Guidelines for Presenting Parallel Performance

Lawrence A. Crowl

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

Technical Report 92-80-07 (revised)

November 1993

Abstract

Programmers use parallelism to improve program performance, so effective presentation of parallel performance is important. Unfortunately, common presentation practice often distorts parallel performance or requires considerable effort to interpret the results. This paper gives advice on how to measure, present, and compare parallel performance so as to reduce distortion and communicate more effectively. The primary emphasis is on visual presentation, with a secondary emphasis on data measurement.

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

Programmers use parallelism to improve the performance of programs, that is, to make them solve a single problem in less time or to solve more (or larger) problems in a fixed time. Because performance is so important to parallel computing, people devote substantial effort to measuring, presenting, and comparing the performance of parallel systems. Unfortunately, many current presentations of performance are not very effective. They often present information that requires substantial interpretation, exaggerates or diminishes performance differences, conceals important performance trends, or ignores significant computational costs. While this list of problems may seem like an account of fraud, the majority of presentation problems may well be entirely unintentional, and arise from a lack of guidelines for how to craft an effective presentation.

This paper provides a set of guidelines that help writers to clearly and effectively present parallel performance. In addition, these guidelines will help readers to better interpret and critically evaluate a writer's presentation. The guidelines cover what aspects of performance to measure (section 2), what information to present (section 3), and how to present that information on paper (section 4). Since writers often compare performance, these guidelines also cover what information to compare (section 5) and how to present that comparison on paper (section 6). Examples from actual parallel program executions illustrate the guidelines. The focus of the paper is primarily on visual presentation, and only secondarily on measurement. See [Jain, 1991] for more detailed advice on performance measurement and analysis.

In applying these guidelines, you should bear in mind three distinct goals.

Present information, not just data. Simply publishing a list of all data points collected would communicate all the data available, but would communicate the information poorly. You should evaluate the presentation in terms of the information presented, not just the data.

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. Some distortion may be inevitable, but choosing the right measures of parallel performance, presenting them well on paper, and making fair comparisons will reduce distortion.

It may not be possible to satisfy all three goals simultaneously; for example, tables of raw data generally minimize distortion, but require large amounts of time to interpret. You must choose a balance among information, space, time, and distortion. However, following these guidelines will often improve the presentation with respect to all three goals.

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.

2.1 Elapsed Time

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

single process CPU time: A common time measured is that of a single process in the system, under the assumption that the process will be somewhat representative. However, unless the operating system guarantees that all parallel processes receive exactly the same CPU time, some processes will use less CPU than others. It may not be possible to determine which processes will use more or less CPU, so the measured CPU time may be somewhat random. 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. This effect is particularly noticeable at eighteen and nineteen processes in figure 1, where not all processes receive equal amounts of CPU time and the CPU time of the measured processes (shown) may be significantly different from the average CPU time (not shown).

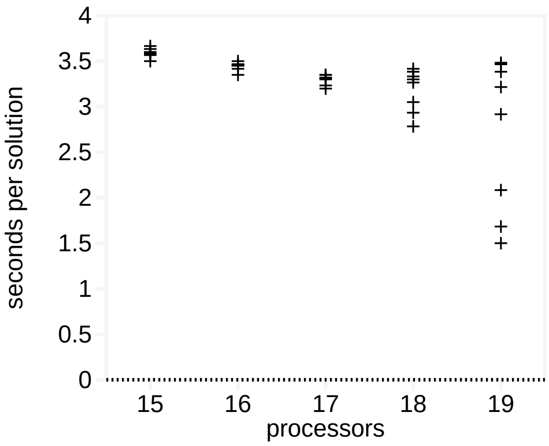

Figure 1: Measuring CPU Time of Only a Single Process.

total process CPU time: By measuring the CPU time (both system and user) 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, as in figure 2. Waiting time may be significant, and total process CPU time fails to capture it.

elapsed (wall clock) time: Most people use a parallel computer to reduce the amount of time they must wait for a solution. 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.

While CPU time may be a tempting and convenient measure, the bottom line in parallel program performance is how long people must wait for a solution, and therefore you should measure elapsed time.

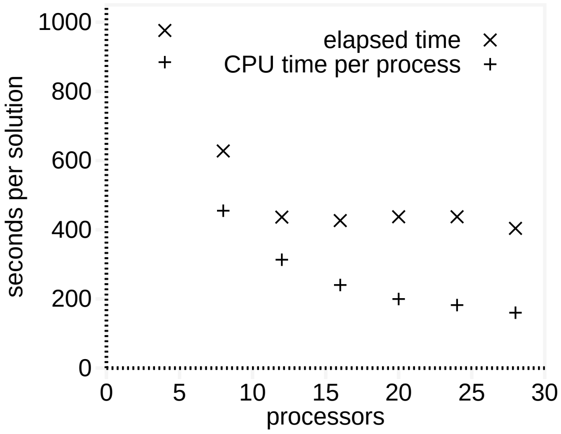

Figure 2: Elapsed Time versus CPU Time.

The data source is Ashish Deshpande and Martin Schultz, "Efficient Parallel Programming with Linda", in the *Proceedings of Supercomputing '89*, pages 759–764, Reno, Nevada, November 1989.

A consequence of using elapsed time as a measure is that most writers would prefer to show the performance of the program on a dedicated machine. This enables writers to avoid showing reduced performance due to contention with unrelated programs for machine resources. Sometimes it is not possible to measure on a dedicated machine, in which case it is tempting to avoid using elapsed time because it is 'unfair'. However, avoiding elapsed time will hide not only time spent waiting on the unrelated programs, but also the waiting time internal to the program. In these cases, you should measure and present both total CPU time and elapsed time, and then describe the mix of unrelated programs contending for system resources. By being explicit about the contention, you enable the reader to determine if the job mix is significant or not.

2.2 Time Variation

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 do not know if the one time measured is an unlikely event, and therefore a distortion of reality. So, measure variation in execution time.

2.3 Cache Transients

When making several execution runs of a parallel algorithm, particularly within a single program, the first few runs can be significantly slower than subsequent runs. This can occur because modern systems rely heavily on caches, such as instruction caches, main memory caches, virtual memory caches, file system caches, etc. This transient behavior will distort performance results when programs are measured during the first run but used multiple times, or when programs are measured multiple times, but only used once.

To avoid distortions due to caches, you should adopt one of two measurement strategies.

measure stable caches: For programs that are run repeatedly and whose cache behavior is relatively independent of the input data, you should measure several successive runs, and only measure (or record) execution time after the caches have stabilized.

measure transient caches: If the program will yield the same answer each time it runs, running the program more than once in actual practice is pointless. In these cases, measuring multiple execution times accurately requires that any caches be flushed before each run so that you include the transient cache startup behavior in each run. In addition to the memory cache, you must also worry about address translation caches, page migration and replication caches, file system caches, etc.

Occasionally the time necessary to collect transient cache timing data exceeds the time available to the experiment. In this case, you should report both the execution time of the first run, and the execution time after the cache is stable. The initial time is more accurate, but less reliable, while the steady-state time is less accurate, but more reliable. Presenting both times permits the readers to evaluate the significance of the caches to performance.

Because nearly all modern systems use caches extensively, you should document clearly the cache measurement strategy you used.

3 What to Present

Once you have a statistical measure of the elapsed time, you must decide what information 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 because it appears to present results that are independent of the specific hardware. However, speedup has several problems, which arise from the fact that it is a derivative measure and there are many choices on how to derive speedup. You cannot assume your reader will infer the 'right' speedup because there is no clear consensus among parallel programmers on how to derive speedup [Barr and Hickman, 1991]. Each time you present speedup, you force your reader to investigate how you derived speedup and how to interpret your results. To interpret speedup results, each reader must ask the following questions.

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]), memory-bound speedup (problem size grows until memory limited [Sun and Ni, 1992]), or accuracy speedup (problem parameters adjusted to increasing accuracy [Singh *et al.*, 1993])? For scaled speedup, does the problem size grow linearly with the number of processors, or does the resulting complexity grow linearly 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? (Many 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 are 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 excellent advice on how to present information on paper, and I recommend his books to any author presenting quantitative information.[1] For discussion of these issues in the context of (sequential) performance analysis, see [Jain, 1991, chapter 10].

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. It takes little space and is easy to interpret. An example table appears in figure 3(a), though with more points than I recommend. For more than three or four data points it is best to use either a bar chart or graph,[2] as in figure 3(b). Bar charts work best when the independent variable is not quantitative, such as the type of machine or 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 3 presents information more effectively than the corresponding table.

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. There are other equally valid in-

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

[2] Your audience rarely needs to see the raw data. Reserve the raw data for an appendix, or better yet, anonymous FTP.

(a) table

number of 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

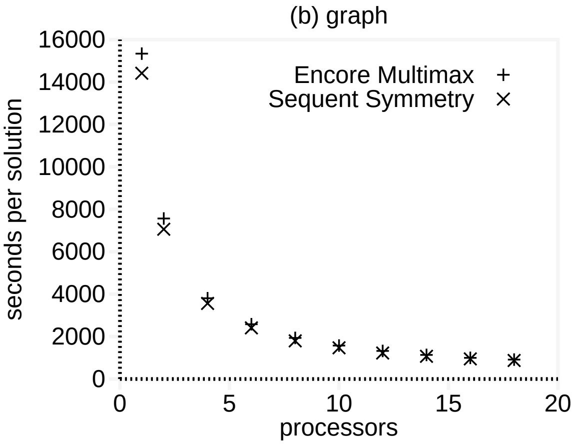

Figure 3: Table Presentation versus Graph Presentation.
The data source is Ashish Deshpande and Martin Schultz, “Efficient Parallel Programming with Linda”, in the *Proceedings of Supercomputing ’89*, pages 759–764, Reno, Nevada, November 1989.

dependent variables, such as system cost. Dependent variables may also differ depending on the application, for example profits or response time.

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. I cannot know how large your units are, or how much paper you have, so 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.

I examine each property in turn and evaluate how well each function satisfies the property.

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 4 shows the same parallel performance under each 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 plots, but difficult to see in the linear time plot. The linear time plot does not show qualitative changes well because performance at high numbers of processors receives very little visual space. Linear time plots show qualitative changes poorly, and you should avoid them.

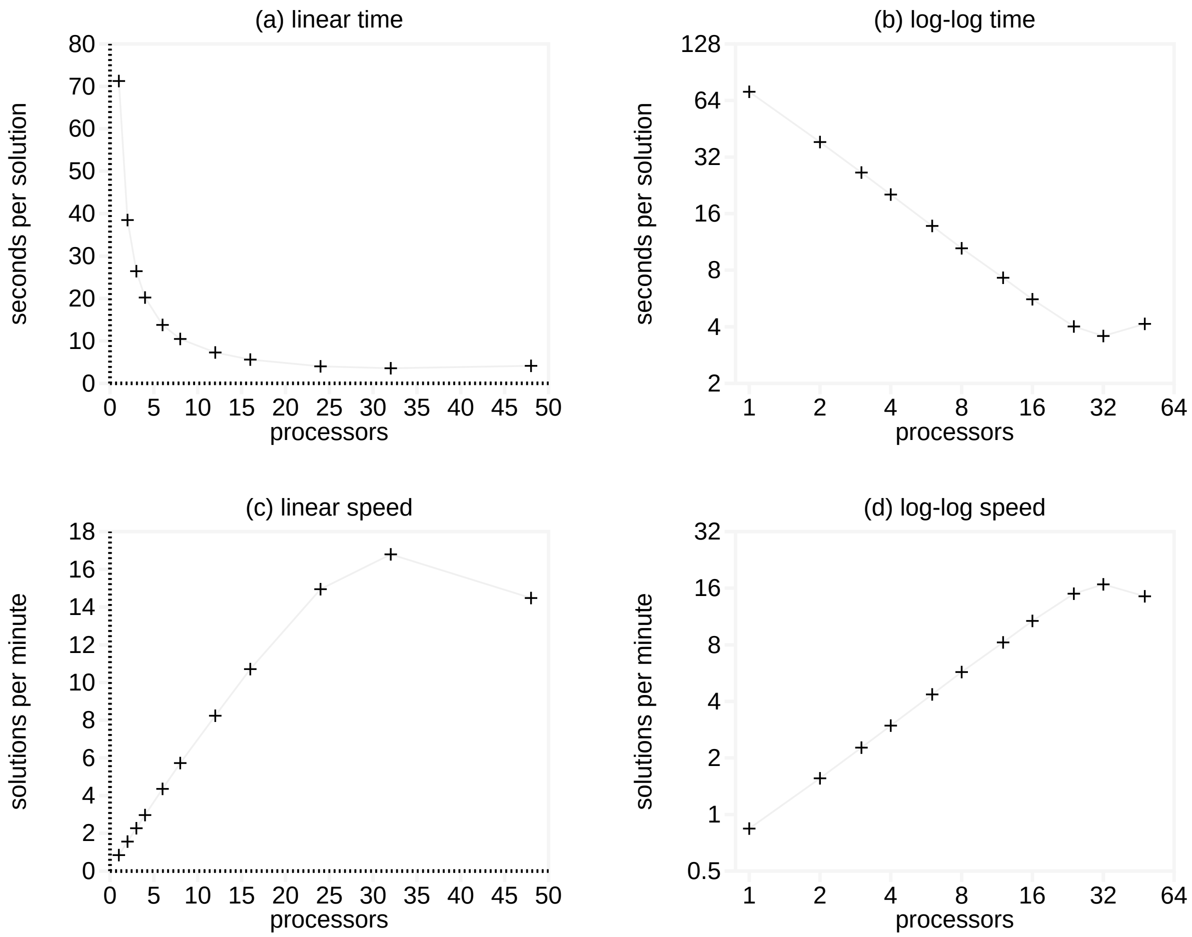

Figure 4: Showing Qualitative Changes.

The data source is Lawrence Crowl, “Architectural Adaptability in Parallel Programming”, Ph.D. Dissertation, Technical Report 381, Computer Science Department, University of Rochester, May 1991.

Linear Performance is Linear on Paper

For linear time plots, linear performance looks like the function $1/x$. Unfortunately, people are not very good at recognizing this function. For example, figure 5(a) is near linear, while figure 6(a) is not. However, their visual appearance is nearly the same, which means that people will not be able to distinguish linear performance easily on linear time plots.

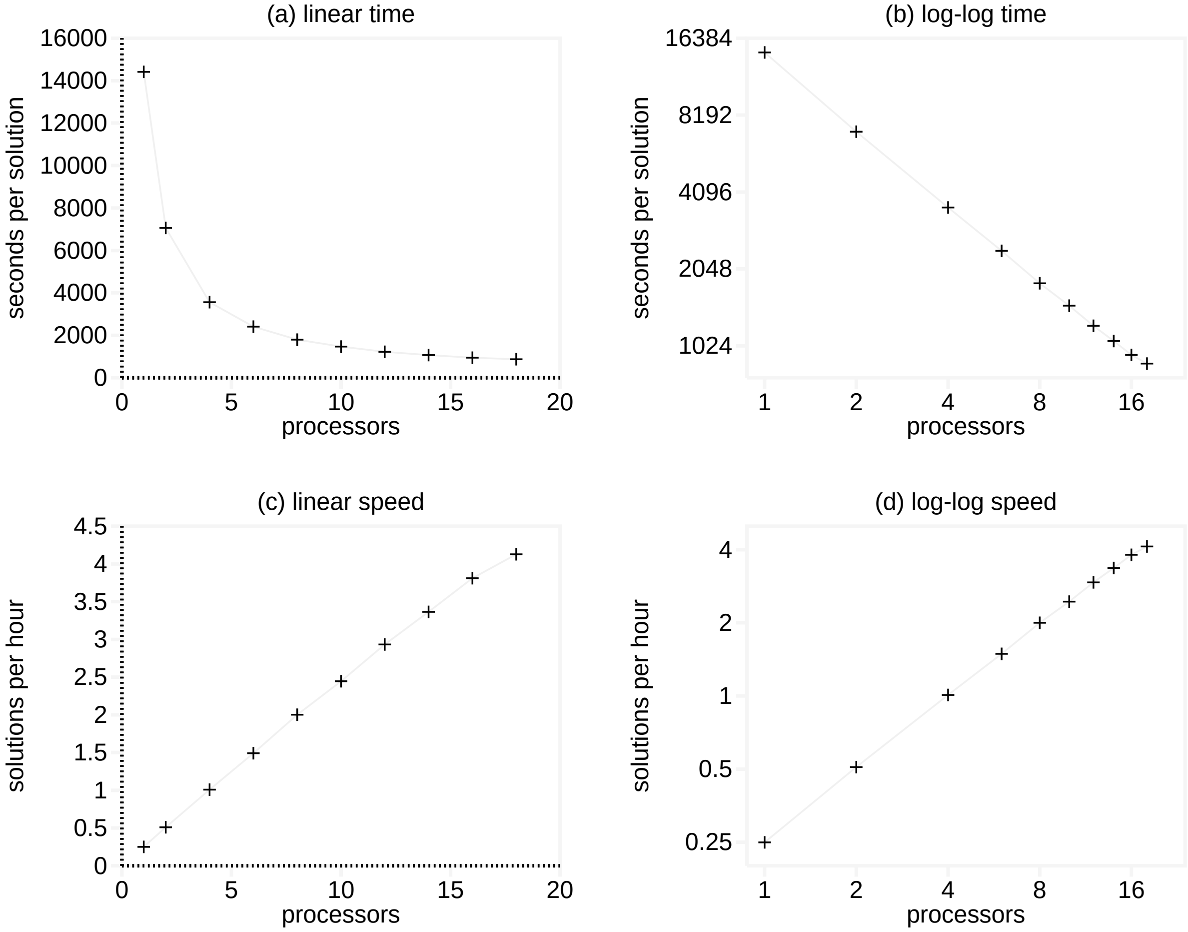

Figure 5: Showing Near-Linear Performance.
The data source is Ashish Deshpande and Martin Schultz, "Efficient Parallel Programming with Linda", in the *Proceedings of Supercomputing '89*, pages 759–764, Reno, Nevada, November 1989.

In contrast, people are good at recognizing straight lines, and the near-linear performance of figure 5 and the non-linear performance of figure 6 are quite easy to see in the linear speed, log-log time, and log-log speed plots.

To help readers evaluate how close performance is to linear, the plot function should make linear performance look linear (straight) on paper. Linear time plots do not have this property, so you should avoid linear time plots.

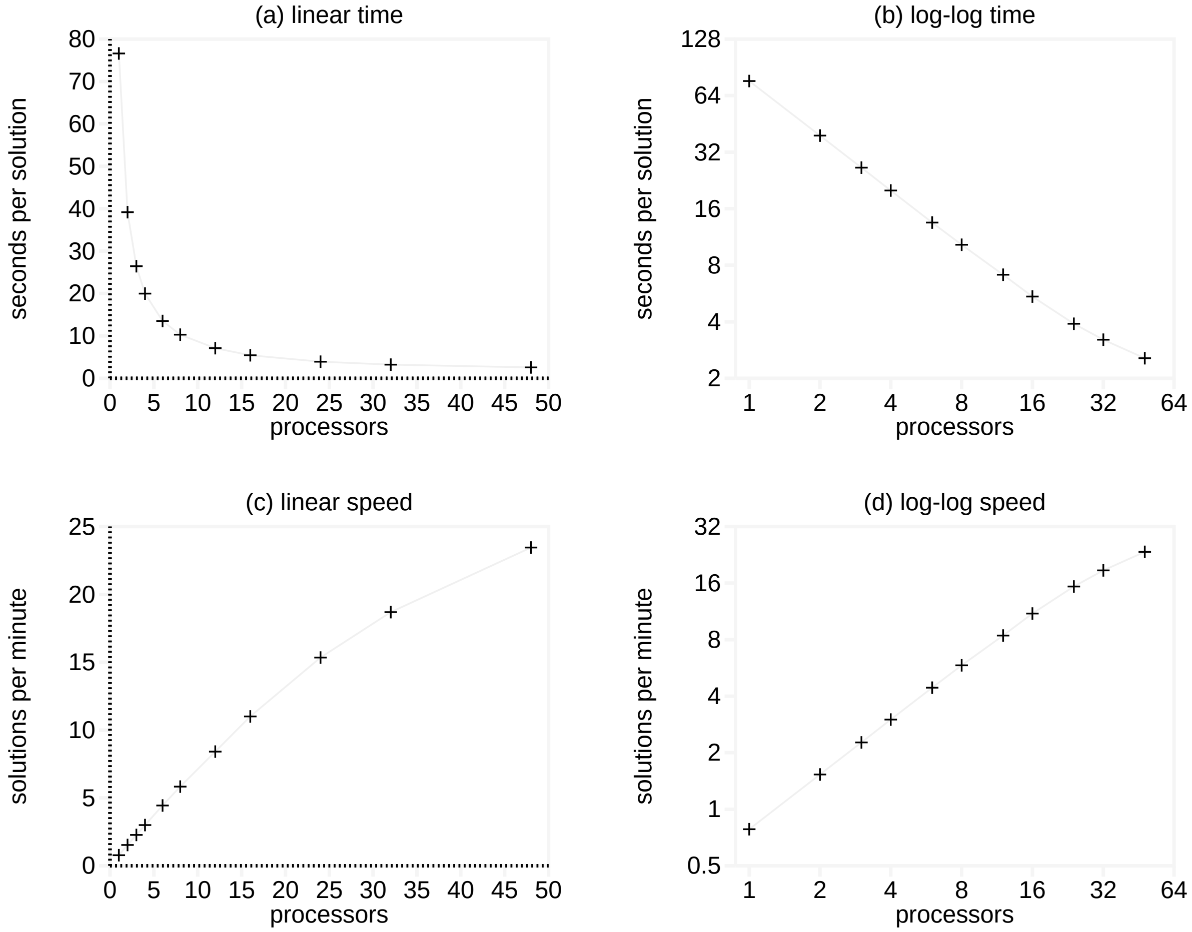

Figure 6: Showing Non-Linear Performance.
The data source is Lawrence Crowl, "Architectural Adaptability in Parallel Programming", Ph.D. Dissertation, Technical Report 381, Computer Science Department, University of Rochester, May 1991.

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 processors 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. In addition, the linear speed plot is visually similar to speedup plots, so rather than plotting speedup, you should just plot speed.

There is a danger with linear speed plots. If the origin of the dependent variable is not at zero, marginally better programs can look much better. This same problem also appears in bar charts that do not have a zero origin, or in bar charts that have a zero origin but that "tear out" the middle part of the bars. For example, figure 7(a) exaggerates the performance improvement for increasing processors. Figure 7(b) does not. So, use zero based origins.

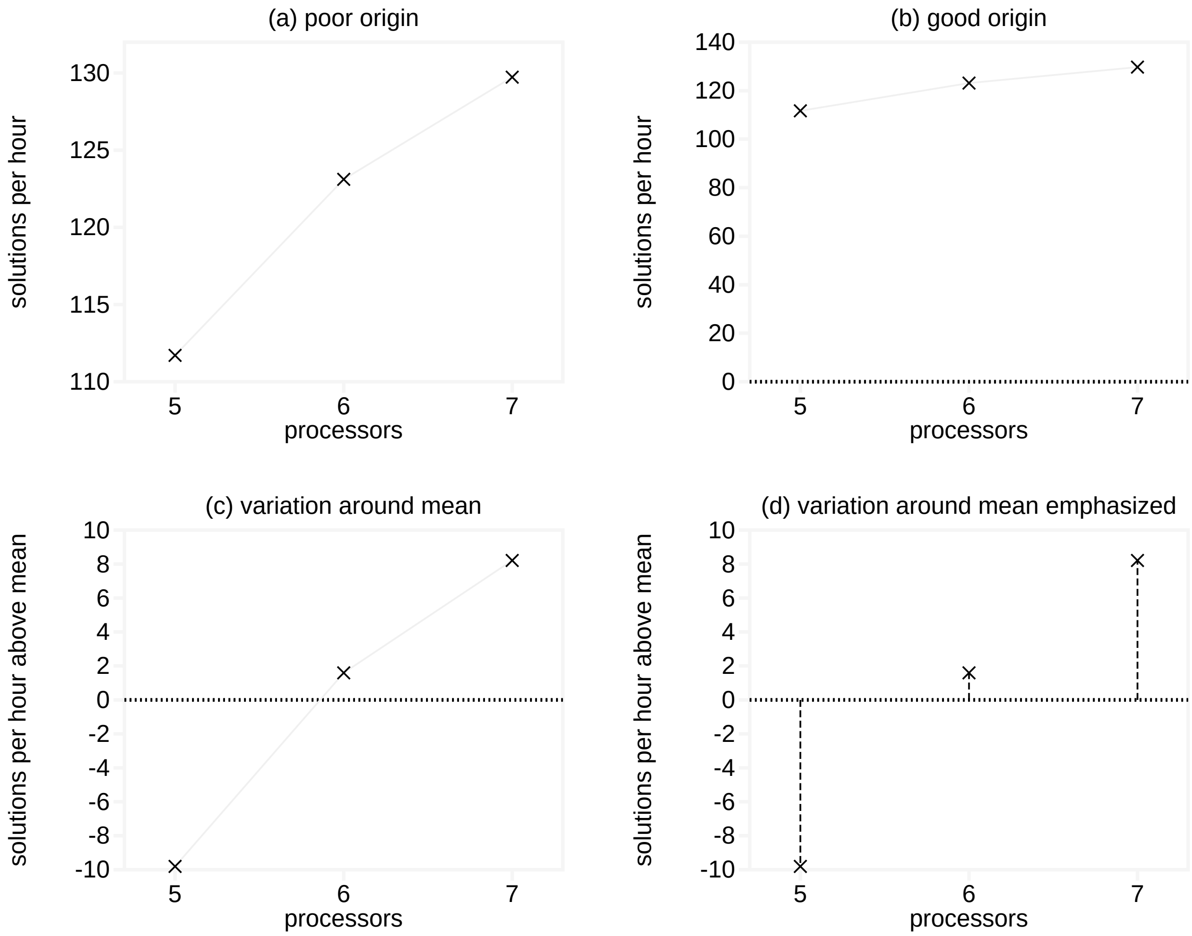

Figure 7: Emphasizing Variation in Performance.

Occasionally the variation in performance is small, and you want to emphasize the variation. Rather than use a non-zero origin, show variation around the mean execution time. Because some entries will be below the mean, as in figure 7(c), it will be clearer that the graph is showing variation, and not absolute performance. Figure 7(d) shows an even stronger emphasis on variation.

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 more easily. For example, figure 8(a) shows the speedup of a simulated machine, the DASH multiprocessor, and the ideal (relative to the parallel program on a single processor), in its originally published linear speedup form. In this graph, the simulated machine appears to be significantly (but not excessively) below the ideal performance. Figure 8(c) shows the same data with a log-log speedup plot. In this graph, we can see that the simulated machine is relatively very close to the ideal. In this case, the authors' choice of a linear speedup plot understated the quality of their results.

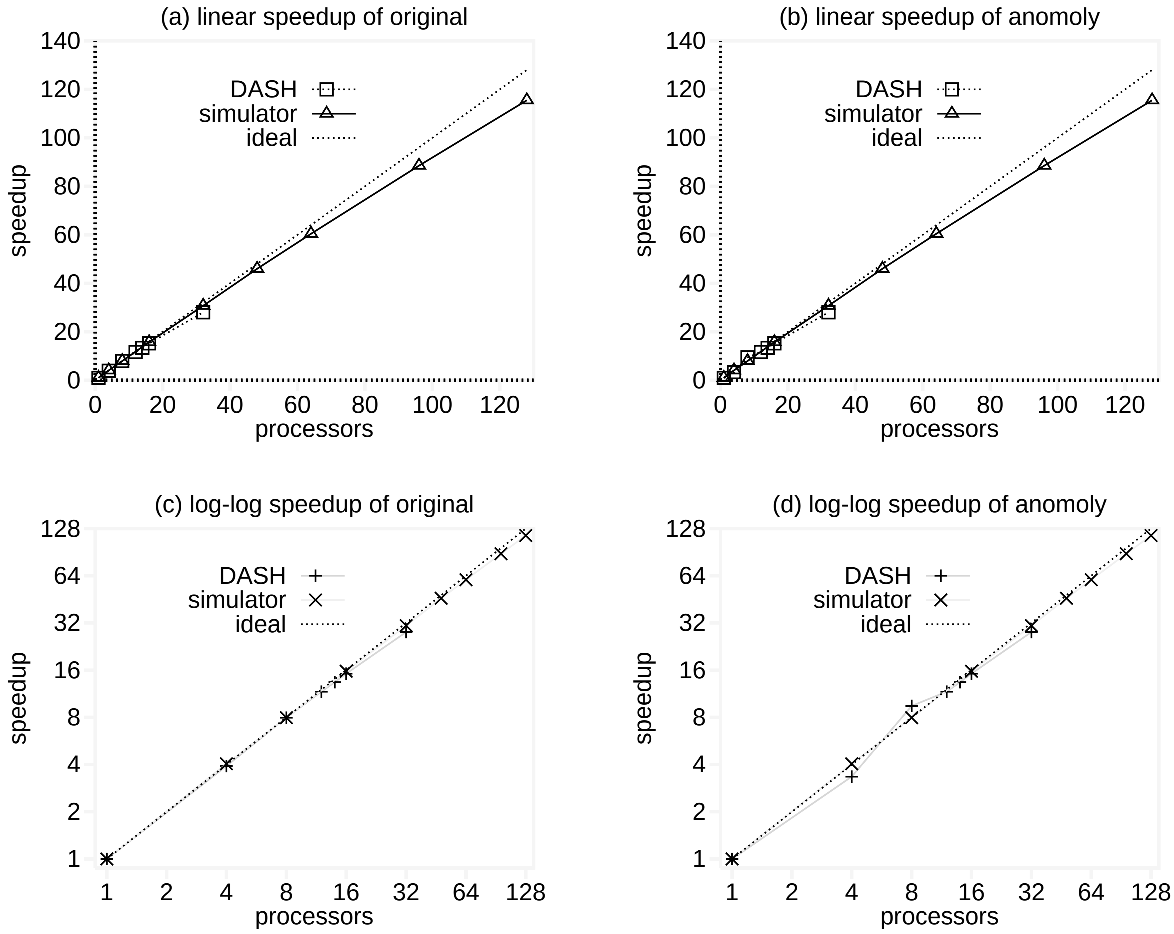

Figure 8: Showing Relative Performance with Log-Log Plots.

The data source is Jaswinder Pal Singh, John L. Hennesy, and Anoop Gupta, "Scaling Parallel Programs for Multiprocessors: Methodology and Examples", *Computer*, volume 26, number 7, pages 42–50, July 1993.

Because log-log plots show relative performance, they show performance well for both small and large numbers of processors. In contrast, 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. For example, the data for low numbers of processors in figures 8(a) and 8(b) is cluttered and very hard to distinguish. However, in figures 8(c) and 8(d), the data for low numbers of processors is much easier to see. So much easier that we can easily identify the artificially introduced anomaly for processor counts 4 and 8 in figure 8(d), but cannot easily see it in figure 8(b).

Performance is Scale Invariant on Paper

As discussed earlier, scaling performance results to the 'ideal' may distort results. To avoid the problems of scaling, we can rely on a useful property of log-log plot functions — scale invariance. Scale invariance means that doubling each value on the curve does not change the shape of the curve. The curve may change position, but it does not change shape. Linear plots do not have this property. For example, in figure 9(a), the Symmetry and Multimax appear to be getting closer together in performance, while in Figure 9(c) they appear to be getting further apart. Figures 9(b) and 9(d) show a nearly constant distance between the Multimax and Symmetry 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).

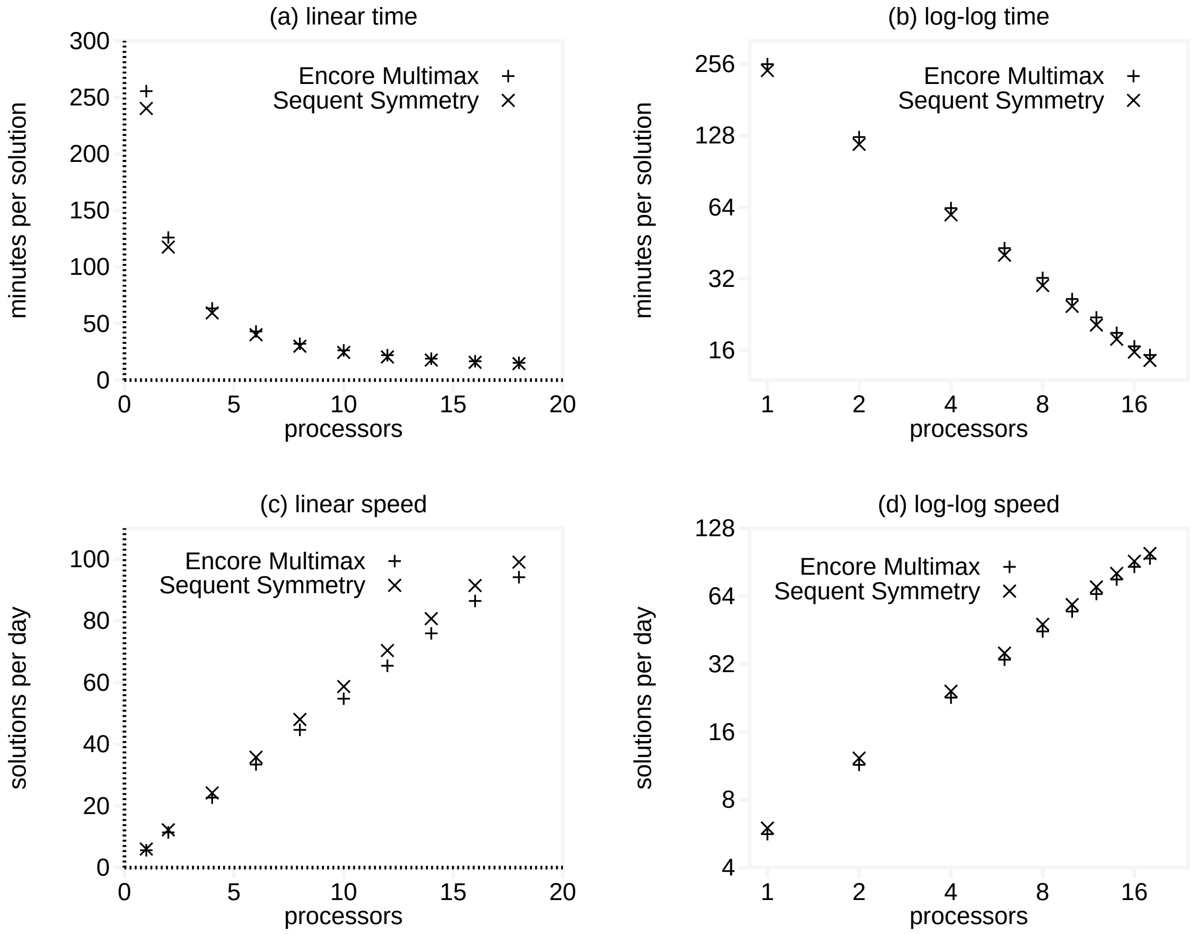

Figure 9: Showing Relative Performance with Scale Invariance.
The data source is Ashish Deshpande and Martin Schultz, "Efficient Parallel Programming with Linda", in the *Proceedings of Supercomputing '89*, pages 759–764, Reno, Nevada, November 1989.

property	linear time	linear speed	log-log time	log-log speed
qualitative changes are obvious		$\surd$	$\surd$	$\surd$
linear performance is linear on paper		$\surd$	$\surd$	$\surd$
absolute performance is linear on paper		$\surd$		
relative performance is linear on paper			$\surd$	$\surd$
performance is scale invariant on paper			$\surd$	$\surd$

Table 1: Plot Properties and Functions

Summary of Properties

Table 1 summarizes the desirable plot properties and the plot functions that satisfy those properties. The linear time plot function fails to meet any of the desired properties; so you should avoid linear time plots. The linear speed plot shows absolute performance well, but shows relative performance poorly and is not scale invariant; so you should use linear speed plots when absolute performance is important, but not otherwise. The log-log plots show absolute performance poorly, but show relative performance well and are scale invariant; so you should use log-log plots when absolute performance is not important.

The above properties do not distinguish between log-log time plots and log-log speed plots. Log-log speed plots and linear speed plots are consistent in that improving performance is positively sloped. However, the (usually) opposite slopes of log-log time plots and linear speed plots makes them easier to distinguish. For this reason, I suggest using log-log time plots rather than log-log speed plots.

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 is not true, but we often will interpret a line to at least mean that you have collected data at every processor. To avoid implying that you have 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 10(a) implies a smooth improvement in performance. In figure 10(b), we see that there are no data for many processor counts. The line-based graph leads us to assume the missing data in figure 10(c), but in fact the missing data are shown in figure 10(d). Plotting your measure points does not 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.

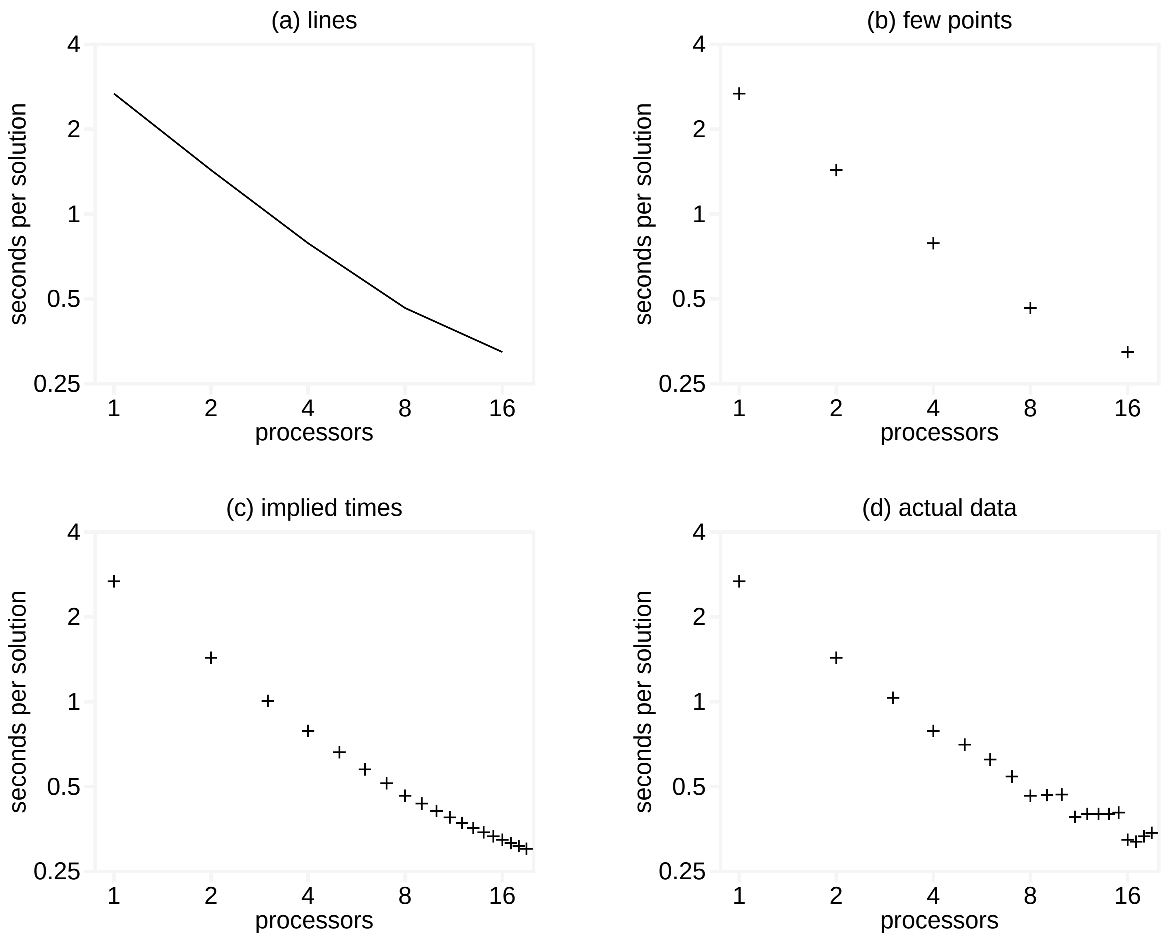

Figure 10: Showing Measure Points.

The above advice works fine when the graph contains a single data set, or a very few widely separated data sets. 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 11(a) and 11(b). Keep in mind that the lines connect measure points, and that you must still 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 11(b), 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.

In connecting discrete data points with lines, you must be careful to avoid encouraging the reader to view the distance between *lines* rather than the distance between *points*. For example, in figure 12(a), the heavy lines connecting the data points encourages people to measure the difference

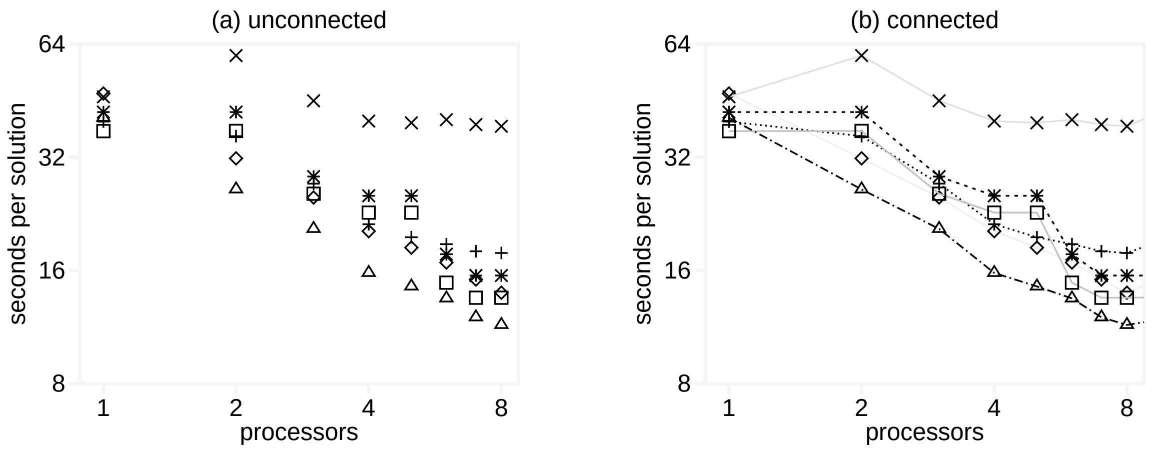

Figure 11: Connecting Lines for Multiple Data Sets.

in performance by the perpendicular distance between the two lines. As a result, the performance of the two approaches appear much closer than they really are at lower numbers of processors. In contrast, figure 12(b), uses a muted line to connect the data points, which leaves the eye to focus on the bold data points, which leads people to measure the distance between the data points. In this particular case, the connecting lines add no information to the graph and should be omitted, as they were in figure 2. In contrast, the connecting lines of figure 6 help to highlight the trends, and should be included. Muted lines connect data and highlight trend lines well, without overwhelming the data points and misleading the eye.

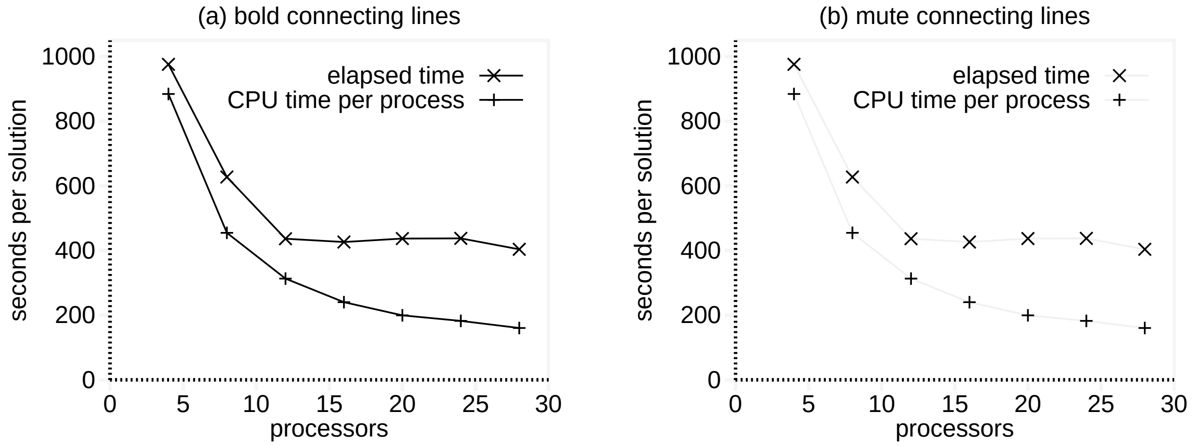

Figure 12: Bold and Muted Connecting Lines.

The data source is Ashish Deshpande and Martin Schultz, “Efficient Parallel Programming with Linda”, in the *Proceedings of Supercomputing '89*, pages 759–764, Reno, Nevada, November 1989.

Heavy graph border lines also encourage the reader's eye to measure the distance between the graph lines and the border. For log-log plots, the distance between graph lines and border lines is meaningless. You can avoid a meaningless comparison by muting border lines as well as connecting lines.

Use simple glyphs that clearly show the measure point. In figure 13 the actual measure points of $+$ and $\times$ are easier to see than those of $\square$ and $\triangle$, which allows readers to make finer comparisons.

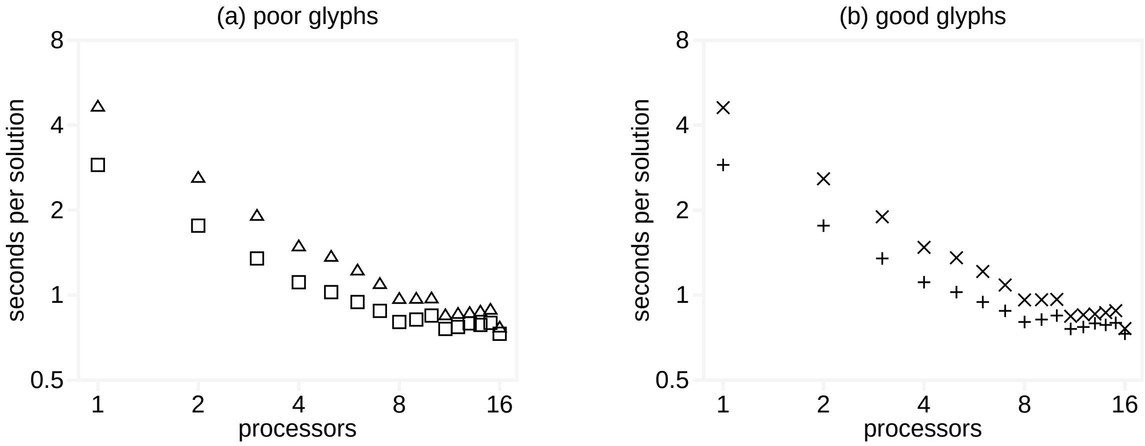

Figure 13: Glyphs Without and With Clear Measure Points.

Error Bars and Multiple Data Points

Section 2.2 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. An error bar shows the mean value and a vertical bar indicating the mean plus or minus one standard deviation. Figures 14(a) and 15(a) illustrate error bars. 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 14(a) and 15(a), show variability via error bars, but provides no clue as to the reason for the variabilities; however figures 14(b) and 15(b) suggest the reasons. In figure 14(b) (eight points per processor), for few processors most points are consistent, but with some outliers. As we start to 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 15(b) there are sixteen points per processor, but at most three are visible because the rest have exactly the same values. This exactness can be explained by clock granularity, which gives us an indication that there could be timing problems. If your data has significant variability, you should explain the observed variability in the text of your paper and support that explanation by presenting that variability on the graph, either through error bars or through plotting all data points.

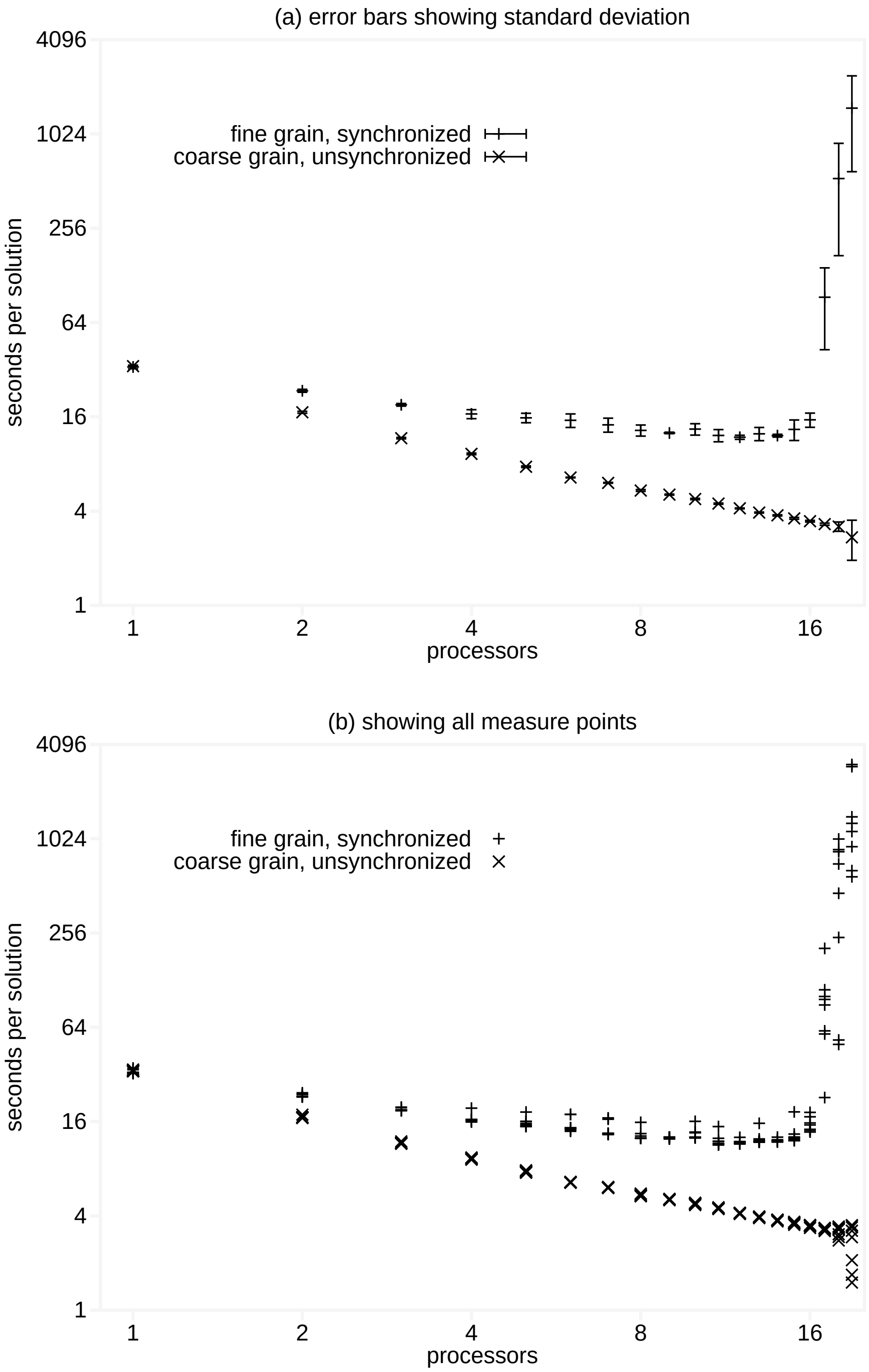

Figure 14: Showing Statistical Measures with Contention.

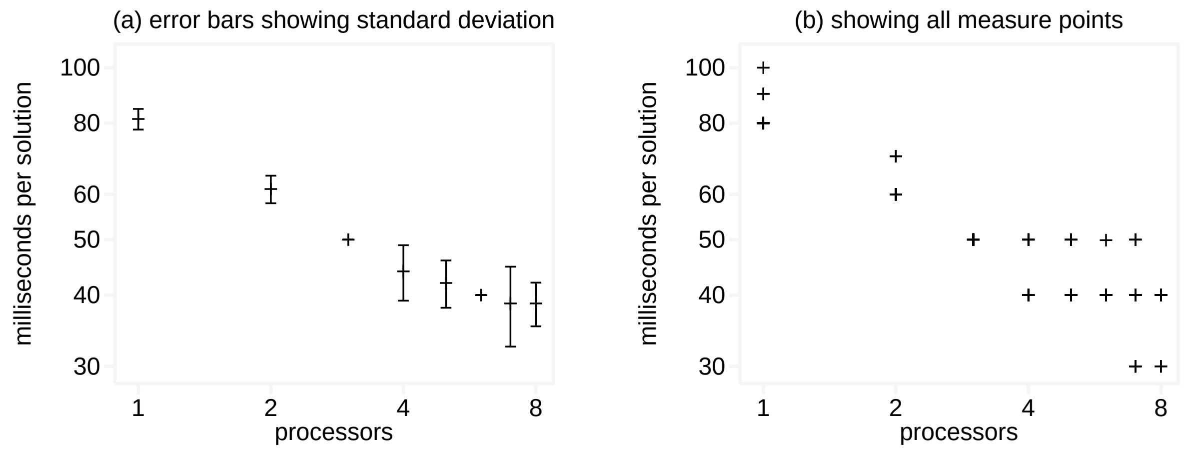

Figure 15: Showing Statistical Measures with Granularity.

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. To make reasonable conclusions from a comparison, your reader must know *all* the independent variables which changed, not just the primary variable. For example, if I tell you that a Mandelbrot program executes faster on one machine than another, you can (and probably will) conclude that the first machine is faster (for this kind of application). If I tell you that the program is optimized for the first machine, and not for the second, you can no longer conclude the first machine is faster. That extra information changes the conclusions substantially.

With so many independent variables in parallel performance, it is easy to dilute results (or distort performance) by changing too many independent variables at once. It is best to change as few independent variables as possible, which makes your results stronger. Some changes may be unavoidable, such as using two different compilers to execute a program on two different machines. To avoid misleading your reader, you should clearly document those variables that change but are not part of the comparison.

Bailey [1991] lists twelve ways to distort parallel performance, most of which arise from undocumented changes in independent variables. To help avoid these distortions, Bailey [1992] proposes nine guidelines for reporting performance. In summary, they are:

1. For a well-known benchmark, compare results only for the same version of the benchmark.
2. Do not compare projections or extrapolations with actual performance rates.
3. Only compare results with similar efforts in tuning performance.
4. Compare run times, preferably true time-of-day, not megaFLOPS, etc.

5. Only present megaFLOPS rates when based on consistent FLOP counts.
6. Speedup should be based on a reasonably tuned sequential program. Clearly identify scaled speedup.
7. Disclose ancillary information that would significantly affect the interpretation of results.
8. Include any important qualifying information in abstracts, figures and tables.
9. Include all the information necessary for other scientists to accurately reproduce the performance results presented in the paper.

You can find more comprehensive guidelines in [Jackson *et al.*, 1990].

In summary, you should minimize the number of independent variables that change in comparisons, and fully disclose all information related to the computational experiment, including hardware and software, and particularly those independent variables that change but are not the point of the comparison.

6 How to Compare

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

6.1 Machines, Programming Systems, or 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 floating point hardware, 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 16 is better depends on whether we look at time scaled to the time of the single-processor parallel program (figure 16(a)), time scaled to the time of the sequential program (figure 16(b)), or elapsed time (figure 16(c)). Compare machines, programming systems, and algorithms by elapsed time to avoid distortion.

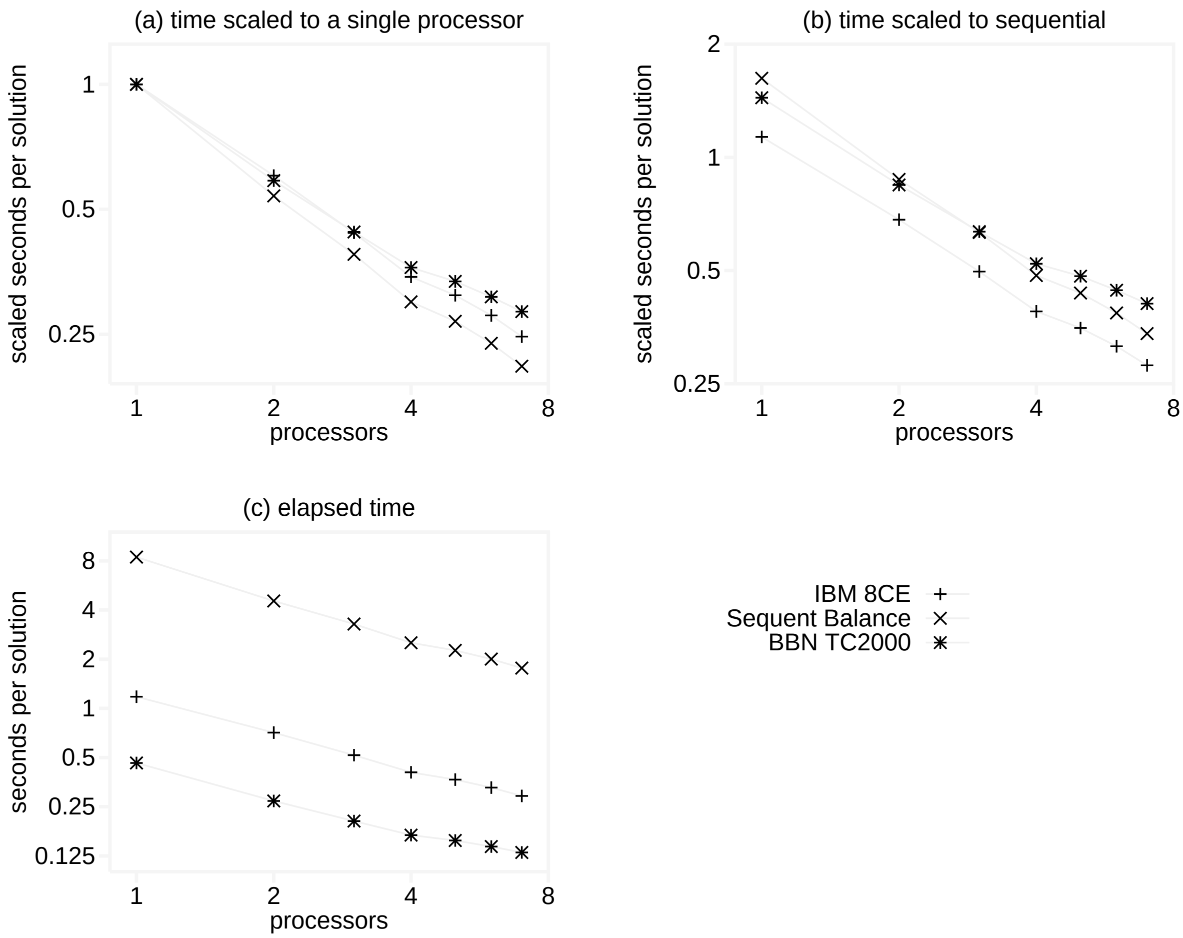

Figure 16: The Effect of Scaling on Performance Graphs.

Writers often compare speedup to compensate for differences in single-processor performance. However, the scale invariance of log-log plots enables us to make such comparisons without computing speedup. If you compare elapsed time on log-log plots, the shapes 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 17 we can see that the TC2000 is always faster than the Symmetry for small numbers of processors, but the TC2000 does not improve so the Symmetry may well perform better at larger numbers of processors. We can also see that the 8CE 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.

6.2 Parallelizations

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

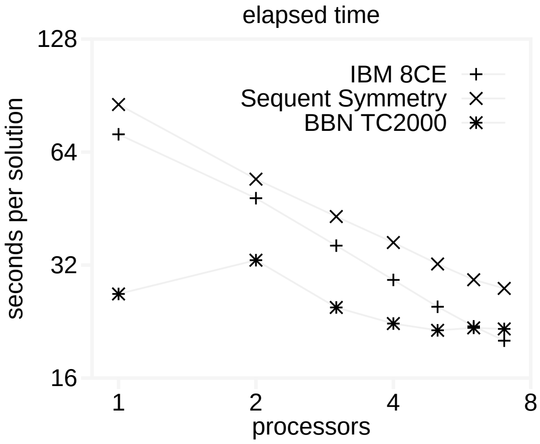

Figure 17: Elapsed Time and Scale Invariance.

processor. The overhead for creating and managing parallelism varies widely, even for a single processor, which could distort the true performance, as in figures 16(a) and 16(b). In this case, figure 16(b) 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.

6.3 Problem Sizes

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, in figure 18(a) 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 in almost the same time as a single processor can complete the smaller problem. This relationship is clearer in figure 18(b), where the lines connect a quadrupled problem size with a quadrupled number of processors, in other words where the lines show problem size scaled with the 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.

6.4 Perfect Performance

Sometimes you may wish to compare observed performance with '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 as a separate line. That is, you show both the actual performance of the parallel program and the perfect speedup of the sequential program. For example, figure 19(a) shows the measured parallel performance with data points and shows the times of the 'perfectly parallel' sequential program with a line.

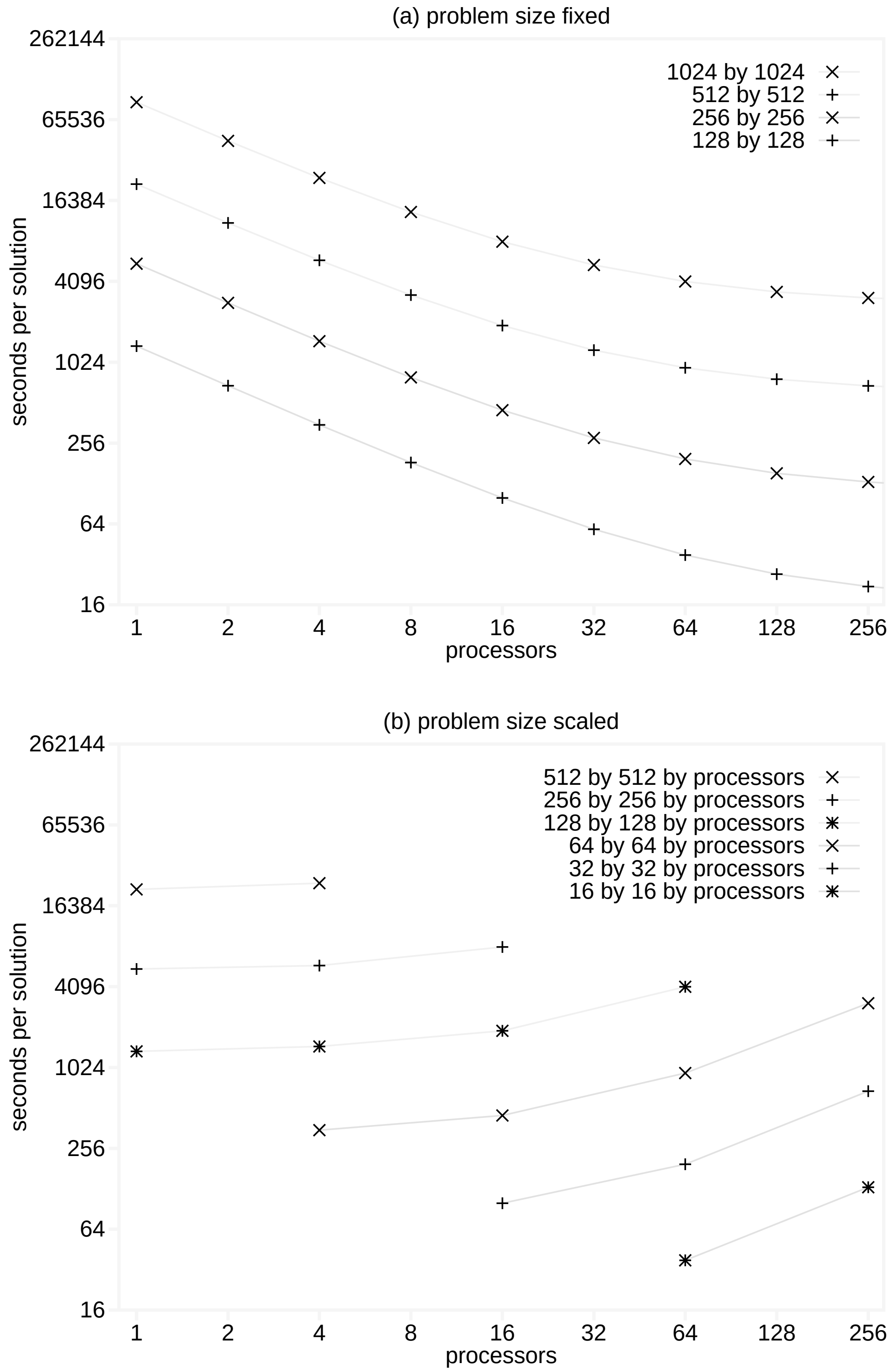

Figure 18: Scaling Problem Size.

The data source is Seetharamakrishnan S, "Strengths and Weaknesses of Dataparallel C", Master's Thesis, Department of Computer Science, Oregon State University, May 1992.

Sometimes a program may parallelize so poorly as to require extending the unextrapolated sequential program time across the graph, as in figure 19(b). This example shows that near 22 processors, the parallel program takes as much time as the sequential program.

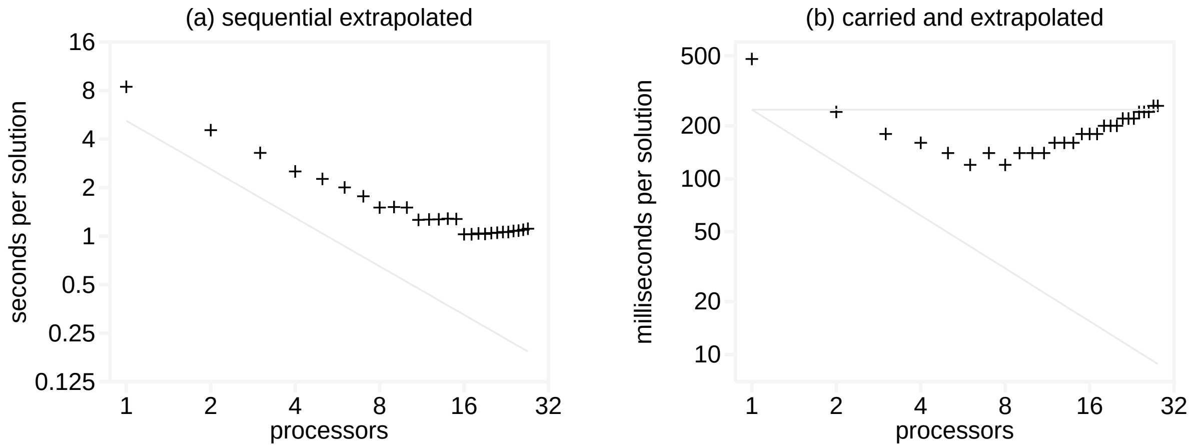

Figure 19: Showing 'Perfect' Performance.

If you are studying different parallel algorithms, you should use the time of the best known sequential program 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. Presenting parallel performance well makes that time more effective. To make your presentation more effective, you should keep in mind three goals:

- Present information, not just data.
- Effectively use space and time.
- Minimize distortion.

In this paper, I have argued that you should:

- Measure elapsed time to account for waiting time in addition to CPU time.
- Measure variance in elapsed time.
- Present elapsed time rather than speedup because speedup is a derivative measure, subject to different and sometimes undocumented derivations. Linear speed is visually similar to speedup, and so may be used in instead of speedup.
- 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 when absolute performance is not important. The scale invariance of log-log plots enables comparison of processors independent of base processor technology.
- Show measure points explicitly on paper.
- Show variability in execution time when it is significant, preferably by plotting all data points.
- Limit the number of independent variables you change when making comparisons, and document those you do change. Document other independent variables needed to reproduce the experiment.
- Scale execution times only when comparing parallelizations, and rely on the scale invariance of log-log plots for all other comparisons.
- Show "perfect" performance as separate lines, rather than scaling the graph.

Following these guidelines should help you present parallel performance more effectively, with less distortion and better use of space and time. On occasion, you may better meet the goal by violating one or more of these guidelines, but making explicit your reasons for violating the guidelines should lead you to more effective presentations of parallel performance.

Acknowledgments The comments of Margaret Burnett, Jesse Chaney, Mark Clement, Phyllis Crandal, Kevin Djang, Cherri Pancake, Rajeev Pandey, and the anonymous reviewers where very helpful in the writing of this paper. Jaswinder Singh provided some of the source data.

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, also published 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.

[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.

[Jain, 1991] Rai Jain, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*, John Wiley & Sons, 1991.

[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.

[Singh *et al.*, 1993] Jaswinder Pal Singh, John L. Hennessy, and Anoop Gupta, "Scaling Parallel Programs for Multiprocessors: Methodology and Examples," *Computer*, 26(7):42–50, July 1993.

[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.