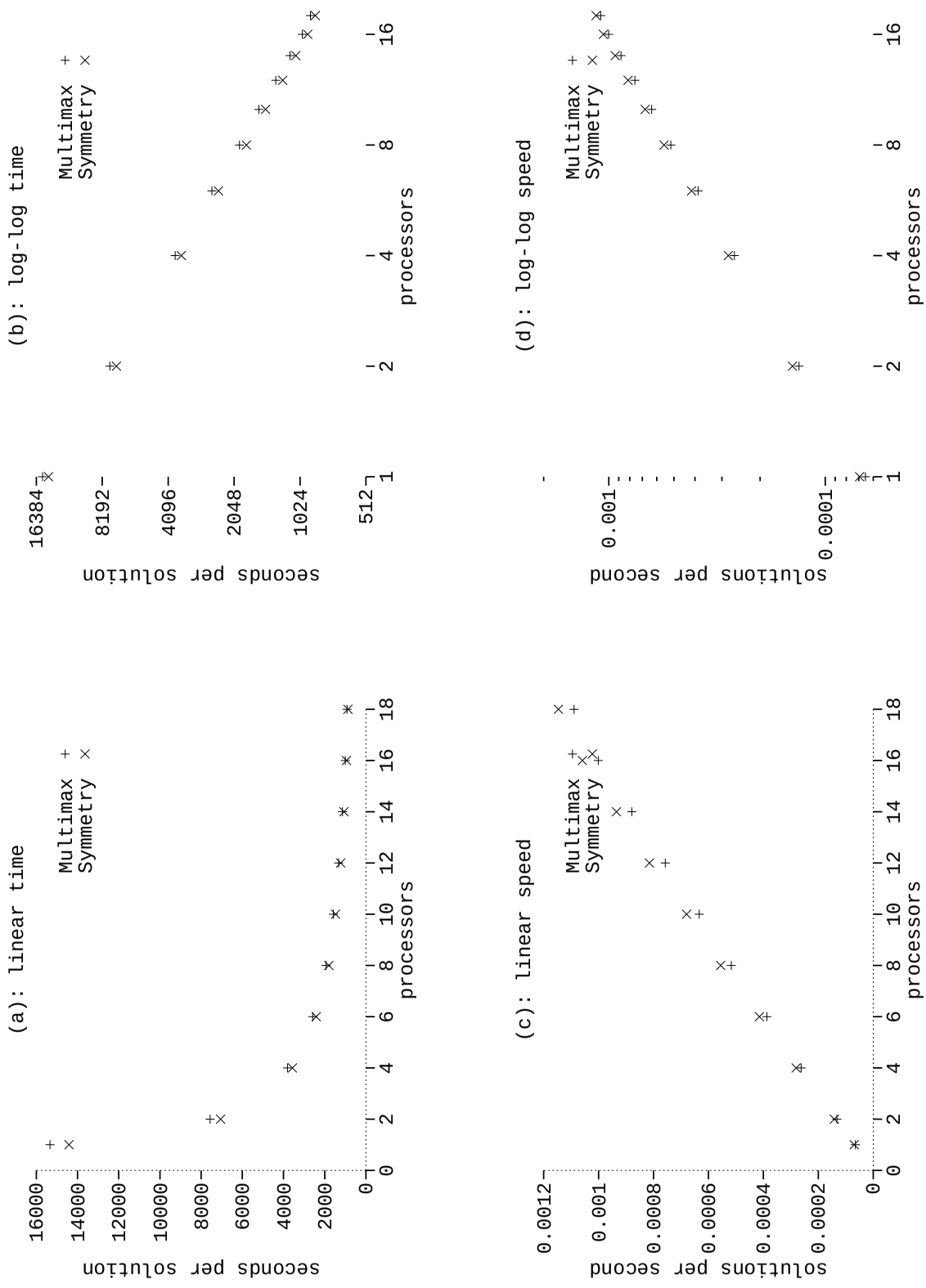

Figure 7: Linda on UMA Multiprocessors

4.3 Measure Points

Nearly all the things we can control in measuring parallel performance are discrete variables; They do not vary continuously. Graphing parallel performance with lines visually implies that performance varies continuously with the number of processors. We all know this isn't true, but we often will interpret a line to at least mean that you've collected data at every processor. To avoid implying that you've collected more data than you actually have, you should not present parallel performance lines, but instead present your measure points. For example, the line-based Figure 8a implies a smooth improvement in performance. In Figure 8b, we see that there is no data for many processor counts. The line-based graph leads us to assume the missing data in Figure 8c, but in fact the missing data are shown in Figure 8d. Plotting your measure points doesn't distort the amount of data you have collected. Of course, readers will now notice when you have sparse data, so measure at as many processor counts as resources permit and then mark those measure points on the graph.

The above advice works fine when the graph contains a single data set, or a very few widely separated datasets. However, if there are many, closely spaced, or overlapping data sets on a single graph, measure points alone are hard to interpret. In this case, you should place lines between the measure points of each data set. These lines lead the reader's eye to "connect the dots", group elements into whole data sets, and perceive the *gestalt* of a data set. Note the difference between Figures 9a and 9b. Keep in mind that the lines connect measure points, and that you must mark each measure point.

Distinct measure points alone may not separate data sets well. Sometimes you must also use distinct lines, either with patterns, as in Figure 9b, or with colors. When using colors, keep in mind some readers may be color-deficient. For example, an inability to distinguish between red and green is the most common form of color deficiency [Hurvich, 1981]. Distinguishing among many different line patterns or colors is difficult, so avoid placing many data sets on a single graph unless they are clearly separated.

Use simple glyphs that clearly show the measure point. In Figure 10 the actual measure points of $+$ and $\times$ are easier to see than those of $\Box$ and $\triangle$, which allows readers to make finer comparisons.

Error Bars and Multiple Data Points

Section 2 argued that you needed to measure multiple execution times to ensure that you have good data. In some cases, the variance (or standard deviation) in your data will be so small that you can almost ignore it in your presentation. Almost. You should state in the text that the data has an insignificant variance so that your reader is aware of the consistency of your data.

When your data is normally distributed around the mean, simply put error bars on your measure points. Error bars concisely represent the variability in normal distributions, and permit readers to visually perceive the variance relative to the data values.

When your measure points do not have a normal distribution, you should plot all the measured data. Figures 11a and 12a, show variability via error bars, but provides no clue as to the reason for the variabilities; however Figures 11b and 12b suggest the reasons. In Figure 11b (eight points per processor), for few processors most points are consistent with some outliers. As we start to

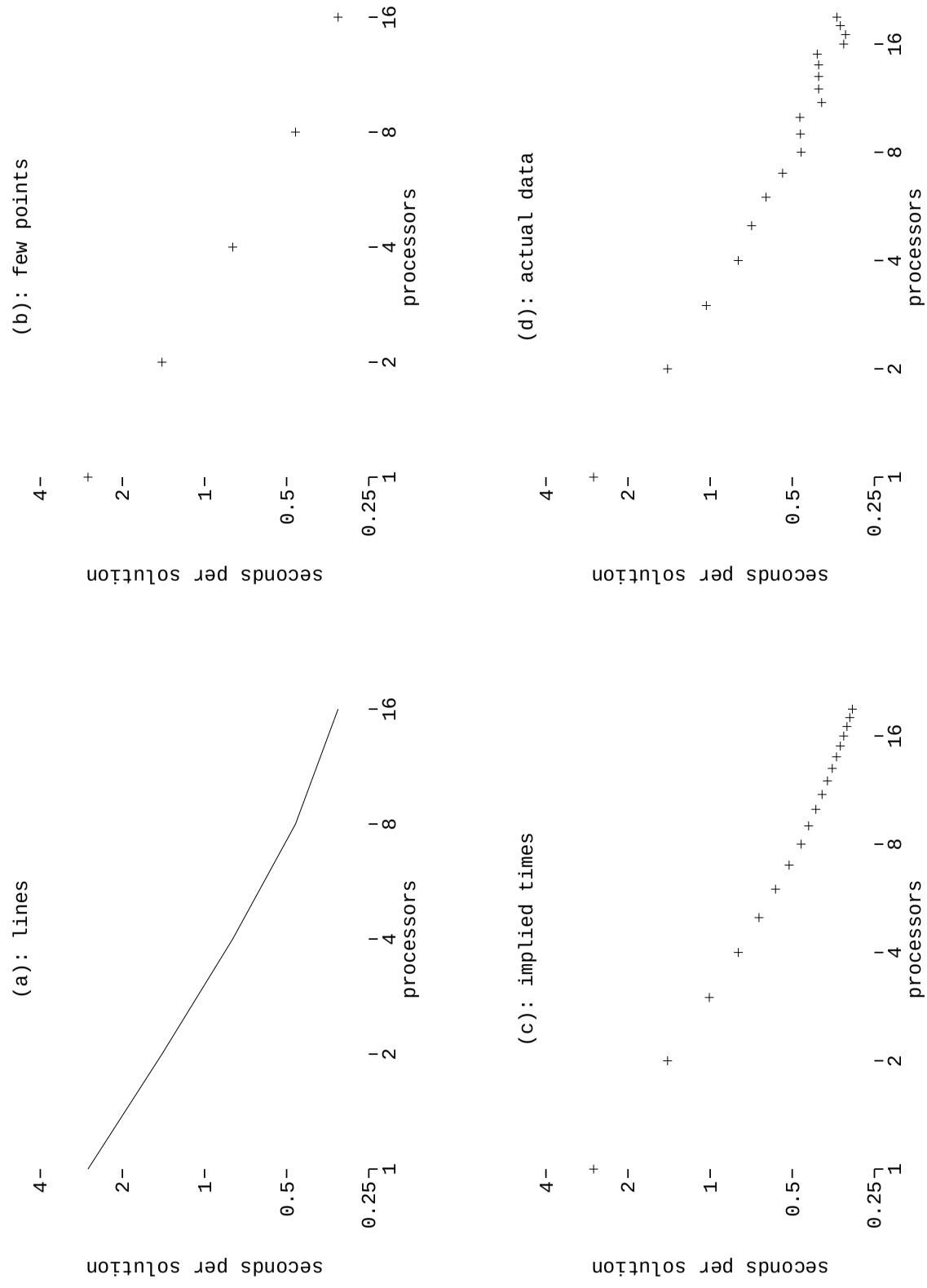

Figure 8: Showing Measure Points

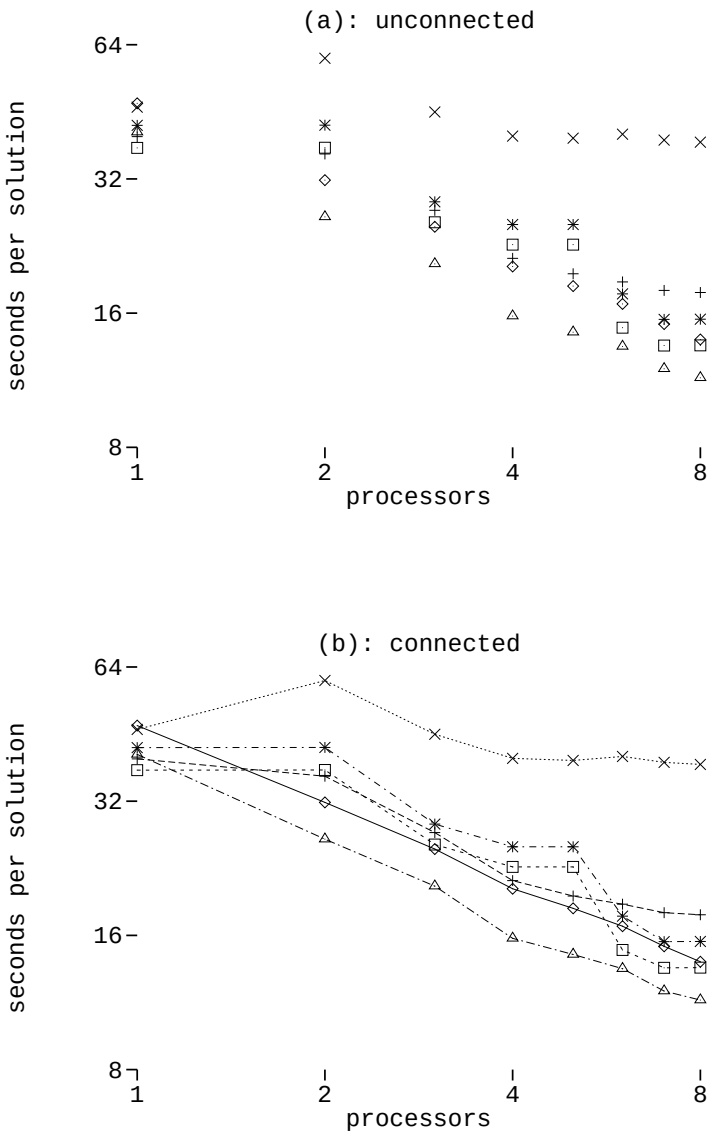

Figure 9: Connecting Lines for Multiple Data Sets

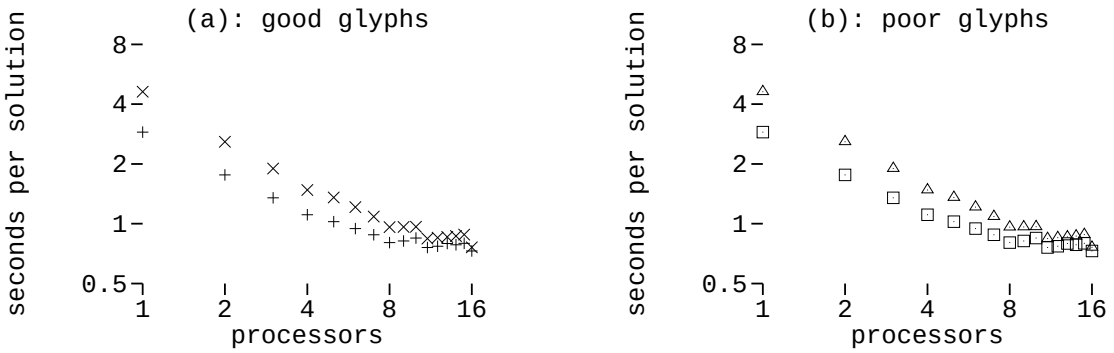

Figure 10: Glyphs With and Without Clear Measure Points

use most of the processors in the machine, points become very inconsistent. Knowing that the machine is time shared, we can infer that contention with other users is upsetting the data. In Figure 12b (sixteen points per processor), we see that data points consistently settle at one of two or three times, which is an artifact of clock granularity. If your data has significant variability, you should explain the observed variability in the text of your paper and support that explaination by presenting that variability on the graph, either through error bars or through plotting all data points.

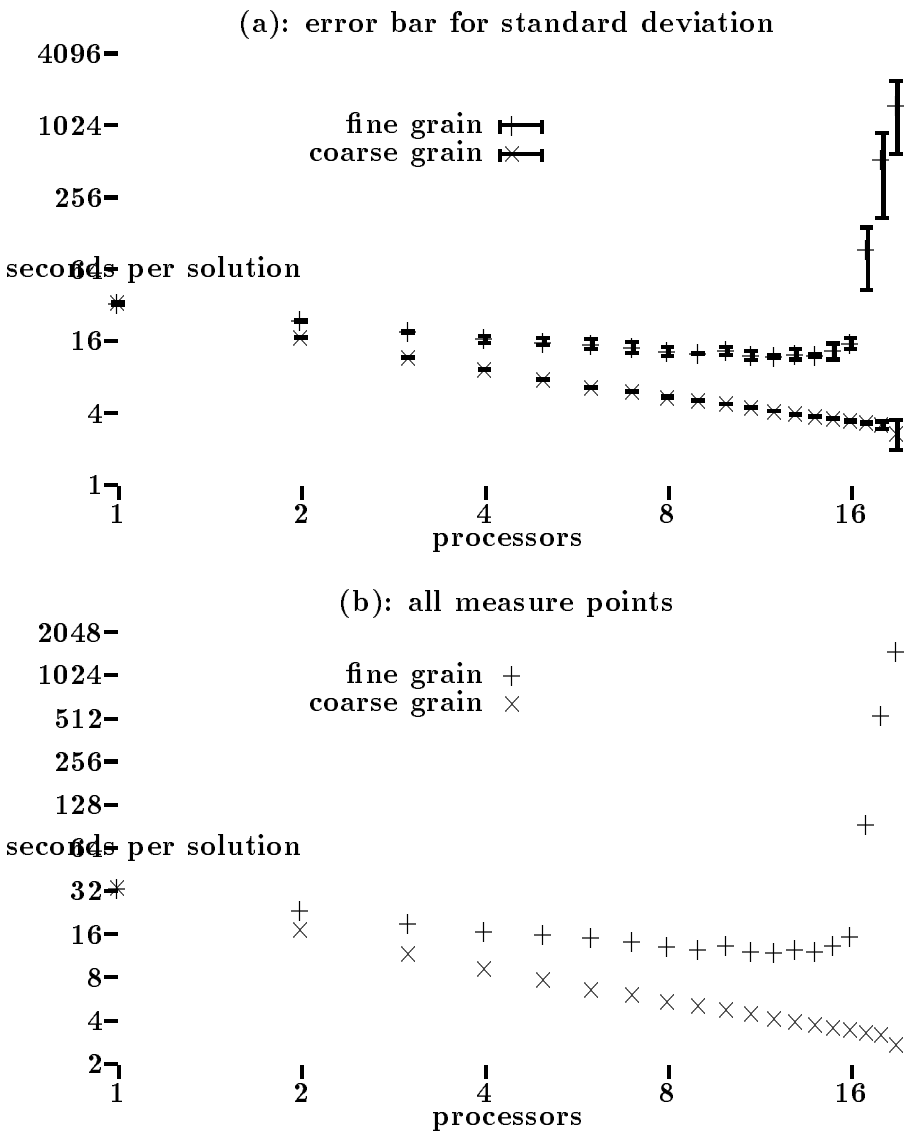

Figure 11: Showing Statistical Measures for Contention

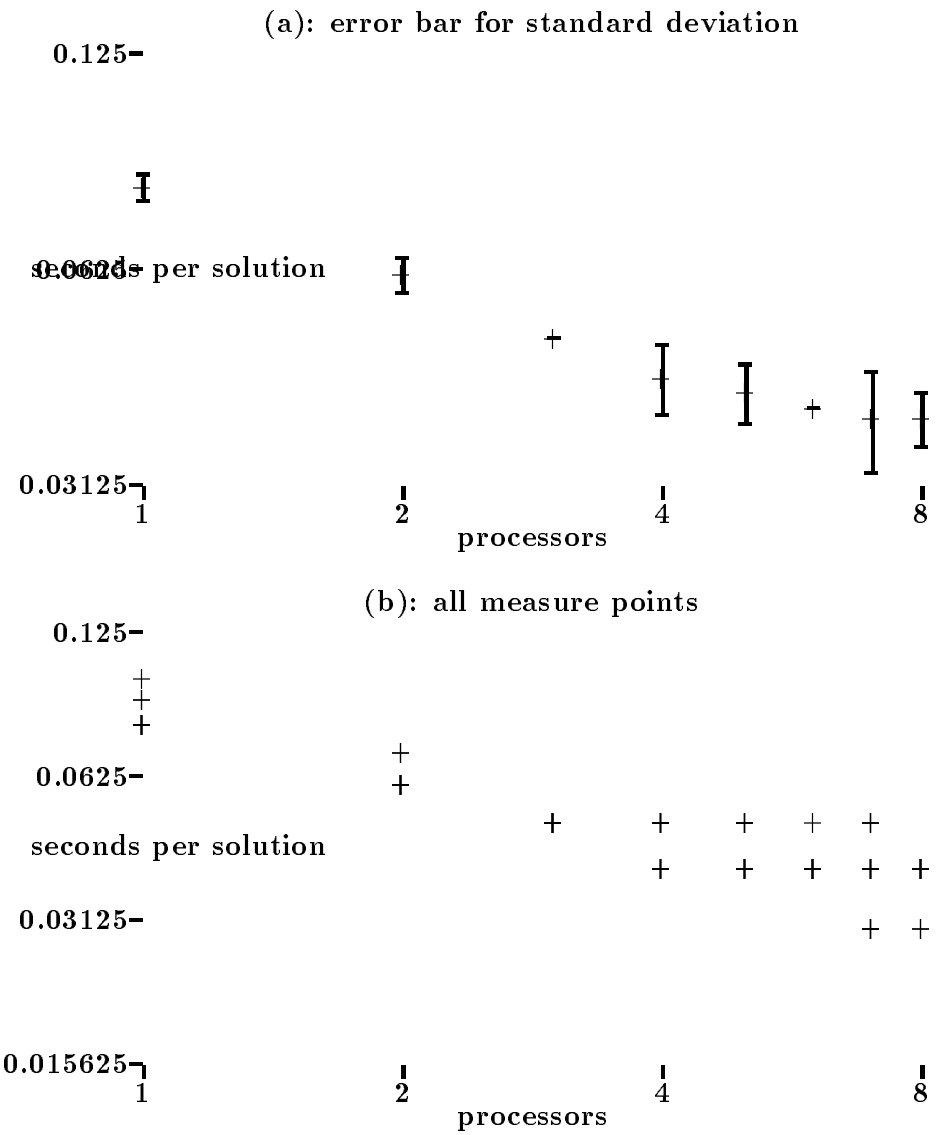

Figure 12: Showing Statistical Measures for Granularity

5 What to Compare

Presenting the parallel performance of a program is most useful in comparison with the performance of other programs. To make a comparison, we must change one (or more) of the many independent variables that go into parallel performance. The independent variables include the algorithm (as defined by the data flow), the parallelization of the algorithm, the programming language, the compiler, the automatic optimization effort, the manual optimization effort, the machine, the numeric precision, the memory available, the number of processors, the problem to be solved, the input data, etc. It is up to the writer to which variables have changed, and the significance of those changes. In general, it is best to change as few independent variables as possible. However, with so many independent variables in parallel performance, it is easy to distort performance comparisons by varying too many of them at a time. You need to hold as many variables constant as you can, and clearly document those which do change. Some of the changes will be unintentional, that is, not part of what you wish to compare. For instance, in comparing the performance of a program on two different machines, you'll likely have to use two different compilers. Clearly document these unintended changes as well the intended changes.

Bailey [1991] lists twelve ways to distort parallel performance, most of which arise from result from undocumented changes in independent variables. In [Bailey, 1992], Bailey proposes nine guidelines for reporting performance. You can find more comprehensive guidelines in [Jackson *et al.*, 1990].

6 How to Compare

How you compare performance on the printed page depends on point of the comparison. Are you comparing machines, programming systems (language and compiler), algorithms, or parallelizations of algorithms?

If you are comparing machines, programming systems, or algorithms, elapsed time is what matters and you should present that. All the reasons that make speedup a poor presentation of the performance of one parallel program also make speedup a poor way to compare two programs. Comparable speedup depends on a consistent "base", which is difficult to achieve because "bases" may differ in several independent variables. In addition, comparing speedup also fails to account for algorithms or processors that are unnecessarily slow in the sequential case. As an example, the BBN Butterfly I had no hardware floating point, so programs using a significant amount of floating point computation spent so much of their time implementing floating point operations that any reasonable communication took a very small fraction of the total time and such programs attained linear speedup quite easily. Comparing the speedup of the Butterfly I on such a program with the speedup on a machine with hardware floating point is a gross distortion. Presenting unadorned execution time also keeps changing technology in perspective — in multiprocessors, processor speed has been improving faster than communication speed [Barton and Withers, 1989], which means that newer machines will show poorer speedup despite executing programs faster. For example, our conclusions about which machine in Figure 13 is better depends on whether we look at time scaled to the time of the single-processor parallel program (13a), time scaled to the time of the sequential program (13b), or elapsed time (13c) Comparing machines, programming systems, and algorithms by elapsed time avoids distortion.

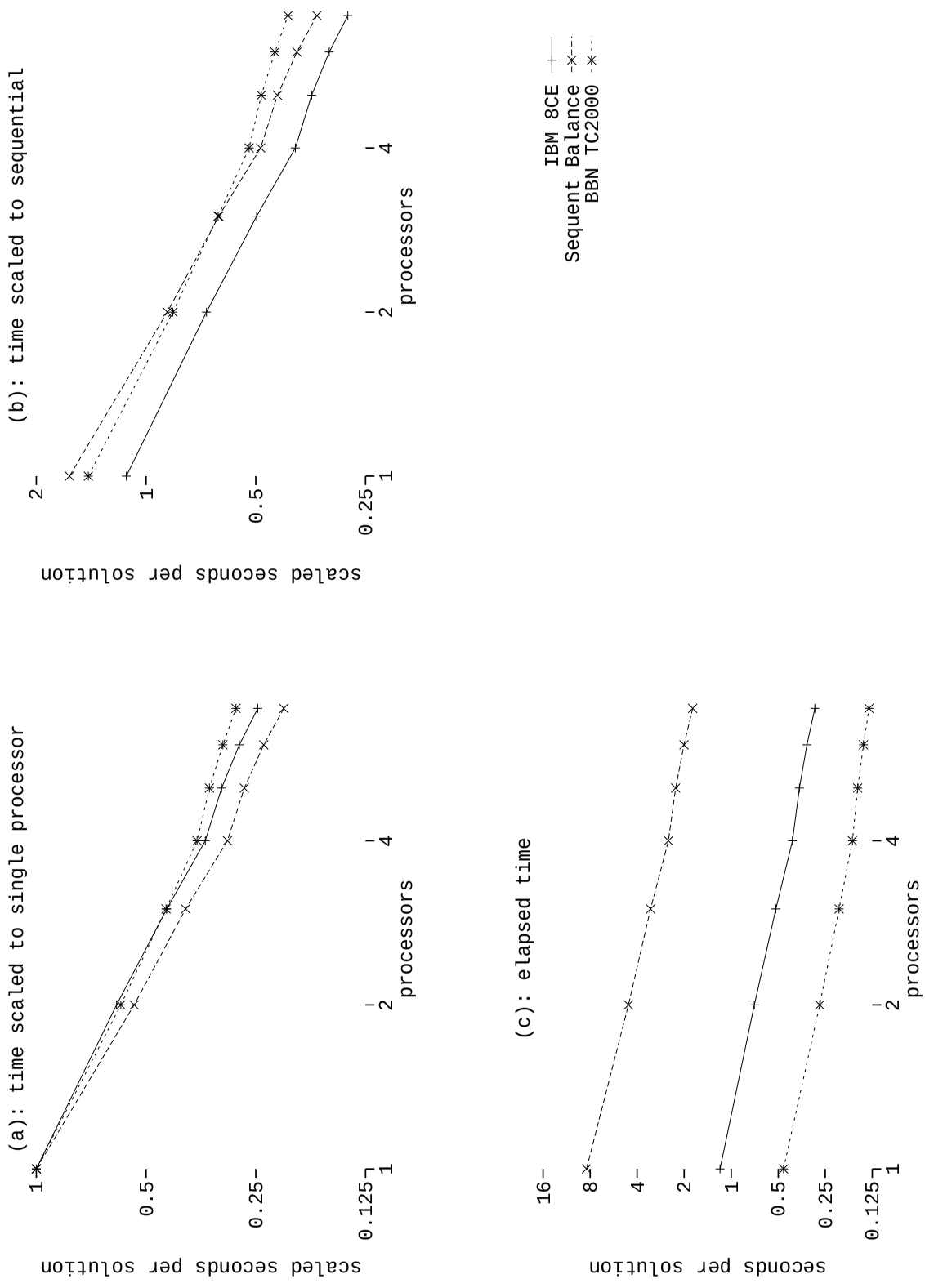

Figure 13: The Effect of Scaling on Performance Graphs

The need to compare based on speedup mostly can be satisfied by log-log plots because of their scale invariance. If you compare elapsed time on log-log plots, the shape of the curves are independent of processor technology, and you can see scaling trends for one program/machine independent of the others. For example, in Figure 14 we can see that the TC2000 is always faster than the Symmetry for small numbers of processors, but the TC2000 is not improving so the Symmetry may well perform better at larger numbers of processors. We can also see that the 8CE

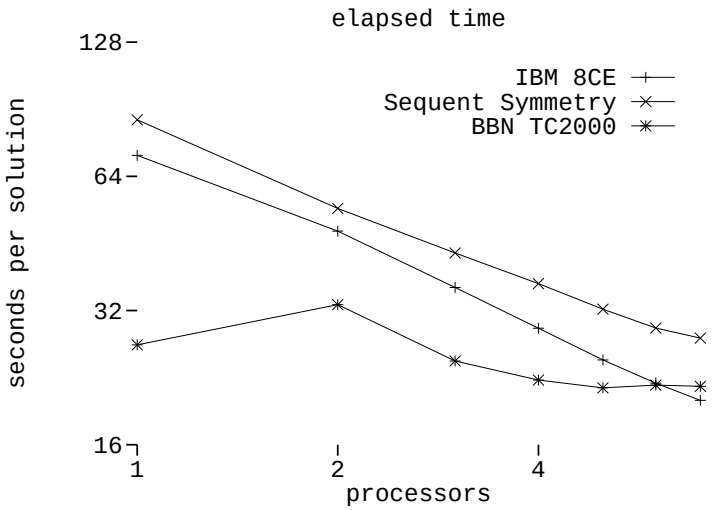

Figure 14: Elapsed Time and Scale Invariance

improves faster than the Symmetry because the slope for the 8CE is greater. Unless comparisons based on absolute performance are important, use log-log plots to compare execution times because they provide scale invariance.

If the point of your comparison is not speed, but rather how algorithms or parallelizations are influenced by machines, then scaling elapsed time *is* appropriate. However, you must be careful to scale to the execution time of a sequential program, and not the parallel program on a single processor. The overhead for creating and managing parallelism varies widely, even for a single processor, which could distort the true performance, as in Figures 13a and 13b. In this case, Figure 13b is the best representation because it explicitly presents the overhead of parallelization. So, if you are comparing the behavior of algorithms or parallelizations on different machines, use log-log time scaled to the sequential processor case.

If you are comparing different problem sizes, the straightforward approach is to plot time versus processors for each relevant problem size. Log-log plots are particularly useful here because they show the relationship between relative increases in problem size and execution time. For example (derived from [Seetharamakrishnan, 1992]), in Figure 15a we see that the quadrupling of problem size leads to a quadrupling of execution time for the single processor case, and that four processors can complete the larger problem almost the same time as a single processor can complete the smaller problem. This relationship is clearer in Figure 15b, where the lines connect a quadrupled problem size with a quadrupled number of processors. If the performance improvement were perfect, we would see horizontal lines. However, where the lines are not horizontal, a scaling problem exists.

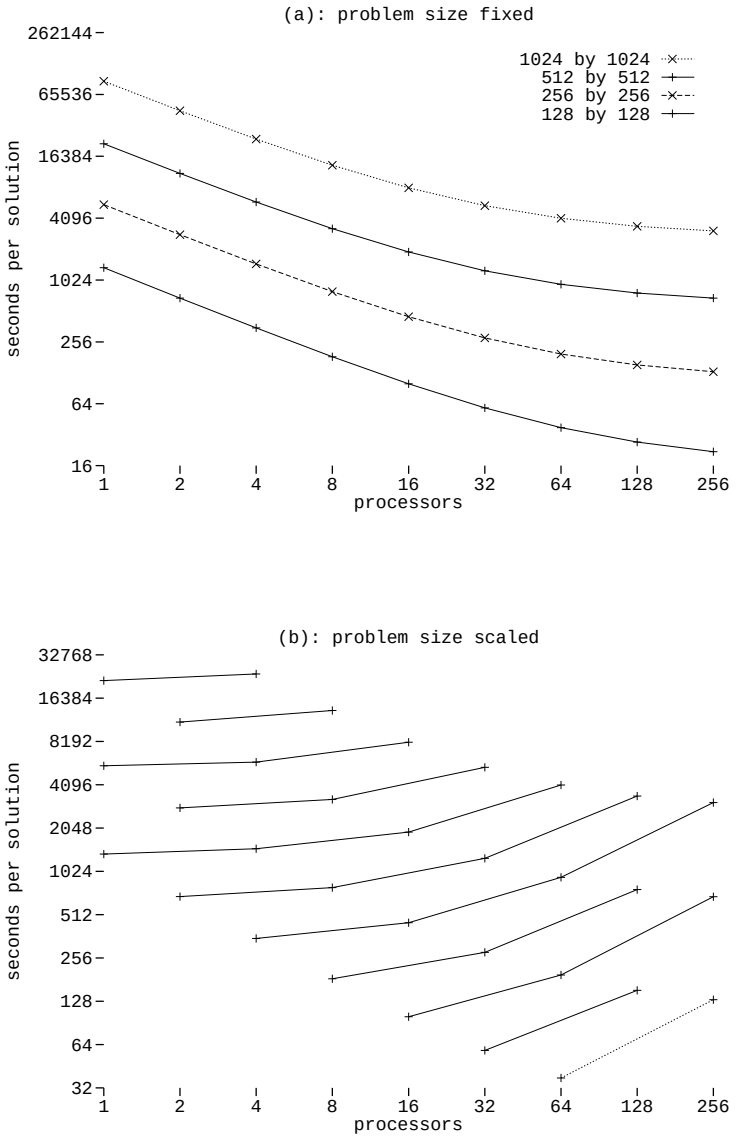

Figure 15: Scaling Problem Size

Sometimes you may wish to compare observed performance with the "perfect" performance. This usually involves scaling to a sequential program. Since the base sequential program is usually subject to controversy, a better approach than scaling the actual data is to put the extrapolated performance of a sequential program on the graph. For example, Figure 16a shows the extrapolated parallel times of the sequential program, as well as the elapsed times of the parallel program. Sometimes a program may parallelize so poorly as to require extending the unextrapolated sequential program time across the graph, as in Figure 16b. If you are studying different parallel algorithms,

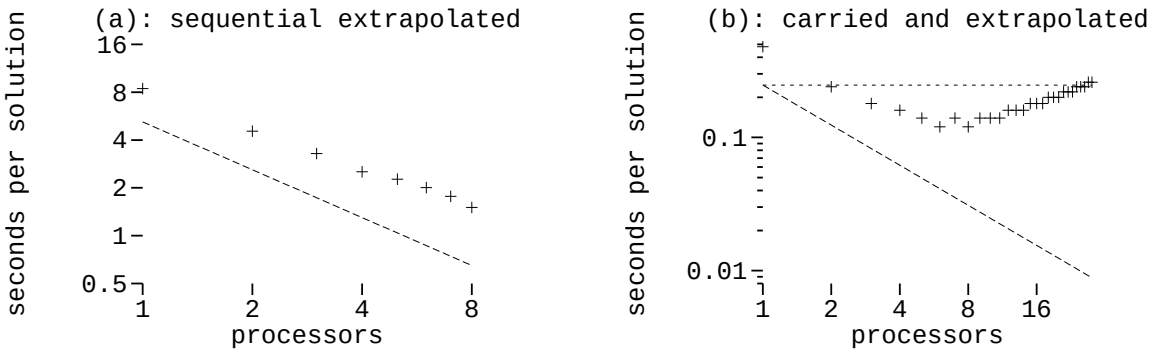

Figure 16: Showing "Perfect" Performance

you should use the time of the best known sequential program as for your extrapolation. If you are studying different parallelizations of a given algorithm, you should use the time of the sequential version of the algorithm. In either case, putting the extrapolated "perfect" performance on paper separately from the actual data helps your reader to understand the interpretation implicit in perfect performance.

7 Summary

Because performance is critical to parallel systems, people spend substantial time in investigating parallel performance, and presenting parallel performance well makes that time more effective. In this paper, I have argued that you should:

- Measure elapsed time to account for waiting time in addition to CPU time.
- Present elapsed time rather than speedup because speedup is a derivative measure, subject to different and sometimes undocumented derivations.
- Use graphs whenever you have more than a few data points.
- Use linear-speed plots (with zero origins) when absolute performance is important.
- Use log-log-time plots in every other case.
- Show measure points explicitly on paper.
- Show variability in execution time when it is significant.

- Limit the number of independent variables you change when making comparisons, and document those you do.
- Rely on the scale invariance of log-log plots when comparing programs on different machines, rather than scaling the data values beforehand.
- Show "perfect" performance as separate lines.

There are tradeoffs in presenting performance; not all these quidelines are applicable at all times. However, following these guidelines and making explicit choices when you do not follow them, should lead you to more effective presentations of parallel performance.

References

[Bailey, 1991] David H. Bailey, "Twelve Ways to Fool the Masses When Giving Performance Results on Parallel Computers," *Supercomputing Review*, 4(8):54–55, August 1991, reprinted in *Supercomputer*, September 1991, pages 4–7.

[Bailey, 1992] David H. Bailey, "Misleading Performance in the Supercomputing Field," In *Proceedings of Supercomputing '92*, pages 155–158, Minneapolis, Minnesota, November 1992.

[Barr and Hickman, 1991] Richard S. Barr and Betty L. Hickman, "On Reporting the Speedup of Parallel Algorithms: A Survey of Issues and Experts," Technical Report 91-CSE-20, Department of Computer Science and Engineering, Southern Methodist University, May 1991.

[Barton and Withers, 1989] M. L. Barton and G. R. Withers, "Computer Performance as a Function of the Speed, Quantity, and Cost of the Processors," In *Proceedings of Supercomputing '89*, pages 759–764, Reno, Nevada, November 1989.

[Crowl, 1991] Lawrence A. Crowl, "Architectural Adaptability in Parallel Programming," Technical Report 381, Department of Computer Science, University of Rochester, Rochester, New York, 14627-0226, May 1991, also [**?**].

[Deshpande and Schultz, 1992] Ashish Deshpande and Martin Schultz, "Efficient Parallel Programming with Linda," In *Proceedings of Supercomputing '92*, pages 238–244, Minneapolis, Minnesota, November 1992.

[Faber *et al.*, 1986] Vance Faber, Olaf M. Lubeck, and Andrew B. White, Jr., "Superlinear Speedup of an Efficient Sequential Algorithm is Not Possible," *Parallel Computing*, 3(3):259–260, July 1986.

[Faber *et al.*, 1987] Vance Faber, Olaf M. Lubeck, and Andrew B. White, Jr., "Comments on the Paper "Parallel Efficiency can be Greater than Unity"," *Parallel Computing*, 4(2):209–210, April 1987.

[Gustafson, 1988] John L. Gustafson, "Amdahl's Law Re-Evaluated," *Communications of the ACM*, 31(5):532–533, May 1988.

[Gustafson *et al.*, 1988] John L. Gustafson, Gary R. Montry, and Robert E. Benner, "Development of Parallel Methods for a 1024-Processor Hypercube," *SIAM Journal on Scientific and Statistical Computing*, 9(4):609–638, July 1988.

[Hurvich, 1981] Leo M. Hurvich, *Color Vision*, Sunderland, Massachusetts, 1981.

[Jackson *et al.*, 1990] R. H. F. Jackson, P. T. Boggs, S. G. Nash, and S. Powell, "Report of the Ad Hoc Committee to Revise the Guidelines for Reporting Computational Experiments in Mathematical Programming," *Mathematical Programming*, 49(3), 1990.

[Janßen, 1987] R. Janßen, "A Note on Superlinear Speedup," *Parallel Computing*, 4(2):211–213, April 1987.

[Parkinson, 1986] D. Parkinson, "Parallel Efficiency Can Be Greater Than Unity," *Parallel Computing*, 3(3):261–262, July 1986.

[Seetharamakrishnan, 1992] S Seetharamakrishnan, "Strengths and Weaknesses of Dataparallel C," Master's thesis, Department of Computer Science, Oregon State University, May 1992.

[Sun and Ni, 1992] Xian-He Sun and Lionel Ni, "Scalable Problems and Memory-Bound Speedup," *Journal of Parallel and Distributed Computing*, 1992.

[Tufte, 1983] Edward R. Tufte, *The Visual Display of Quantitative Information*, Graphics Press, Box 430, Cheshire, Connecticut 06410, 1983.

[Tufte, 1990] Edward R. Tufte, *Envisioning Information*, Graphics Press, Box 430, Cheshire, Connecticut 06410, 1990.