Isoefficiency: Measuring the Scalability of Parallel Algorithms and Architectures

Ananth Y. Grama, Anshul Gupta, and Vipin Kumar
University of Minnesota

Isoefficiency analysis helps us determine the best algorithm/architecture combination for a particular problem without explicitly analyzing all possible combinations under all possible conditions.

An earlier version of this article appeared as "Analyzing Performance of Large-Scale Parallel Systems" by Anshul Gupta and Vipin Kumar on pp. 144-153 of the *Proceedings of the 26th Hawaii International Conference on System Sciences*, published in 1993 by IEEE Computer Society Press, Los Alamitos, Calif.

The fastest sequential algorithm for a given problem is the best sequential algorithm. But determining the best parallel algorithm is considerably more complicated. A parallel algorithm that solves a problem well using a fixed number of processors on a particular architecture may perform poorly if either of these parameters changes. Analyzing the performance of a given parallel algorithm/architecture calls for a comprehensive method that accounts for scalability: the system's ability to increase speedup as the number of processors increases.

The isoefficiency function is one of many parallel performance metrics that measure scalability.[1-6] It relates problem size to the number of processors required to maintain a system's efficiency, and it lets us determine scalability with respect to the number of processors, their speed, and the communication bandwidth of the interconnection network. The isoefficiency function also succinctly captures the characteristics of a particular algorithm/architecture combination in a single expression, letting us compare various combinations for a range of problem sizes and numbers of processors. Thus, we can determine the best combination for a problem without explicitly analyzing all possible combinations under all possible conditions. (The sidebar on page 14 defines many basic concepts of scalability analysis and presents an example that is revisited throughout the article.)

1063-6552/93/0800-0012 $3.00 © 1993 IEEE

Scalable parallel systems

The number of processors limits a parallel system's speedup: The speedup for a single processor is one, but if more are used, the speedup is usually less than the number of processors.

Let's again consider the example in the sidebar. Figure 1 shows the speedup for a few values of n on up to 32 processors; Table 1 shows the corresponding efficiencies. The speedup does not increase linearly with the number of processors; instead, it tends to saturate. In other words, the efficiency drops as the number of processors increases. This is true for all parallel systems, and is often referred to as Amdahl's law. But the figure and table also show a higher speedup (efficiency) as the problem size increases on the same number of processors.

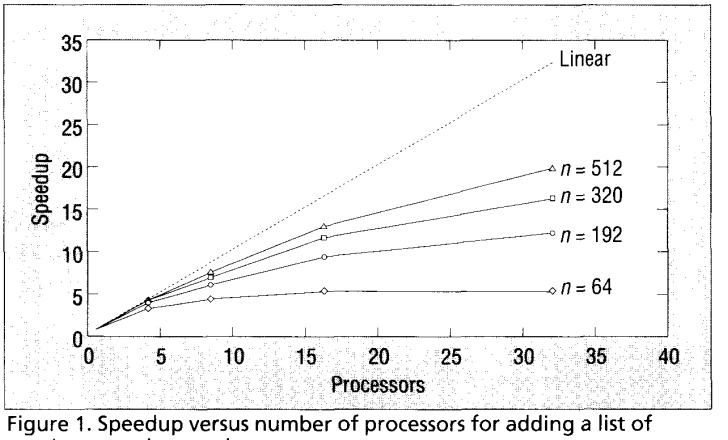


Figure 1. Speedup versus number of processors for adding a list of numbers on a hypercube.

Table 1. Efficiency as a function of n and p for adding n numbers on p-processor hypercubes.

	$p = 1$	$p = 4$	$p = 8$	$p = 16$	$p = 32$
$n = 64$	1.0	.80	.57	.33	.17
$n = 192$	1.0	.92	.80	.60	.38
$n = 320$	1.0	.95	.87	.71	.50
$n = 512$	1.0	.97	.91	.80	.62

If increasing the number of processors reduces efficiency, and increasing the problem size increases efficiency, we should be able to keep efficiency constant by increasing both simultaneously. For example, the table shows that the efficiency of adding 64 numbers on a four-processor hypercube is 0.80. When we increase p to eight and n to 192, the efficiency remains 0.80, as it does when we further increase p to 16 and n to 512. Many parallel systems behave in this way. We call them *scalable* parallel systems.

The isoefficiency function

A natural question at this point is: At what rate should we increase the problem size with respect to the number of processors to keep the efficiency fixed? The answer varies depending on the system.

In the sidebar, we noted that the sequential execution time T_1 equals the problem size W multiplied by the cost of executing each operation (t_c). Making this substitution in the efficiency equation gives us

$$E = \frac{1}{1 + \frac{T_0}{Wt_c}}$$

If the problem size W is constant while p increases, then the efficiency decreases because the total overhead T_0 increases with p. If W increases while p is constant, then, for scalable parallel systems, the efficiency increases because T_0 grows slower than $\Theta(W)$ (that is, slower than all functions with the same growth rate as W). We can maintain the efficiency for these parallel systems at a desired value (between 0 and 1) by increasing p, provided W also increases. For different parallel systems, we must increase W at different rates with respect to p to maintain a fixed efficiency. For example, W might need to grow as an exponential function of p. Such systems are poorly scalable: It is difficult to obtain good speedups for a large number of processors on such systems unless the problem size is enormous. On the other hand, if W needs to grow only linearly with respect to p, then the system is highly scalable: Its speedups increase linearly with respect to the number of processors for problem sizes increasing at reasonable rates.

For scalable parallel systems, we can maintain efficiency at a desired value ($0 < E < 1$) if T_0/W is constant:

$$E = \frac{1}{1 + \frac{T_0}{Wt_c}}$$

$$\frac{T_0}{W} = t_c\left(\frac{1-E}{E}\right)$$

$$W = \frac{1}{t_c}\left(\frac{E}{1-E}\right)T_0$$

If $K = E/(t_c(1 - E))$ is a constant that depends on the efficiency, then we can reduce the last equation to

$$W = KT_0$$

Definitions and assumptions

A parallel algorithm cannot be evaluated apart from the architecture it is implemented on, so we define a *parallel system* as the combination of a parallel algorithm and a parallel architecture. The time taken by an algorithm to execute on a single processor is its *sequential execution time*, T_1. The execution time of the corresponding parallel algorithm on p identical processors is its *parallel execution time*, T_P.

During execution, a parallel algorithm incurs overhead due to idling, communication, contention over shared data structures, and so on. The total time spent by all processors doing work that is not done by the sequential algorithm is the *total overhead*, T_o. In general, T_o is a function of the problem size and the number of processors. The total time spent by all processors is pT_P, and the total overhead is T_o, so

$$pT_P = T_1 + T_o$$

or

$$T_P = \frac{T_1 + T_o}{p}$$

A parallel system's speedup S is the ratio of sequential execution time to parallel execution time:

$$S = \frac{T_1}{T_P} = \frac{pT_1}{T_1 + T_o}$$

Its efficiency E is the ratio of the speedup to the number of processors used:

$$E = \frac{S}{p} = \frac{T_1}{T_1 + T_o} = \frac{1}{1 + \frac{T_o}{T_1}}$$

For certain parallel algorithm/architecture combinations, T_o can be negative, implying that the speedup on p processors could exceed p. This phenomenon is called *superlinear speedup*. A parallel system might exhibit such behavior if its memory is hierarchical and if access time increases (in discrete steps) with the memory used by the program. In this case, the effective computation speed of a large program could be slower on a serial processor than on a parallel computer using similar processors. This is because a sequential algorithm using M bytes of memory will use only M/p bytes on each processor of a p-processor parallel computer. Cache and virtual memory effects could reduce the serial processor's effective computation rate. (To simplify this article, we'll assume that T_o is nonnegative.)

Problem size

One way to express problem size is as a parameter of the input size. For example, for any matrix problem involving $n \times n$ matrices, the problem size could be n. But this definition allows problem size to be interpreted differently for different problems: For example, doubling the input size results in an eight-fold increase in serial execution time for matrix multiplication but only a four-fold increase for matrix addition.

A better definition would not lead to such varying interpretations; doubling the problem size would always lead to twice as much computation. Therefore, we express problem size in terms of the total number of basic operations in the problem: The problem size for $n \times n$ matrix multiplication is $\Theta(n^3)$, while that for the addition of two $n \times n$ matrices is $\Theta(n^2)$ (where $\Theta(x)$ is the set of all functions that have the same growth rate as x). To keep the problem size unique for

Through algebraic manipulations, we can use this equation to obtain W as a function of p. This function dictates how W must grow to maintain a fixed efficiency as p increases. This is the system's *isoefficiency function*. It determines the ease with which the system yields speedup in proportion to the number of processors. A small isoefficiency function implies that small increments in the problem size are sufficient to use an increasing number of processors efficiently; hence, the system is highly scalable. Conversely, a large isoefficiency function indicates a poorly scalable parallel system. Furthermore, the isoefficiency function does not exist for some parallel systems, because their efficiency cannot be kept constant as p increases, no matter how fast the problem size increases.

For the equation above, if we substitute the value of T_o from the example in the sidebar, we get $W = 2Kp \log p$. Thus, this system's isoefficiency function is $\Theta(p \log p)$. If the number of processors increases from p to p', the problem size (in this case n) must increase by a factor of $(p' \log p')/(p \log p)$ to maintain the same efficiency. In other words, increasing the number of processors by a factor of p'/p requires n to be increased by a factor of $(p' \log p')/(p \log p)$ to increase the speedup by a factor of p'/p.

In this simple example of adding n numbers, the communication overhead is a function of only p. But a typical overhead function can have several terms of different orders of magnitude with respect to both the problem size and the number of processors, making it impossible (or at least cumbersome) to obtain the isoefficiency function as a closed form function of p.

Consider a parallel system for which

$$T_o = p^{3/2} + p^{3/4} W^{3/4}$$

The equation $W = KT_o$ becomes

$$W = Kp^{3/2} + Kp^{3/4} W^{3/4}$$

For this system, it is difficult to solve for W in terms of p. However, since the condition for constant efficiency is that the ratio of T_o and W remains fixed, then if p and W increase, the efficiency will not drop if none of the terms of T_o grows faster than W. We thus balance each term of T_o against W to compute the corresponding iso-

a given problem, we define it as the number of operations the best sequential algorithm executes to solve the problem on a single processor. For some problems, the best algorithm is not known; for others, the generally best algorithm may perform worse than others for particular instances of the problem. In these cases, we can use the number of operations in the serial algorithm that is considered best for each instance. For example, for matrix multiplication, the simple $\Theta(n^3)$ algorithm is often the algorithm of choice, even though Strassen's algorithm has a better asymptotic complexity. We will use the symbol W to denote problem size. If the cost of executing each operation is t_c, then $T_1 = Wt_c$.

An example

If we add n numbers on a sequential machine, the number of operations — and hence the problem size W — equals n. If each addition takes time t_c, then the sequential execution time T_1 equals nt_c. (This is an approximation for large values of n; in reality, W is $n - 1$, and T_1 is $(n - 1)t_c$.)

Now consider a parallel algorithm for adding n numbers using a p-processor hypercube (Figure A shows this algorithm for $n = 16$ and $p = 4$). Each processor is allocated n/p numbers. In the first step, each processor locally adds its n/p numbers in $\Theta(n/p)$ time. The problem is now reduced to adding the p partial sums on p processors, which can be done by propagating and adding the partial sums. A single step consists of one addition and one nearest-neighbor communication of a partial sum (typically a single word). If we assume that it takes one unit of time to add two numbers, and one unit of time to communicate a number between two processors, then n/p time is spent adding the n/p local numbers at each processor. After the local addition, the p partial sums are added in $\log p$ steps, each consisting of one addition and one communication. Thus, the total parallel execution time T_P is $n/p + 2 \log p$.

So, of the $n/p + 2 \log p$ time units that each processor spends in parallel execution, n/p time is spent performing useful work. The remaining $2 \log p$ units of time per processor contribute to a total overhead of

$$T_o = 2p \log p$$

The speedup S and efficiency E are

$$S = \frac{T_1}{T_P} = \frac{n}{\frac{n}{p} + 2 \log p}$$

$$E = \frac{S}{p} = \frac{n}{n + 2p \log p}$$

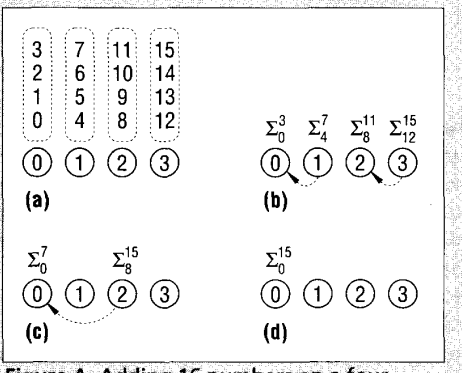


Figure A. Adding 16 numbers on a four-processor hypercube.

efficiency function. The term that causes the problem size to grow fastest with respect to p determines the system's overall isoefficiency function. Solving for the first term in the above equation gives us

$$W = Kp^{3/2} = \Theta(p^{3/2})$$

Solving for the second term gives us

$$W = Kp^{3/4}W^{3/4}$$
$$W^{1/4} = Kp^{3/4}$$
$$W = K^4p^3 = \Theta(p^3)$$

So if the problem size grows as $\Theta(p^{3/2})$ and $\Theta(p^3)$, respectively, for the first two terms of T_o, then efficiency will not decrease as p increases The isoefficiency function for this system is therefore $\Theta(p^3)$, which is the higher rate. If W grows as $\Theta(p^3)$, then T_o will remain of the same order as W.

Optimizing cost

A parallel system is *cost-optimal* if the product of the number of processors and the parallel execution time is proportional to the execution time of the best serial algorithm on a single processor:

$$pT_P \propto W$$

In the sidebar, we noted that $pT_P = T_1 + T_o$, so

$$T_1 + T_o \propto W$$

Since $T_1 = Wt_c$, we have

$$Wt_c + T_o \propto W$$
$$W \propto T_o$$

This suggests that a parallel system is cost-optimal if its overhead function and the problem size are of the same order of magnitude. This is exactly the condition required to maintain a fixed efficiency while increasing the number of processors. So, conforming to the isoefficiency relation between W and p keeps a parallel system cost-optimal as it is scaled up.

How small can an isoefficiency function be, and what is an ideally scalable parallel system? If a problem con-

sists of W basic operations, then a cost-optimal system can use no more than W processors. If the problem size grows at a rate slower than $\Theta(p)$ as the number of processors increases, then the number of processors will eventually exceed W. Even in an ideal parallel system with no communication or other overhead, the efficiency will drop because the processors exceeding W will have no work to do. So, the problem size has to increase at least as fast as $\Theta(p)$ to maintain a constant efficiency; hence $\Theta(p)$ is the lower bound on the isoefficiency function. It follows that the isoefficiency function of an ideally scalable parallel system is $\Theta(p)$.

Degree of concurrency

The lower bound of $\Theta(p)$ is imposed on the isoefficiency function by the algorithm's *degree of concurrency*: the maximum number of tasks that can be executed simultaneously in a problem of size W. This measure is independent of the architecture. If $C(W)$ is an algorithm's degree of concurrency, then given a problem of size W, at most $C(W)$ processors can be employed effectively. For example, using Gaussian elimination to solve a system of n equations with n variables, the total amount of computation is $\Theta(n^3)$. However, the n variables have to be eliminated one after the other, and eliminating each variable requires $\Theta(n^2)$ computations. Thus, at most $\Theta(n^2)$ processors can be kept busy at a time.

Now if $W = \Theta(n^3)$ for this problem, then the degree of concurrency is $\Theta(W^{2/3})$. Given a problem of size W, at most $\Theta(W^{2/3})$ processors can be used, so given p processors, the size of the problem should be at least $\Theta(p^{3/2})$ in order to use all the processors. Thus, the isoefficiency function of this computation due to concurrency is $\Theta(p^{3/2})$.

The isoefficiency function due to concurrency is optimal — $\Theta(p)$ — only if the algorithm's degree of concurrency is $\Theta(W)$. If it is less than $\Theta(W)$, then the isoefficiency function due to concurrency is worse (greater) than $\Theta(p)$. In such cases, the system's overall isoefficiency function is the maximum of the isoefficiency functions due to concurrency, communication, and other overhead.

Isoefficiency analysis

Isoefficiency analysis lets us test a program's performance on a few processors and then predict its performance on a larger number of processors. It also lets us study system behavior when other hardware parameters change, such as processor and communication speeds.

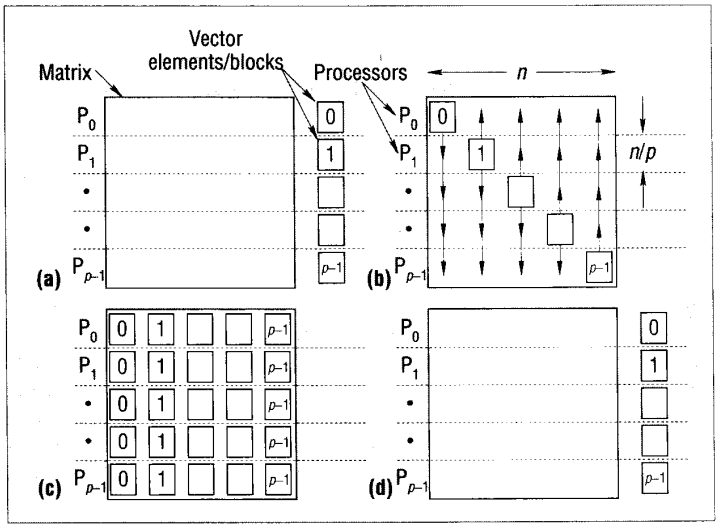


Figure 2. Multiplication of an $n \times n$ matrix with an $n \times 1$ vector using "rowwise" striped data partitioning.

Comparing algorithms

We often must compare the performance of two parallel algorithms for a large number of processors. The isoefficiency function gives us the tool to do so. The algorithm with the smaller isoefficiency function yields better performance as the number of processors increases.

Consider the problem of multiplying an $n \times n$ matrix with an $n \times 1$ vector. The number of basic operations (the problem size W) for this matrix-vector product is n^2. If the time taken by a single addition and multiplication operation together is t_c, then the sequential execution time of this algorithm is $n^2 t_c$ (that is, $T_1 = n^2 t_c$).

Figure 2 illustrates a parallel version of this algorithm based on a striped partitioning of the matrix and the vector. Each processor is assigned n/p rows of the matrix and n/p elements of

the vector. Since the multiplication requires the vector to be multiplied with each row of the matrix, every processor needs the entire vector. To accomplish this, each processor broadcasts its n/p elements of the vector to every other processor (this is called an *all-to-all broadcast*). Each processor then has the vector available locally and n/p rows of the matrix. Using these, it computes the dot products locally, giving it n/p elements of the resulting vector.

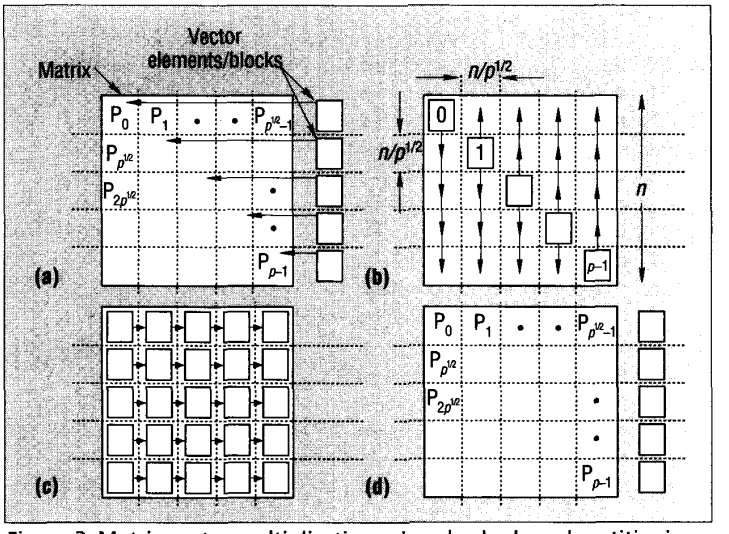


Figure 3. Matrix-vector multiplication using checkerboard partitioning.

Let's now analyze this algorithm on a hypercube. The all-to-all broadcast can be performed in $t_s \log p + t_w n(p-1)/p$ (t_s is the startup time of the communication network, and t_w is the per-word transfer time).[7] For large values of p we can approximate this as $t_s \log p + t_w n$. Assuming that an addition/multiplication pair takes t_c units of time, each processor spends $t_c n^2/p$ units of time in multiplying its n/p rows with the vector. Thus, the parallel execution time of this procedure is

$$T_P = t_c(n^2/p) + t_s \log p + t_w n$$

The speedup and efficiency are

$$S = \frac{p}{1 + \frac{p(t_s \log p + t_w n)}{t_c n^2}}$$

$$E = \frac{1}{1 + \frac{t_s p \log p + t_w np}{t_c n^2}}$$

Using the relation $T_o = pT_P - T_1$, the total overhead is

$$T_o = t_s p \log p + t_w np$$

Now we can determine the isoefficiency function. Rewriting the equation $W = KT_o$ using only the first term of T_o gives the isoefficiency term due to the message startup time:

$$W = Kt_s p \log p$$

Similarly, we can balance the second term of T_o (due to per-word transfer time) against the problem size W:

$$n^2 = Kt_w np$$

$$n = Kt_w p$$

$$W = n^2 = K^2 t_w^2 p^2$$

From the equations for both terms, we can infer that the problem size needs to increase with the number of processors at an overall rate of $\Theta(p^2)$ to maintain a fixed efficiency.

Another example

Now instead of partitioning the matrix into stripes, let's use *checkerboard partitioning*: divide it into p squares, each of dimensions $(n/\sqrt{p}) \times (n/\sqrt{p})$.[7] Figure 3 shows the algorithm.

The vector is distributed along the last column of the mesh. In the first step, all processors of the last column send their $n/\sqrt{p}$ elements of the vector to the diagonal processor of their respective rows (Figure 3a). Then the processors perform a columnwise one-to-all broadcast of the $n/\sqrt{p}$ elements (Figure 3b). The vector is then aligned along the rows of the matrix. Each processor performs n^2/p multiplications and locally adds the $n/\sqrt{p}$ sets of products. Each processor now has $n/\sqrt{p}$ partial sums that need to be accumulated along each row to obtain the product vector (Figure 3c). The last step is a single-node accumulation of the $n/\sqrt{p}$ values in each row, with the last processor of the row as the destination (Figure 3d).

On a hypercube with store-and-forward routing, the first step can be performed in at most $t_s + t_w(n/\sqrt{p}) \log \sqrt{p}$ time.[7] The second step can be performed in $(t_s + t_w n/\sqrt{p}) \log \sqrt{p}$ time. If a multiplication and an addition are assumed to take t_c units of time, then each processor spends about $t_c n^2/p$ time performing computation. If the product vector must be placed in the last column (like the starting vector), then a single-node accumulation of vector components of size $n/\sqrt{p}$ must be performed in each row. Ignoring the time needed to perform additions during this step, the accumulation can be performed with a communication time of $(t_s + t_w n/\sqrt{p}) \log \sqrt{p}$. The total parallel execution time is

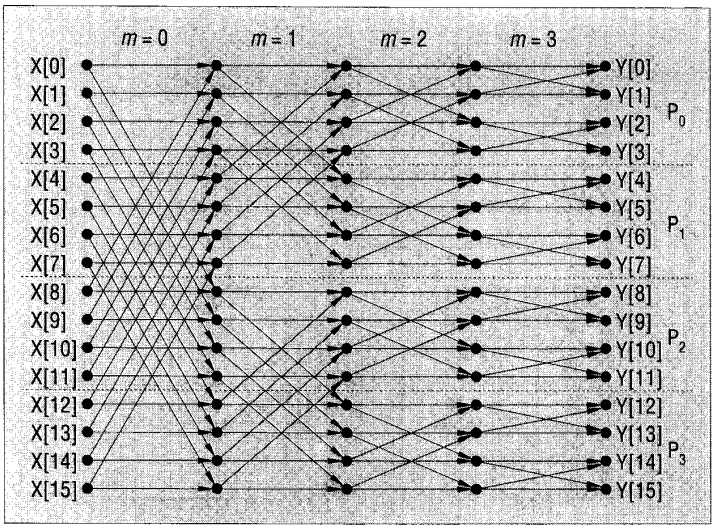


Figure 4. A 16-point fast Fourier transform on four processors. P_i is processor i and m is the iteration number.

$$T_P = t_c(n^2/p) + t_s + 2t_s \log \sqrt{p} + 3t_w(n/\sqrt{p}) \log \sqrt{p}$$

We can approximate this as

$$T_P = t_c(n^2/p) + t_s \log p + (3/2)t_w(n/\sqrt{p}) \log p$$

So, the total overhead T_o is

$$T_o = t_s p \log p + (3/2)t_w n \sqrt{p} \log p$$

As before, we equate each term of T_o with the problem size W. For the isoefficiency due to t_s, we get $W \propto Kt_s p \log p$. For isoefficiency due to t_w, we get

$$n^2 t_c = K(3/2)t_w n \sqrt{p} \log p$$

$$n = K(3/2)(t_w/t_c)\sqrt{p} \log p$$

$$n^2 = K^2(9/4)(t_w^2/t_c^2)\, p \log^2 p$$

The isoefficiency due to t_w is $\Theta(p \log^2 p)$, which is also the overall isoefficiency, since it dominates the $\Theta(p \log p)$ term due to t_s.

Based on this and the previous example, the isoefficiency function of the stripe-based algorithm is $\Theta(p^2)$, which is higher than the $\Theta(p \log^2 p)$ of the checkerboard-based algorithm. This implies that the stripe-based version is less scalable; as the number of processors increases, it requires much larger problem sizes to yield the same efficiencies as the checkerboard-based version.

Machine-specific parameters

Changing processor and communication speeds affects the scalability of some parallel systems only moderately; it affects others significantly. Isoefficiency analysis can help predict the effects of changes in such machine-specific parameters.

Consider the Cooley-Tukey algorithm for computing an n-point, single-dimensional, unordered, radix-2 fast Fourier transform.[7,8] The sequential complexity of this algorithm is $\Theta(n \log n)$. We'll use a parallel version based on the binary exchange method for a d-dimensional ($p = 2^d$) hypercube (see Figure 4). We partition the vectors into blocks of n/p contiguous elements ($n = 2^r$) and assign one block to each processor. In the mapping shown in Figure 4, the vector elements on different processors are combined during the first d iterations while the pairs of elements combined during the last $(r - d)$ iterations reside on the same processors. Hence, this algorithm involves interprocessor communication only during $d = \log p$ of the $\log n$ iterations. Each communication operation exchanges n/p words of data, so communication time over the entire algorithm is $(t_s + t_w n/p) \log p$. During each iteration, a processor updates n/p elements of vector R. If a complex multiplication and addition take time t_c, then the parallel execution time is

$$T_P = t_c(n/p) \log n + t_s \log p + t_w(n/p) \log p$$

The total overhead T_o is

$$T_o = t_s p \log p + t_w n \log p$$

We know that the problem size for an n-point fast Fourier transform is

$$W = n \log n$$

Using the same method as in the previous subsection, we can now determine the system's isoefficiency by equating the problem size with each term in the total overhead. For the first term (t_s), $W \propto t_s p \log p$, which corresponds to an isoefficiency function of $\Theta(p \log p)$. We can similarly determine the isoefficiency for the second term (t_w):

$$n \log n = Kt_w n \log p$$

$$\log n = Kt_w \log p$$

$$n = p^{K\frac{t_w}{t_c}}$$

$$n \log n = Kt_w p^{Kt_w} \log p$$

$$W = \frac{E}{1-E}\frac{t_w}{t_c} p^{\frac{E}{1-E}\frac{t_w}{t_c}} \log p$$

For this last equation, if $t_w E/(t_c(1 - E))$ is less than 1, then

W's rate of growth is less than $\Theta(p \log p)$, so the overall isoefficiency function is $\Theta(p \log p)$. But if $t_w E/(t_c(1-E))$ is greater than 1, then the overall isoefficiency function is greater than $\Theta(p \log p)$. The isoefficiency function depends on the relative values of $E/(1-E)$, t_w, and t_c. Thus, this algorithm is unique in that the isoefficiency function is a function not only of the desired efficiency, but also of the hardware-dependent parameters. In fact, the efficiency corresponding to $t_w E/(t_c(1-E)) = 1$ — that is, $E/(1-E) = t_c/t_w$, or $E = t_c/(t_c + t_w)$ — acts as a threshold value for efficiency. For a hypercube, efficiencies up to this value can be obtained easily. But much higher efficiencies can be obtained only if the problem size is extremely large.

Let's examine the effect of the value of $t_w E/(t_c(1-E))$ on the isoefficiency function. If $t_w = t_c$, then the isoefficiency function is $E/(1-E)p^{E/(1-E)} \log p$. Now for $E/(1-E) \leq 1$ (that is, $E \leq 0.5$), the overall isoefficiency is $\Theta(p \log p)$, but for $E > 0.5$ it is much worse. For instance, if $E = 0.9$, then $E/(1-E) = 9$ and the isoefficiency function is $\Theta(p^9 \log p)$. Now if $t_w = 2t_c$ and the threshold efficiency is 0.33, then the isoefficiency function for $E = 0.33$ is $\Theta(p \log p)$, for $E = 0.5$ it is $\Theta(p^2 \log p)$, and for $E = 0.9$ it is $\Theta(p^{18} \log p)$.

These examples show that the efficiency we can obtain for reasonable problem sizes is limited by the ratio of the CPU speed to the hypercube's communication bandwidth. We can raise this limit by increasing the bandwidth, but making the CPU faster without improving the bandwidth lowers this threshold. In other words, this algorithm performs poorly on a hypercube whose communication and computation speeds are not balanced. However, the algorithm is fairly scalable on a balanced hypercube with an overall isoefficiency function of $\Theta(p \log p)$, and good efficiencies can be expected for a reasonably large number of processors.

Some parallel algorithms that seem attractive because of their low overhead have limited concurrency, making them perform poorly as the number of processors grows. Isoefficiency analysis can capture this effect.

Concurrency

Some parallel algorithms that seem attractive because of their low overhead have limited concurrency, making them perform poorly as the number of processors grows. Isoefficiency analysis can capture this effect.

Consider Dijkstra's all-pairs shortest-path problem for a dense graph with n vertices.[7,9] The problem involves finding the shortest path between each pair of vertices. The best-known serial algorithm takes $\Theta(n^3)$ time. We can also solve this problem by executing one instance of the single-source shortest-path algorithm for each of the n vertices. The latter algorithm determines the shortest path from one vertex to every other vertex in the graph. Its sequential complexity is $\Theta(n^2)$.

We can derive a simple parallel version of this algorithm by executing a single-source shortest-path problem independently on each of n processors. Since each of these computations is independent of the others, the parallel algorithm requires no communication, making it seem that it is the best possible algorithm. But the algorithm can use at most n processors ($p = n$), and since the problem size W is $\Theta(n^3)$, W must grow at least as $\Theta(p^3)$ to use more processors. So the overall isoefficiency is relatively high; other algorithms with better isoefficiencies are available.

Contention for shared data structures

An algorithm can have low communication overhead and high concurrency, but still have overhead from contention over shared data structures. Such overhead is difficult to model, making it difficult to compute the parallel execution time. However, we can still use isoefficiency analysis to determine the scalability.

Consider an application that solves discrete optimization problems by performing a depth-first search of large unstructured trees. Some parallel algorithms solve this problem by using a dynamic load-balancing strategy.[10,11] All work is initially assigned to one processor. An idle processor P_i selects a processor P_a using some selection criterion and sends it a work request. If processor P_a has no work, it responds with a reject message; otherwise, it partitions its work into two parts and sends one part to P_i (as long as the work is larger than some minimum size). This process continues until all processors exhaust the available work.

One selection criterion — *global round robin* — maintains a global pointer G at one of the processors. This

pointer initially points to the first processor. When an idle processor needs to select P_a, it reads the current value of G, and requests work from P_G. The pointer is incremented by one (modulo p) before the next request is processed. The pointer distributes the work requests evenly over the processors.

The nondeterministic nature of this algorithm makes it impossible to estimate the exact parallel execution time beforehand. We can, however, set an upper bound on the communication cost.[7,10] Under certain assumptions,[10] the upper bound on the number of communications is $O(p \log W)$ (that is, it is of the same order or smaller than $p \log W$). If each communication takes $O(\log p)$ time, then the total overhead from the communication of work is bounded by $O(p \log p \log W)$. As before, we can equate this term with the problem size to derive the isoefficiency due to communication overhead:

$$W \propto O(p \log p \log W)$$

If we take the W on the right hand side of this expression, put the *value* of W in its place, and ignore the double log terms, then the isoefficiency due to communication overhead is $O(p \log^2 p)$.

But this term does not specify the system's overall isoefficiency because the algorithm also has overhead due to contention: Only one processor can access the global variable at a time; others must wait. So, we must also analyze the system's isoefficiency due to contention.

The global variable is accessed a total of $O(p \log W)$ times (for the read and increment operations). If the processors are used efficiently, then the total execution time is $\Theta(W/p)$. If there is no contention while solving a problem of size W on p processors, then W/p is much greater than the total time during which the shared variable is accessed. Now, as we increase the number of processors, the total execution time (W/p) decreases, but the number of times the shared variable is accessed increases. At some point, the shared variable access becomes a bottleneck, and the overall execution time cannot be reduced further. We can eliminate this bottleneck by increasing W at a rate such that the ratio between W/p and $O(p \log W)$ remains the same. Equating W/p and $O(p \log W)$ and then simplifying yields an isoefficiency of $O(p^2 \log p)$. Thus, since the isoefficiency due to contention dominates the isoefficiency due to communication, the overall isoefficiency is $O(p^2 \log p)$. (It has been shown elsewhere that dynamic load-balancing schemes with better isoefficiency functions outperform those with poorer isoefficiency functions.[10])

The isoefficiency metric is useful when we want performance to increase at a linear rate with the number of processors: If the problem size grows at the rate specified by the isoefficiency function, then the system's speedup is linear. In some cases, though, we might not want (or be able) to increase the problem size at the rate specified by the isoefficiency function; if the problem size grows at a smaller rate, then the speedup is sublinear.

For a given growth rate, we can use the speedup curve as a scalability metric. If the problem size increases at a linear rate with the number of processors, the curve shows *scaled speedup*.[2] The growth rate can also be constrained by the computer's memory, in which case the problem size increases at the fastest rate allowed by the available memory.[2,4,6]

In many situations, the growth rate is dictated by the time available to solve the problem, in which case the problem size increases with the number of processors in such a way that the run time remains constant.[2,4,6] We can also keep the problem size fixed and use the speedup curve as a scalability metric.[12]

There are interesting relationships between isoefficiency and some of these metrics. If the isoefficiency function is greater than $\Theta(p)$, then the problem size for a scalable parallel system cannot increase indefinitely while maintaining a fixed execution time, no matter how many processors are used.[1,12] Also, for a class of parallel systems, the isoefficiency function specifies the relationship between the problem size's growth rate and the number of processors on which the problem executes in minimum time.[12]

Acknowledgments

This work was supported by Army Research Office grant 28408-MA-SDI to the University of Minnesota, and by the Army High Performance Computing Research Center at the University of Minnesota. We also thank Daniel Challou and Tom Nurkkala for their help in preparing this article.

References

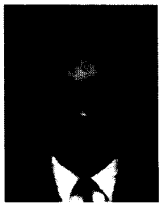
1. V. Kumar and A. Gupta, "Analyzing Scalability of Parallel Algorithms and Architectures," Tech. Report 91-18, Computer Science Dept., Univ. of Minnesota, Minneapolis, 1991.

2. J.L. Gustafson, "Reevaluating Amdahl's Law," *Comm. ACM*, Vol. 31, No. 5, 1988, pp. 532-533.

3. J.L. Gustafson, "The Consequences of Fixed-Time Performance Measurement," *Proc. 25th Hawaii Int'l Conf. System Sciences, Vol.*

III, IEEE Computer Soc. Press, Los Alamitos, Calif., 1992, pp. 113-124.

4. X.-H. Sun and L.M. Ni, "Another View of Parallel Speedup," *Proc. Supercomputing '90*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1990, pp. 324-333.
5. X.-H. Sun and D.T. Rover, "Scalability of Parallel Algorithm-Machine Combinations," Tech. Report IS-5057, Ames Lab., Iowa State Univ., 1991.
6. P.H. Worley, "The Effect of Time Constraints on Scaled Speedup," *SIAM J. Scientific and Statistical Computing*, Vol. 11, No. 5, Sept. 1990, pp. 838-858.
7. V. Kumar et al., *Introduction to Parallel Computing: Algorithm Design and Analysis*, Benjamin/Cummings, Redwood City, Calif., to be published (1994).
8. A. Gupta and V. Kumar, "The Scalability of FFT on Parallel Computers," *IEEE Trans. Parallel and Distributed Systems*, Vol. 4, No. 7, July 1993.
9. V. Kumar and V. Singh, "Scalability of Parallel Algorithms for the All-Pairs Shortest-Path Problem," *J. Parallel and Distributed Computing*, Vol. 13, No. 2, Oct. 1991, pp. 124-138.
10. V. Kumar, A. Grama, and V.N. Rao, "Scalable Load-Balancing Techniques for Parallel Computers," Tech. Report 91-55, Computer Science Dept., Univ. of Minnesota, Minneapolis, 1991.
11. V. Kumar and V.N. Rao, "Parallel Depth-First Search, Part II: Analysis," *Int'l J. Parallel Programming*, Vol. 16, No. 6, Dec. 1987, pp. 501-519.
12. A. Gupta and V. Kumar, "Performance Properties of Large-Scale Parallel Systems," to appear in *J. Parallel and Distributed Computing*, Nov. 1993.



Ananth Y. Grama is a doctoral candidate in computer science at the University of Minnesota. His research interests include the design and analysis of scalable parallel algorithms, and architecture-independent parallel programming. He received his MS in computer engineering from Wayne State University, Detroit, in 1990, and his BE in computer science from the University of Roorkee, India, in 1989.

Anshul Gupta is a doctoral candidate in computer science at the University of Minnesota. His research interests include parallel algorithms, scientific computing, and scalability and performance evaluation of parallel and distributed systems. He received his B.Tech. degree in computer science from the Indian Institute of Technology, New Delhi, in 1988.

Vipin Kumar is an associate professor in the Department of Computer Science at the University of Minnesota. His research interests include parallel processing and artificial intelligence. He is coeditor of *Search in Artificial Intelligence*, *Parallel Algorithms for Machine Intelligence and Vision*, and *Introduction to Parallel Computing*. Kumar received his PhD in computer science from the University of Maryland, College Park, in 1982; his ME in electronics engineering from Philips International Institute, Eindhoven, The Netherlands, in 1979; and his BE in electronics and communication engineering from the University of Roorkee, India, in 1977. He is a senior member of IEEE, and a member of ACM and the American Association for Artificial Intelligence.

The authors can be reached at the Department of Computer Science, 200 Union St. SE, University of Minnesota, Minneapolis, MN 55455; Internet: kumar, ananth, or agupta@cs.umn.edu