Scaling Parallel Programs for Multiprocessors: Methodology and Examples

Jaswinder Pal Singh, John L. Hennessy and Anoop Gupta

Computer Systems Laboratory
Stanford University
Stanford, CA 94305

In IEEE COMPUTER, vol. 27, no. 7, July 1993.

Abstract

To design effective large-scale multiprocessors, computer architects need to understand how parallel applications will be scaled to use these machines. The methods used to study scaling so far have been based on the computation and communication complexities of individual algorithms, with the data set size typically being the only application parameter that is scaled. The scaling of real applications, however, is considerably more complicated. Other parameters often need to be scaled along with the data set size to reflect how an application scientist would really use additional computing power. We show that system designers must understand these application considerations and use appropriate scaling methods for their benchmark applications, to reach the correct conclusions about the effectiveness and design of larger machines. To demonstrate this, we propose a realistic scaling methodology, apply it to some important scientific applications, and compare the resulting architectural implications with those that would be obtained by scaling only the data set size.

1 Introduction

A primary goal of parallel processing research is to design systems with large numbers of processors that can solve real problems effectively. Since users are likely to run much larger problems on these systems than they are able to run today, it is important for system designers to understand how the applications of interest will scale to use larger machines, and how the scaling of application characteristics will impact the effectiveness and design of future systems.

Two important questions must be answered when scaling an application to evaluate a larger multiprocessor: First, *how much* should the application be scaled; that is, under what constraints (execution time, memory usage, etc.) should the size of the problem be increased to use the larger machine? Second, *how* should the application be scaled; that is, how should the parameters governing the application's execution be changed to satisfy these constraints? The former question has been addressed adequately in the scaling literature. However, the latter, which we believe is critical for the proper design and evaluation of parallel systems, has not. Although several researchers have studied the scaling of parallel algorithms and architectures [9, 17, 8, 5, 6, 13], the work so far has mostly been based on the computation and communication complexity of individual algorithms, without incorporating considerations imposed by the applications that use these algorithms. Since computer scientists think about the complexity of an algorithm in terms of its input data set size, this size is typically the only application parameter whose impact on scaling is considered [10, 8, 5, 6, 13].

The scaling of real applications is more complicated in practice. Particularly in scientific applications that simulate physical phenomena—which are among the most important applications for large-scale multiprocessing—other parameters often have to be scaled together with the data set size to reflect the application scientist's goals in running larger problems. In scientific applications, these other parameters usually control different sources of error that arise from approximations made in the simulation. These parameters (such as the numerical accuracy of an algorithm or the interval between time-steps) can significantly affect the scaling of execution characteristics relevant to the effectiveness and design of larger machines, and it is therefore important to scale the parameters appropriately. We demonstrate this importance, both

analytically and experimentally, by developing a realistic scaling methodology and comparing the architectural implications of some important scientific applications under it and under a simpler methodology that scales only the data set size. Our primary example application is a simulation of galaxies using the Barnes-Hut hierarchical N-body method [3], though we use some other applications to support our arguments.

Section 2 of this paper discusses the question of how much to scale an application, and the development of scaling models (constraints) in the literature. Section 3 addresses how an application should be scaled: A realistic scaling methodology is developed, together with a principle for scaling application parameters that is appropriate for a wide range of scientific applications. Section 4 describes the Barnes-Hut application that is our primary example. In Section 5, we apply our parameter scaling principle to this and other applications, and discuss the interactions of the principle with different scaling models in Section 6. In Section 7, we examine the impact of realistic versus naive scaling on some important architectural implications, including the communication-to-computation ratio that the machine must support and the size of the active working set that a processor needs to maintain in a fast level of the memory hierarchy for effective performance. Finally, Section 8 summarizes our conclusions.

2 How Much to Scale: The Development of Scaling Models

Several models for the constraints under which an application should be scaled have been developed in the literature. The first among these, which we call *constant problem size* (CPS) scaling, assumes that the application parameters will not be changed at all—that the same problem size will be used regardless of the number of processors on the machine. The problem size is specified by the input parameters that the application is run with; for a particular machine, it may be defined as the number of instructions executed in a uniprocessor run, for example.

As pointed out by Amdahl [1], CPS scaling leads to a rapid reduction in parallel efficiency as more processors are used to solve a fixed-size, deterministic problem. Amdahl argued that most parallel programs have some portion of their execution that is inherently serial and must be executed by a single processor while others remain idle. This serial portion may, for example, include initialization, task creation, or some other phase of computation. Amdahl observed that for a fixed problem size, if s is this fraction of the uniprocessor execution time that cannot be parallelized, then the maximum obtainable speedup is $1/s$, even if the rest of the program is executed in infinitesimal time by an infinite number of processors.

The reality of CPS scaling is in fact more pessimistic than predicted by Amdahl's law. There are parts of parallel programs that can use some but not all of the available processors, and parallel execution incurs overheads that increase with the number of processors used. In fact, there comes a point when using more processors to solve a fixed-size problem actually increases the execution time rather than reducing it. Figure 1 demonstrates these effects for two fixed problem sizes in the Barnes-Hut application (measured on the simulated multiprocessor described in the Appendix). At least asymptotically, this decay in performance is not a fault of the architecture, but is dictated by the limited parallelism that the fixed problem size affords. Architectural evaluations using CPS scaling thus put larger machines at an inherent disadvantage.

In the last few years, researchers have pointed out that although Amdahl's law is valid for constant problem size scaling, it is unduly pessimistic for most applications in practice. The reason is that keeping the problem size fixed is probably not how larger parallel machines will actually be used. Application users, particularly in the scientific community, almost always want to run larger problems and will do so when given more powerful machines. Since the serial fraction, s, in most applications does not remain constant but is a diminishing function of the problem size, Amdahl's law does not apply directly when the problem is scaled.

Researchers at Sandia National Laboratories were the first to propose scaling models other than CPS scaling. They obtained very good "scaled speedups" on a 1024 processor machine by scaling problems so that their memory requirements grew linearly with the number of processors [10]. This is known as *memory-constrained* (MC) scaling. It assumes that a user always wants to run the largest data set possible without overflowing the machine's memory.

Memory-constrained scaling can lead to unacceptable increases in execution time, which can make it unrealistic from a user's point of view. Consider, for example, an algorithm that performs $O(n^2)$ computation on $O(n)$ data. If the problem is run with a data set size of n_0 on a single-processor machine, a p-processor machine will run a data set size of $p\,n_0$ under MC scaling. The total number of operations required now increases from ${n_0}^2$ to $(p\,n_0)^2$ which, even with perfect speedup on the p processors, leads to an increase in execution (wall-clock) time by a factor of p.

Another scaling model has therefore been proposed

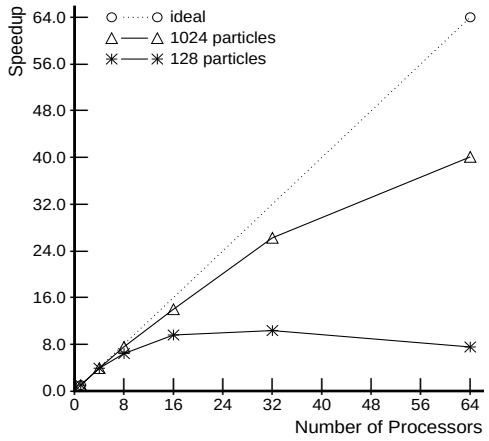


Figure 1: Parallel performance degradation under constant problem size scaling in the Barnes-Hut application.

that constrains the execution time of the parallel program rather than its memory utilization [17]. This model is called *time-constrained* (TC) scaling: The problem is scaled so that the machine takes the same amount of absolute (wall-clock) time to solve it as processors are added. The assumption is that the user always wants to solve the largest possible problem in the fixed amount of time at hand.

Time-constrained scaling is often asymptotically limited as a scaling model, in that the problem size cannot be increased with the number of processors after a certain point under this model. One reason for this is that the concurrency in an application often scales at most linearly with the data set size. When the data set size grows less than linearly with the number of processors (as in the example above) there will come a point when the number of processors exceeds the concurrency, and larger problems can no longer be run within the fixed execution time regardless of the number of processors used. The overheads of parallel execution further inhibit the growth of problem size under TC scaling, and can in fact limit TC scaling even when the available concurrency grows linearly with the number of processors. Despite these asymptotic limitations, however, TC scaling is often the most realistic scaling model in practice.

Finally, there may of course be situations in which a user does not want to run a larger problem, but will use a larger machine to simply solve the same problem faster, despite the limitations of constant problem size scaling. No one of these scaling models (CPS, TC, MC) can be claimed to be the most realistic for all applications and all users. Different users have different goals, work under different constraints, and are in any case unlikely to follow a given model very strictly. However, these three models constitute useful tools for an analysis of scaling, and we use them here.

3 How to Scale: A Methodology

Scaling studies performed by computer scientists usually scale only the input data set size of an application to meet the constraints of one of the above models. Based on our experience with a variety of scientific applications, we believe that the scaling of applications is more complicated in practice. Other application parameters often have to be scaled together with the data set size to reflect how an application scientist would run larger problems to obtain useful results. These parameters have their own impact on important execution characteristics of the application, such as its computation, communication and memory requirements. Any study of scaling must therefore be based on an understanding of the relationships among application parameters, if it is to reach the correct conclusions about the effectiveness and design of larger machines. This need to scale other parameters is implicitly addressed by Worley in one of the differential equation solvers he examines (the hyperbolic equation example in [17]). However, his reason for scaling the other parameter that he scales is the stability of the algorithm itself, and even his study ignores considerations imposed by real applications that might use the algorithm.

Scientific applications often simulate physical phenomena that occur in nature. They do this by discretizing a continuous physical problem, in both space and time, and solving the discretized problem through numerical approximation algorithms. Consequently, there are several application parameters that are sources of simulation error: some that determine the resolutions of the problem discretizations, and others that control the accuracy of the

numerical algorithms used. The goal of an application scientist in running larger problems—those that require more computation and perhaps memory—is usually to reduce the overall error in the simulation and have it more accurately reflect the phenomenon being simulated.[1] For example, the reason to increase the data set size of a simulation is usually to refine the spatial discretization, and hence reduce that component of the overall simulation error. However, this is usually just one source of error, and reducing it without reducing the error contributions of other sources is often not the most effective way to reduce the overall simulation error: Parameters governing the errors arising from other sources should be scaled simultaneously.

Unfortunately, even in scientific applications, there are no universal rules for how scientists will scale different approximations in their simulations relative to one another. Errors often do not combine in an analytically clean manner, which makes changes in the overall simulation error difficult to even predict. However, the notion of different identifiable sources of simulation error gives rise to a principle that has both intuitive appeal and widespread practical applicability in many physical domains [2]:

- *All sources of error should be scaled so that their error contributions are about equal.*

This is the scaling principle we use in our study.

Clearly, this scaling principle does not apply to all classes of applications. For example, in many non-numerical applications—such as database or search problems, for instance—the notion of simulation error is either not applicable or not quantifiable. Even in scientific applications, there may be parameters that are important to scale but that either do not affect error or affect it in unknown ways. And finally, even when an error-based principle is appropriate, there may be other error-based principles that are more appropriate for a particular application than the one we use here. For example, if errors due to different sources combine in predictable ways, it may make sense to scale so that the overall simulation error is minimized subject to a constraint on execution time or memory usage, or so that time or memory usage is minimized subject to a bound on error. The truth is that there is probably no specific parameter scaling principle that can be universally applied. However, the central point of this paper—that it is important to understand how to scale all relevant parameters in an application according to the dictates of the application domain, since this can significantly affect the scaling of execution characteristics relevant to a system designer—applies universally. And the particular, equal-error, scaling principle we use to demonstrate this point is widely applicable to scientific applications that model physical phenomena.

Our methodology for studying scaling, then, is as follows. We first understand the relationships between application parameters in terms of their error contributions. Then, we use our equal-error principle to develop a rule for scaling the parameters relative to one another. This tells us how the different parameters should be scaled to use larger multiprocessors under different scaling models or constraints. We assume that a multiprocessor is scaled by adding identical processors to it, each added processor having the same amount of main memory as the others in the ensemble. Finally, we examine how various execution characteristics of interest scale with the number of processors under the appropriate rule and models.

Let us now introduce the application that we use as our primary example in this paper, and apply our scaling methodology to it.

4 The Example Application

The application simulates the evolution of galaxies under the influence of gravitational forces. It is a classical N-body simulation, and uses the Barnes-Hut hierarchical N-body algorithm to compute the interactions among n bodies in $O(n \log n)$ sequential time. Hierarchical methods, of which the Barnes-Hut algorithm is an important example, are becoming increasingly popular in scientific computing. Besides being highly parallel, these methods substantially reduce computational complexity by exploiting a property that is fundamental to many of the physical phenomena that scientific applications simulate. This property is the large range of scales in spatial and temporal information requirements. That is, a point in the physical domain requires progressively less information progressively less frequently from parts of the domain that are progressively further away from it. In N-body problems, for example, the magnitude of the interaction between bodies typically falls off with the distance between them. This property leads to the following insight, first observed by Isaac Newton in the year 1687: The effect of a large group of bodies may be approximated by that of a single equivalent body, if the group of bodies is far enough away from the point at which the effect is being evaluated. Let us see how the Barnes-Hut algorithm exploits this insight hierarchically.

[1] There may, of course, be other reasons for running larger problems, such as simulating new aspects of a phenomenon or exploring new conjectures without too much concern for accuracy; however, improving accuracy is a dominant motivation in many domains that use numerical approximation techniques, including the ones we consider here.

4.1 Sequential Algorithm

The three-dimensional Barnes-Hut algorithm is based on a hierarchical octree (at most eight children per node) representation of space. The root of this tree represents a space cell containing all bodies in the system. Internal nodes represent recursively subdivided space cells, and the subdivision proceeds until a leaf cell contains only a single body. The tree therefore extends to more levels in regions that have high particle densities. Although we use a three-dimensional problem in this paper, Figure 2 shows a small two-dimensional example domain and the corresponding quadtree (at most four children per node).

Since the application simulates an evolving system, it proceeds over a large number of time-steps. The sequential Barnes-Hut algorithm divides every time-step into four phases: computing the dimensions of the root cell and building the tree, computing the cell centers of mass by an upward pass through the tree, computing forces, and updating the properties of the bodies. A fifth, partitioning, phase is added in our parallel implementation. Let us look at the force computation phase in more detail, since it is where most of the execution time is spent.

The tree is traversed once per body to compute the net force acting on that body. The force-calculation algorithm for a body starts at the root of the tree and conducts the following test recursively for every cell it visits: If the center of mass of the cell is far enough away from the body, the entire subtree under that cell is approximated by a single particle at the center of mass of the cell, and the force this center of mass exerts on the body computed; if, however, the center of mass is not far enough away, the cell must be "opened" and each of its subcells visited. A cell is determined to be far enough away if the following condition is satisfied:

$$\frac{l}{d} < \theta \qquad (1)$$

where l is the length of a side of the cell, d is the distance of the body from the center of mass of the cell, and θ is a user-defined accuracy parameter (θ is usually between 0.3 and 1.2). In this way, a body traverses more levels of those parts of the tree which represent space that is physically close to it, and groups other bodies at a hierarchy of length scales.

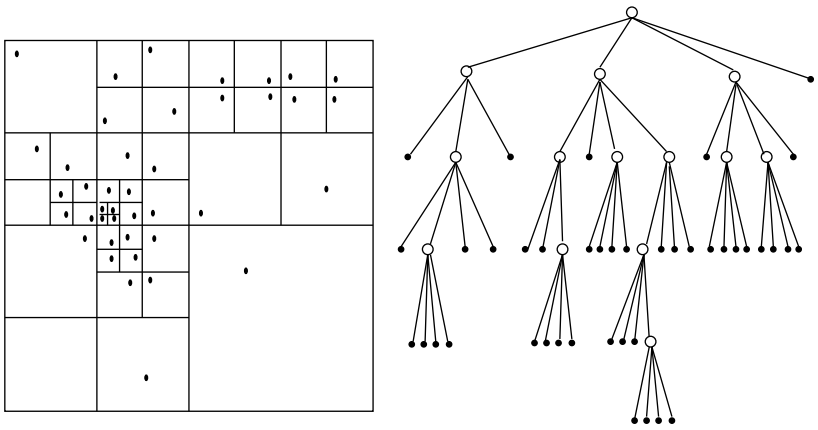
4.2 Parallel Implementation

Our experiments are conducted with a shared-memory parallel program, which executes each of the phases within a time-step in parallel. The partitioning technique we use yields a balanced workload across processors and provides data locality by ensuring that particles assigned to a processor are close together in space [16]. To demonstrate that the program achieves successful parallel performance, Figure 3 shows the speedups obtained under constant problem size scaling on a state-of-the-art experimental multiprocessor, the Stanford DASH multiprocessor, as well as on a larger simulated multiprocessor described in Section 7.1. Five time-steps are measured in these and all results we present. The initial distribution is a pair of highly nonuniform galaxies that interact with each other. Let us now apply our scaling methodology to this application, and examine how the resulting architectural implications differ from those of scaling only the data set size.

5 Development of a Scaling Rule

Our scaling principle for scientific applications requires an understanding of how application parameters contribute to errors in the simulation (see Section 3). Discrete N-body simulations, such as the Barnes-Hut application, have several sources of error in their approximation of a continuous physical process, each controlled by a different application parameter. For the collisionless gravitational problems that we consider here, these sources of error include the following: (i) Monte-Carlo sampling of phase-space to approximate the system by a finite number of bodies (spatial discretization), parametrized by the number of bodies n; (ii) the use of a discrete, non-zero time-step in integrating the equations of motion (temporal discretization), parametrized by the time-step duration Δt; (iii) approximations made in force-computation within a time-step, parametrized by the Barnes-Hut accuracy parameter θ; (iv) force-softening to render the force-computation expression $F = \frac{G\, m_1\, m_2}{r^2 + \epsilon^2}$ non-singular, parametrized by ϵ; and (v) roundoff due to finite word length in the computer. The last source is a static machine characteristic, and we ignore it. Studies have been done in the astrophysics community to investigate the impacts of the other four parameters on simulation accuracy [4, 11]. While some of the error contributions are not completely independent, the following rules emerge as being valid in interesting parameter ranges.

- n: The error from the increased relaxation rate due to Monte Carlo sampling scales as $\frac{1}{\sqrt{n}}$; thus, an increase in n by a factor of s leads to a decrease in simulation error by a factor of $\sqrt{s}$.
- Δt: The leap-frog method used to integrate the particle orbits has a global error of the order of $(\Delta t)^2$. Thus, reducing the error by a factor of $\sqrt{s}$ (to match that due to a s-fold increase in n) requires a decrease in Δt by a factor of $\sqrt[4]{s}$, and hence that many more time-steps to simulate a fixed amount of physical time, which is usually held constant.



(a) The Spatial Domain (b) Quadtree Representation

Figure 2: A two-dimensional particle distribution and the corresponding quadtree.

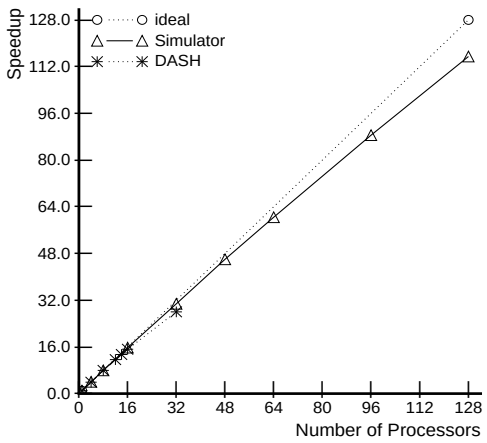


Figure 3: Speedups on the Simulator and the DASH multiprocessor (16K particles, θ = 0.7).

- θ: The results in [4, 11] demonstrate a scaling of the force-calculation error proportional to θ^2 in the range of practical interest (this is for the original Barnes-Hut algorithm: If higher order moments are incorporated in the force calculation, the error scales more quickly with θ). Reducing the error by a factor of $\sqrt{s}$ thus requires a decrease in θ by a factor of $\sqrt[4]{s}$.
- ϵ: The error due to softening can be ignored if the value of ϵ is smaller than the average inter-particle spacing, as it usually is.

The rule for scaling application parameters that results is: If the number of bodies n is scaled by a factor of s, the Δt and θ must each be scaled by a factor of $1/\sqrt[4]{s}$.

Most scientific applications similarly have several parameters that should be scaled simultaneously. We shall present some other examples in the next section. But first, let us see how the scaling rule for the Barnes-Hut application interacts with different scaling models.

6 Realistic versus Naive Scaling: Implications for Scaling Models

We first examine how the memory requirements and serial computational complexity of the application depend on the different application parameters. Then, we see how the rule for simultaneously scaling these parameters (which we call *realistic* scaling) impacts the TC and M-C models differently than scaling only the data set size (which we call *naive* scaling).

6.1 Scaling of Memory Requirements and Computational Complexity

Memory requirements Of the three parameters we scale (n, θ and Δt), only the number of bodies, n, has an impact on the memory requirements of the application in a single address space. The main sources of memory requirement are the data for particles and tree cells. The former require memory proportional to n. The latter depend on the spatial distribution as well as on n, but are also found to be proportional to n for distributions of interest [11]. Thus, there is no difference between realistic and naive scaling for memory requirements in a single address space (multiprocessors that don't provide a shared address space may require substantial replication of memory in private address spaces, the amount of this memory being likely to depend on other parameters than n as well [15]).

Computational Complexity The serial time complexity of the application depends on n in an $O(n\ log\ n)$ fashion for realistic ranges of θ. The dependence on θ is found to be roughly proportional to $\frac{1}{\theta^2}$ for fixed n [11]. Finally, the complexity is roughly proportional to the number of time-steps; that is, inversely proportional to Δt when a fixed amount of physical time is being simulated, which is usually the case. Since the complexity scales quite independently with these three parameters, an increase in input data set size (n) by a factor of s leads to a serial time complexity increase of a factor of

$$s \ * \ \left(1 + \frac{log\ s}{log\ n}\right) \ * \ \sqrt[4]{s} \ * \ \sqrt{s}$$

under our scaling rule. The first two factors in the above expression are due to an increase in n by a factor of s, the third due to an increase in the number of time-steps by a factor of $\sqrt[4]{s}$, and the last due to a decrease in θ by a factor of $\sqrt[4]{s}$ (given the above dependence of time complexity on different parameters). Thus, the last two factors would not be present if only n were scaled. Let us see how these effects of realistic and naive scaling interact with different scaling models.

6.2 Impact on Memory-constrained and Time-constrained Scaling

The issue of scaling problem parameters is irrelevant to constant problem-size scaling. We therefore examine how the effects of scaling parameters interact with the memory- and time-constrained scaling models, given the impact of different parameters on memory requirements and computational complexity discussed above. Specifically, we examine how the parallel execution time grows under memory-constrained scaling, and how the data set size (or number of particles that can be simulated) grows under time-constrained scaling.

Execution Time under Memory-constrained Scaling: Under memory-constrained scaling, an increase in the number of processors and hence memory by a factor of k allows us to run a problem with k times as many bodies (assuming a shared address space and no need for data replication in main memory, which is the case in this application [15]). Increasing the number of processors does not affect the shared data requirements, and the amount of per-processor private memory used is negligible in comparison. Under naive scalability analysis, the $O(n\ log\ n)$ complexity of the algorithm suggests that the parallel execution time, assuming perfect speedup, would increase slightly—by a factor of $\frac{kn_0\ log\ (kn_0)}{k * n_0\ log\ n_0}$ or $\left(1 + \frac{log\ k}{log\ n_0}\right)$, where n_0 is the number of bodies run on the smaller machine. Under our more realistic rule for scaling parameters, however, an additional factor of $k^{\frac{3}{4}}$ applies (see Section 6.1), even if perfect speedups are assumed. This additional factor is much larger than that already yielded by the naive analysis, and in fact constitutes an unacceptable increase in execution time under memory-constrained scaling (see Figure 4(a)).

For example, suppose that a scientist runs a problem with 16K particles that takes one day to complete on a uniprocessor. Given a machine with 1024 such processors and 1024 times as much memory, using MC scaling in a shared address space would require running a problem with 16K * 1024 or 16 million particles. Naive scalability analysis would predict that the application would now take 1.7 days, which is probably quite reasonable from the scientist's point of view. In fact, however, the scaling of other parameters under the equal error scaling principle dictates that a simulation with 16 million particles would take more than 309 days, even if perfect speedups are achieved. Realistic scaling turns the problem from perhaps being memory bound to being entirely computation-time bound. *Time-constrained scaling is therefore likely to be more important than memory-constrained scaling in practice*.

Data Set Size under Time-constrained Scaling: If only the number of particles, n, were scaled under time-constrained scaling, the $O(n\ log\ n)$ complexity of the algorithm suggests that n would scale almost linearly with the number of processors, i.e. almost as fast as under memory-constrained scaling. The scaling of other parameters, however, causes the number of particles that can be simulated in a fixed amount of time to grow much more slowly (Figure 4(b)). Under our equal-error scaling rule, for example, the growth in the number of particles is closer to a factor of $\sqrt{k}$ (even assuming perfect speedup, which is not achievable under TC scaling), where k is the

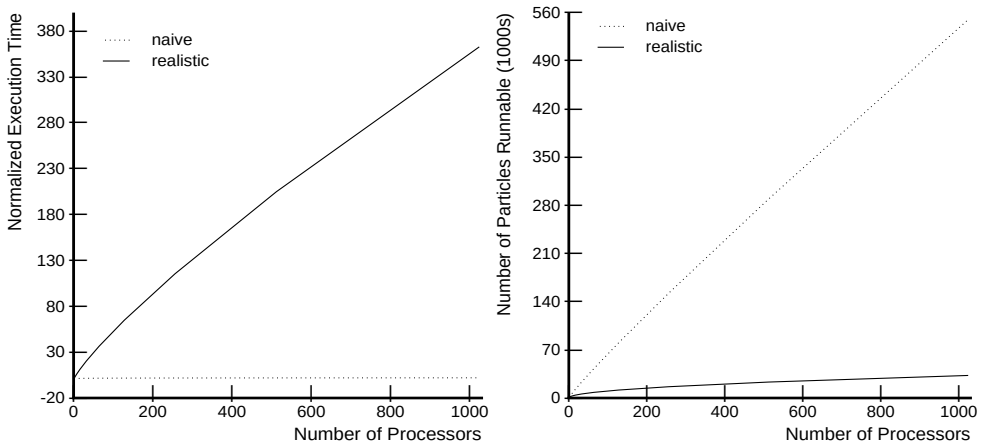


Figure 4: Impact of Realistic versus Naive Scaling.

factor by which the number of processors is increased. More precisely, the factor s by which the number of particles can grow is related to k by the following expression:

$$k \;=\; s^{\frac{7}{4}} \left(1 + \frac{log\ s}{log\ n_0}\right),$$

where n_0 is the number of particles run on the smaller machine. Thus, if a problem with 16K particles takes one day to complete on a uniprocessor, then a machine with 1024 processors would run a problem with only about 700K particles in the same time, rather than the almost 10 million particles that scaling only the data set size would predict.

The realistic scaling of parameters has already led us to two different results for the Barnes-Hut application than scaling only the data set size: First, that memory-constrained scaling is unrealistic; and second, that the number of particles that can be simulated in a fixed amount of time grows much more slowly with computational power than the $O(n\ log\ n)$ complexity in terms of data set size might suggest. Both these results arise from two general characteristics of scientific applications: (i) the need to reduce the error contributions of parameters other than the data set size, and (ii) the fact that reducing the error contribution of a parameter causes an increase in execution time. Not surprisingly, therefore, similar results hold for many scientific applications that we have examined. Some examples are summarized in Table 1. The Fast Multipole application uses the Fast Multipole Method [7]—a recent and very promising algorithm for solving N-body problems—in a galactic simulation. The ocean application [14] simulates eddy currents in an ocean basin, using a multigrid algorithm to solve elliptic partial differential equations at every time-step. And the Wave Propagation application, used in the Sandia experiments [10], simulates the passage of a two-dimensional wave through a set of acoustic deflectors.

We now show that realistic and naive scaling methods can also lead to different conclusions about the scaling of execution characteristics that are of interest to a multiprocessor architect.

7 Realistic versus Naive Scaling: Architectural Implications

We first focus on an execution characteristic that has implications for the effectiveness of larger machines in solving larger problems: the communication-to-computation ratio that a machine must support. Then, we briefly present results for some important architectural design parameters, including the working set size, the average number of sharers of a datum, the synchronization frequency, and the input/output frequency and volume.

7.1 Communication to Computation Ratio

The inherent communication-to-computation ratio in a parallel application is one of the important determinants of its performance on any architecture. The higher the ratio, the less likely is a machine to provide effective performance on that application.

The communication-to-computation ratio usually increases quickly with the number of processors under CPS scaling. It is, however, tempting to believe that the ratio can be kept constant on larger machines by simply running "proportionally scaled" problems. For example, in the programs used in the Sandia experiments [10], for example, the primary dependence of both communication-and-computation on the data set size (n) and the number of processors (p) could be expressed as a simple function of $\frac{n}{p}$. The memory-constrained scaling used in those ex-

Table 1: Relative Scaling of Application Parameters, and their Effects on Execution Time.

Application	Parameters	Contribution to Exec. Time	Relative Scaling Factors (Equal-error Scaling)	Contribution to Exec. Time under Equal-error Scaling	Scaling of Parallel Exec. Time with s
Barnes-Hut	No. of particles (n)	$O(n\ log\ n)$	s	$s\ *\ \left(1+\frac{log\ s}{log\ n}\right)$	$s^{0.75}\ *\ \left(1+\frac{log\ s}{log\ n}\right)$
	Force-calc accuracy (θ)	$O(\frac{1}{\theta^2})$	$\frac{1}{\sqrt[4]{s}}$	$\sqrt{s}$	
	Time-step (Δt)	$O(\frac{1}{\Delta t})$	$\frac{1}{\sqrt[4]{s}}$	$\sqrt[4]{s}$	
Fast Multipole	No. of particles (n)	$O(n)$	s	s	$s^{0.25}\ *\ log^2\ \sqrt{s}$
	Force-calc accuracy (ϵ)	$O(log^2\ \epsilon)$	$\frac{1}{\sqrt{s}}$	$log\ \sqrt{s}$	
	Time-step (Δt)	$O(\frac{1}{\Delta t})$	$\frac{1}{\sqrt[4]{s}}$	$\sqrt[4]{s}$	
Ocean Simulation	Grid points (n)	$O(n\ *\ \alpha(\sqrt{n}))^a$	s	$s\ *\ \alpha(\sqrt{s})$	$s\ *\ \alpha(s)\ *\ \beta(s)$
	Multigrid Convergence Tol. (ϵ)	$O(\beta(\frac{1}{\epsilon}))$	$1/s$	$\beta(s)$	
	Time-step (Δt)	$O(\frac{1}{\Delta t})$	$1/s$	s	
Wave Propagation	Grid points (n)	$O(n)$	s	s	$\sqrt{s}$
	Time-step (Δt)	$O(\frac{1}{\Delta t})$	$\frac{1}{\sqrt{s}}$	$\sqrt{s}$	

[a] α, β are functions that describe the number of iterations taken by the multigrid equation solver to converge for a given number of grid points, as a function of the grid resolution ($1/n$) and convergence accuracy (ϵ). Typically, they are slowly growing functions of n, $\frac{1}{\epsilon}$.

periments increased n proportionally with p, and therefore kept the communication-to-computation ratio nearly constant. Memory-constrained scaling, however, is inappropriate even for the application used in those experiments. In fact, how the communication-to-computation ratio scales for a particular application depends on the scaling method used, as our results in this section show.

Figure 5 shows how the inherent communication-to-computation ratio in the Barnes-Hut application scales under time- and memory-constrained scaling. The experimental environment and methodology used to obtain these results is described in the Appendix. "Naive" scaling in the figure refers to scaling only the number of particles (data set size) and not any other parameters. The base problem size on a single processor is the same in all cases. We can see that the trends for naive versus realistic TC (or MC) scaling are different, as are the trends for realistic TC versus realistic MC scaling. These results underscore the importance of using an appropriate scaling method to arrive at the correct conclusions.

Modeling the communication-to-computation ratio analytically is difficult in this application, particularly for a nonuniform domain. To understand the observed trends for different scaling methods, let us first intuitively understand how the ratio depends on different parameters. The parameters that we use in our discussion are the granularity of the partitions (parametrized by the average number of particles, n_g, assigned to a processor in a time-step), the spatial extent of the partitions (parametrized by $\frac{1}{p}$, the reciprocal of the number of processors), the force-calculation accuracy parameter θ, and the time-step interval Δt.

As the granularity, n_g, increases for fixed p, θ and Δt, the amount of computation per processor scales as $O(n_g\ log\ n)$. Communication scales more slowly, however, primarily because most of the communicated data are heavily reused by other particles in the partition. The communication-to-computation ratio also improves with increased force-calculation accuracy (decreased θ) for the same n, p and Δt, for the same reasons of data reuse. And for the same n_g, θ, and Δt, the amount of communication increases with p (i.e. under naive MC scaling), both because the $log\ n$ in $O(n\ log\ n)$ increases and because more of the data that a processor references are not in its (now spatially smaller) partition. Computation in this case increases as only $O(log\ n)$, and the result is a slow increase in the communication-to-computation ratio. Finally, as Δt is refined, the spatial distribution changes more slowly across time-steps, which should also reduce communication. These observations can be used to explain the trends for different scaling methods in Figure 5 as follows.

We have already seen why there is a slow increase in the communication-to-computation ratio under naive MC scaling. To prevent the ratio from increasing in fact takes realistic MC scaling, in which case θ and Δt decrease and the ratio actually falls a little with scaling. However, we have seen that MC scaling is a highly unlikely method for this application in practice.

The results of greatest practical interest are those for TC scaling, in which case the trends for naive and realistic scaling are also quite different. Under naive TC scaling, the $O(n\ log\ n)$ complexity of the algorithm implies that n scales only a little slower than under naive MC scaling, so that n_g decreases very slowly; θ and Δt remain constant, and the ratio in this case increases very slowly as well. Realistic TC scaling causes n to scale

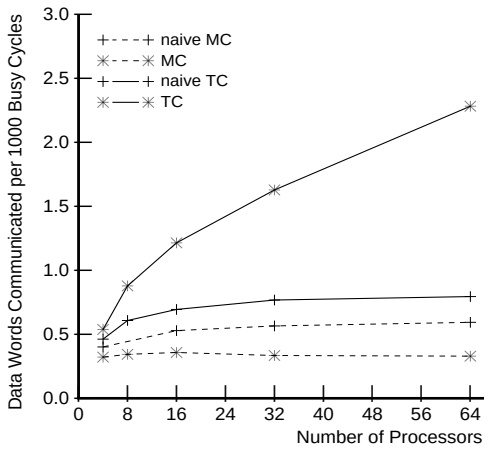


Figure 5: Communication-to-computation Ratio under TC and MC scaling.

much more slowly (see Section 6), and n_g to therefore decrease quite quickly as the number of processors increases. The communication-to-computation ratio therefore increases much faster than under naive TC scaling, although the rate of increase is inhibited a little by the scaling of θ and Δt. Such differences in trends for realistic and naive scaling methods are also observed in other applications we have studied, highlighting the need to scale applications correctly. Owing primarily to reductions in granularity, the communication-to-computation ratios are found to grow under realistic TC scaling in those applications as well.

7.2 Other Architectural Issues

Simplistic scaling methods can lead to incorrect conclusions about the scaling of architectural design parameters as well. Some examples follow.

Working Set Size: For example, one important design parameter is the size of a processor's active working set. This working set size indicates how much cache (or local memory, if local memory is used as a software-controlled cache) is needed per processor to obtain effective performance. In N-body applications, for example, we find that caching communicated data is critical to obtaining good performance, and that the working set size is very sharply defined. Performance is very good if caches are larger than the working set size, and substantially degraded if they are not. The working set size in these applications is independent of the number of processors used, and depends only on application parameters.

In the Barnes-Hut case, the working set size turns out to be roughly proportional to $\frac{1}{\theta^2} log\, n$. Under our equal error scaling rule, if n scales by a factor of s, then the accuracy parameter θ should scale by a factor of $\frac{1}{\sqrt[4]{s}}$. Thus, the contributions of n and θ to the scaling of the working set size are:

- n: an *additive* term of $O(log\ s)$
- θ: a *multiplicative* term of $O(\sqrt{s})$.

The contribution of scaling the accuracy parameter is clearly much more significant than that of scaling only the data set size. This would be true under most realistic scaling principles, and reinforces the importance of scaling under application considerations: The naive method of scaling only the data set size leads to incorrect conclusions about how the working set size—and hence the ideal cache or local memory size per processor—scales on larger machines. The Fast Multipole application makes this point more dramatically. There, the size of the active working set is largely independent of both the number of processors as well as the data set size, and therefore remains constant under the naive scaling models. However, it grows proportionally to the logarithm of the force calculation accuracy, and thus grows (albeit slowly) under realistic TC and MC scaling.

Average Number of Sharers: As another example of the importance of realistic scaling, architects care about the number of processing nodes that hold copies of a data object when that object is modified. This number has implications for the amount of message traffic generated in a machine, as well as for the design of directory-based cache-coherence protocols in shared-address-space multiprocessors. In the Barnes-Hut application, for instance, this number also scales more rapidly due to the change in the accuracy parameter θ than due to the change in n under the equal error scaling rule.

Synchronization Frequency: The fact that more time-steps are simulated in a given amount of execution time under realistic TC scaling (since the time-step parameter is scaled down) implies that the granularity of computation between synchronization points decreases and the relative overhead of synchronization therefore increases.

Input/Output Frequency and Volume: Finally, the amount of data output by the program at every output point decreases under realistic TC scaling, since this amount of data is proportional to the number of particles in the simulation. The number of output points in the program execution per unit time may increase, however, depending on whether output is written once every fixed number of time-steps or once every fixed number of simulated seconds.

All the above effects, which are found in many other scientific applications as well, are not seen if only the data set size is scaled.

8 Concluding Remarks

Studies of scaling in the literature have been successful in defining useful models for the constraints under which parallel applications might be scaled, but quite lacking in their treatment of how applications might actually be scaled to meet these constraints. The key reason for this is that scaling has been examined from the viewpoint of individual algorithms, without considering the context of complete applications that use these algorithms in practice. Since computer scientists think of the complexity of algorithms in terms of the data set size that they operate on, this data set size is typically the only parameter that is scaled, both in scaling analyses as well as in experimental studies that have recently begun to use real parallel applications as benchmarks.

In this paper, we have argued that the scaling of real applications is actually more complicated. In the large and extremely important class of scientific applications that simulate physical phenomena, for example, a scientist will most often want to scale other application parameters in conjunction with the data set size in order to obtain useful results. Realistic scaling methods for these applications are usually based on the contributions that different application parameters make to the error in the simulation. One widely applicable principle is to scale parameters so that their error contributions are about equal. This is the principle we have used in this paper. Through some representative examples, we have shown that scaling in realistic ways can lead to different conclusions about the effectiveness and design of larger multiprocessors than naively scaling only the data set size. We therefore conclude that system designers must make the effort to understand the relevant characteristics of the benchmark applications they use, if they are to arrive at the correct design decisions for building larger multiprocessor systems.

Acknowledgements

We thank Joshua Barnes and Leslie Greengard for providing us with the sequential Barnes-Hut and FMM programs, for their patience in many electronic mail discussions about various aspects of scaling the applications, and for pointers to the relevant literature. This work was supported by DARPA under Contract No. N00039-91-C-0138. Anoop Gupta is also supported by a Presidential Young Investigator Award, with matching grants from Ford, Sumitomo, Tandem and TRW.

References

[1] G.M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings*, pages 483–485, April 1967.

[2] Joshua E. Barnes. Personal communication. May 1991.

[3] Joshua E. Barnes and Piet Hut. A hierarchical O(N log N) force calculation algorithm. *Nature*, 324(4):446–449, 1986.

[4] Joshua E. Barnes and Piet Hut. Error analysis of a tree code. *Astrophysics Journal Supplement*, 70:389–417, 1989.

[5] Graham Carey, Joe Schmidt, Vineet Singh, and Dennis Yelton. A scalable, object-oriented finite element solver for partial differential equations on multicomputers. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, 1992.

[6] Ian Foster, William Gropp, and Rick Stevens. The parallel scalability of the spectral transform method. (MCS-P215-0291), September 1991.

[7] Leslie Greengard and Vladimir Rokhlin. A fast algorithm for particle simulation. *Journal of Computational Physics*, 73(325), 1987.

[8] Anshul Gupta and Vipin Kumar. On the scalability of FFT on parallel computers. In *Proceedings of the Frontiers 90 Conference on Massively Parallel Computation*, October 1990.

[9] John L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[10] John L. Gustafson, Gary R. Montry, and Robert E. Brenner. Development of parallel methods for a 1024-processor hypercube. *SIAM Journal of Scientific and Statistical Computing*, 9(4):532–533, July 1988.

[11] Lars Hernquist. Performance characteristics of tree codes. *Astrophysics Journal Supplement*, 64:715–734, 1987.

[12] Dan Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148–159, May 1990.

[13] Daniel Nussbaum and Anant Agarwal. Scalability of parallel machines. *Communications of the ACM*, 34(3):57–61, 1991.

[14] Jaswinder Pal Singh and John L. Hennessy. Finding and exploiting parallelism in an ocean simulation program: Experiences, results, implications. *Journal of Parallel and Distributed Computing*, 15(1):27–48, May 1992.

[15] Jaswinder Pal Singh, John L. Hennessy, and Anoop Gupta. Implications of hierarchical N-body techniques for multiprocessor architecture. Technical Report CSL-TR-92-506, Stanford University, 1992.

[16] Jaswinder Pal Singh, Chris Holt, Takashi Totsuka, Anoop Gupta, and John L. Hennessy. Load balancing and data locality in hierarchical N-body methods. *Journal of Parallel and Distributed Computing*. To appear. Preliminary version available as Stanford Univeristy Tech. Report no. CSL-TR-92-505, January 1992.

[17] Patrick H. Worley. The effects of time constraints on scaled speedup. *SIAM Journal of Scientific and Statistical Computing*, 11(5):838–858, September 1990.

Appendix: Experimental Environment

This appendix describes the experimental environment and methodology used to measure communication-to-computation ratios in the applications.

We are interested in the communication-to-computation behavior that is inherent to the parallel program, not that which is an artifact of how the program's data requirements or referencing patterns interact with the particular memory system organization (finite cache size, large cache lines or page sizes, etc.). We therefore use a simulated multiprocessor architecture with infinite per-processor caches and a small cache line size (8 bytes). To eliminate cold-start misses, which would be negligible in realistic runs that last many hundreds of time-steps, we do not measure the first two time-steps of the computation. Our cache coherence protocol uses invalidations rather than updates to maintain coherence on shared data that are modified, and is similar to the protocol used in the Stanford DASH multiprocessor [12].

The multiprocessor simulator we use is composed of two parts: the Tango event-driven reference generator, and a memory system simulator. Every processor has its own cache (in this case infinite), and no contention is modeled anywhere in the machine. Latencies for memory references are as follows: cache hits cost a single cycle, read misses that are satisfied in the local memory unit stall the processor for 15 cycles, and read misses satisfied in some other memory unit for 60 cycles. Since write miss latencies can be hidden very effectively by hardware techniques, so that they essentially do not stall the processor in applications that don't have much fine-grained synchronization, write misses are assumed to cost only a single cycle.

We measure the communication-to-computation ratio as the average number of read misses per 1000 busy cycles per processor. In the multiprocessor model described above, read misses represent the fundamental communication required to solve the problem (in fact, they represent the minimum amount of inherent communication required, regardless of whether a shared-address-space or message-passing machine is used). Write references can also generate communication when a real cache coherence protocol is used, but the trend for communication-to-computation ratios is independent of whether they are included in the measurements or not.