CHAPTER 2

Analytical Modeling of Parallel Programs

A sequential algorithm is usually evaluated in terms of its execution time, expressed as a function of the size of its input. The execution time of a parallel algorithm depends not only on input size but also on the architecture of the parallel computer and the number of processors. Hence, a parallel algorithm cannot be evaluated in isolation from a parallel architecture. A *parallel system* is the combination of an algorithm and the parallel architecture on which it is implemented. In this chapter, we study various metrics for evaluating the performance of parallel systems. The scalability of a parallel algorithm on an architecture is a measure of its ability to achieve performance proportional to the number of processors. We address scalability in detail and describe a metric to analyze it.

2.1 Performance Metrics for Parallel Systems

In this section, we introduce some metrics that are commonly used to measure the performance of parallel systems.

2.1.1 Run Time

The serial run time of a program is the time elapsed between the beginning and the end of its execution on a sequential computer. The *parallel run time* is the time that elapses from the moment that a parallel computation starts to the moment that the last processor finishes execution. We denote the serial run time by T_s and the parallel run time by T_P. We also formalize the size of the problem W as the serial runtime of the program to solve the problem instance. This is in contrast to other measures of problem size such as the input size or output size.

2.1.2 Speedup

When evaluating a parallel system, we are often interested in knowing how much performance gain is achieved by parallelizing a given application over a sequential implementation. Speedup is a measure that captures the relative benefit of solving a problem in parallel. It is defined as the ratio of the time taken to solve a problem on a single processor to the time required to solve the same problem on a parallel computer with p identical processors. We denote speedup by the symbol S.

For a given problem, more than one sequential algorithm may be available, but all of these may not be equally suitable for parallelization. When a serial computer is used, it is natural to use the sequential algorithm that solves the problem in the least amount of time. Given a parallel algorithm, it is fair to judge its performance with respect to the fastest sequential algorithm for solving the same problem on a single processor. Sometimes, the fastest sequential algorithm to solve a problem is not known, or its run time has a large constant that makes it impractical to implement. In such cases, we will take the fastest known algorithm that would be a practical choice for a serial computer to be the best sequential algorithm. We compare the performance of a parallel algorithm to solve a problem with that of the best sequential algorithm to solve the same problem. We formally define the *speedup* S as the ratio of the serial run time of the best sequential algorithm for solving a problem to the time taken by the parallel algorithm to solve the same problem on p processors. The p processors used by the parallel algorithm are assumed to be identical to the one used by the sequential algorithm.

Theoretically, speedup can never exceed the number of processors, p. If the best sequential algorithm takes T_S units of time to solve a given problem on a single processor, then a speedup of p can be obtained on p processors if none of the processors spends more than T_S/p time. A speedup greater than p is possible only if each processor spends less than T_S/p time solving the problem. In that case, a single processor could emulate the p processors and solve the problem in fewer than T_S units of time. This is a contradiction because speedup, by definition, is computed with respect to the best sequential algorithm. If T_S is the serial run time of that algorithm, then the problem cannot be solved in less than time T_S on a single processor.

In practice, a speedup greater than p is sometimes observed (a phenomenon known as *superlinear speedup*). This is usually due either to a nonoptimal sequential algorithm (Problem **??**) or to hardware characteristics that put the sequential algorithm at a disadvantage. For example, the data for a problem might be too large to fit into the main memory of a single processor, thereby degrading its performance due to the use of secondary storage. But when partitioned among several processors, the individual data-partitions would be small enough to fit into their respective processors' main memories. In the remainder of this book, we disregard superlinear speedup due to hierarchical memory.

2.1.3 Efficiency

Only an ideal parallel system containing p processors can deliver a speedup equal to p. In practice, ideal behavior is not achieved because while executing a parallel algorithm, the processors cannot devote 100 percent of their time to the computations of the algorithm. As we saw in Example **??**, part of the time required by the processors to compute the sum of n numbers is spent in communication. *Efficiency* is a measure of the fraction of time for which a processor is usefully employed; it is defined as the ratio of speedup to the number of processors. In an ideal parallel system, speedup is equal to p and efficiency is equal to one. In practice, speedup is less than p and efficiency is between zero and one, depending on the degree of effectiveness with which the processors are utilized. We denote efficiency by the symbol E. Mathematically, it is given by

$$E = \frac{S}{p}. \tag{2.1}$$

2.2 Scalability of Parallel Systems

Very often, programs are designed and tested for smaller problems on fewer processors. However, the real problems the programs are intended to solve are much larger, and the machines contain larger number of processors. Whereas code development is simplified by using scaled down versions of the machine and the problem, the performance and correctness of the code is much more difficult to establish based on scaled down systems. In this section, we will investigate on techniques for evaluating the performance of parallel programs using analytical tools. We will also investigate the effect of increasing number of processors and problem size on the overall performance.

Why is performance extrapolation so difficult? The difficulty in extrapolating performance of parallel programs stems from a conspiracy between constants and asymptotics yielding potentially conflicting results as the number of processors change. Let us consider two parallel programs running on 8 processors of a parallel machine. Program S1 solves a particular problem in time 12 seconds and program S2 solves the same problem instance in time 10 seconds. This would lead us to believe that program S2 is superior to program S1. Such a conclusion could be erroneous if generalized to larger problem sizes, or to larger configurations of the machine.

2.2.1 Sources of Overhead in Parallel Programs

To understand the scaling of performance of parallel programs, we must first investigate the sources of overheads in programs. The typical execution profile of a parallel program is illustrated in Figure 2.2. In addition to performing essential computation (computation that would be performed by the serial program for solving the same problem instance), the parallel program also spends time in interprocessor communication, idling due to load imbalance and extra computation (computation not performed by the serial formulation).

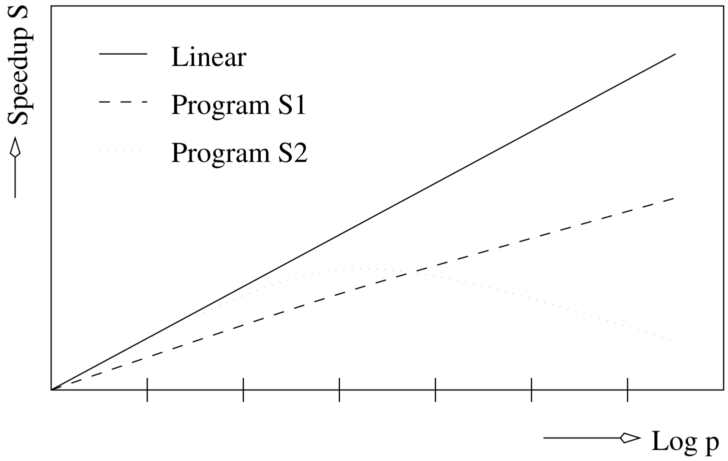

Figure 2.1 Speedup curves of two parallel programs for computing a 32K point FFT on a parallel computer.

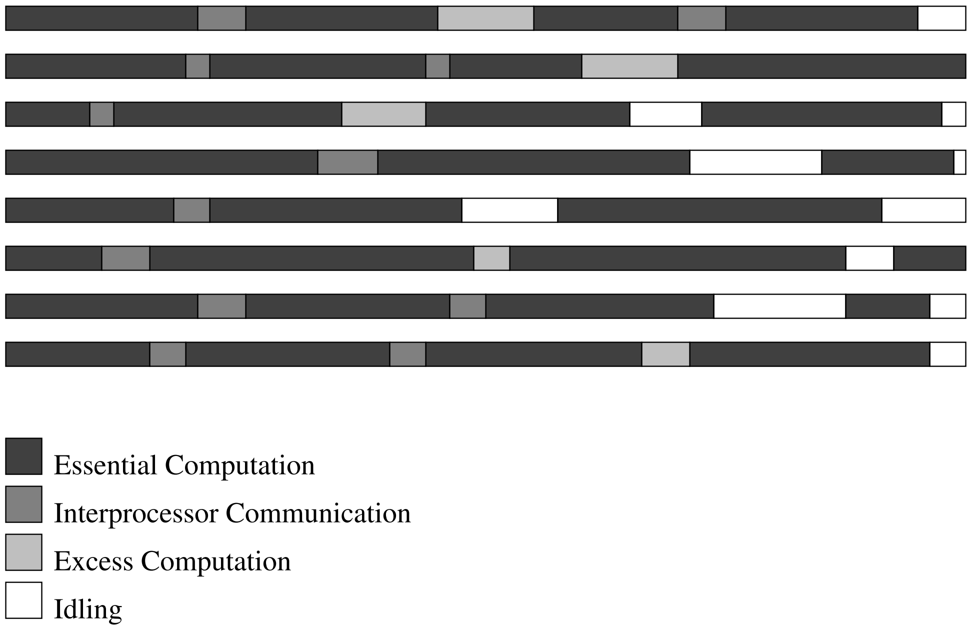

Figure 2.2 The execution profile of a parallel program executing on eight processors. Profile indicates times spent performing computation, communication, excess computation and idling overheads.

Interprocessor Communication Any nontrivial parallel system requires communication among processors. The time to transfer data between processors is usually the most significant source of parallel processing overhead.

Load Imbalance In many parallel applications (for example, search and optimization), it is impossible (or at least difficult) to predict the size of the subtasks assigned to various processors. Hence, the problem cannot be subdivided statically among the processors while maintaining uniform work load. If different processors have different work loads, some processors may be idle during part of the time that others are working on the problem.

Often some or all processors must synchronize at certain points during the parallel program execution. If all processors are not ready for synchronization at the same time, then the ones that are ready sooner will be idle until all the rest are ready. A special case of overhead due to processor idling is the presence of a sequential component in the parallel algorithm. Part of an algorithm may be unparallelizable, allowing only a single processor to work on it. While this processor is working on the serial part, all the other processors must wait. This contributes to the idling overhead associated with the program.

Extra Computation The fastest known sequential algorithm for a problem may be difficult or impossible to parallelize, forcing us to use a parallel algorithm based on a poorer but easily parallelizable (that is, one with a higher degree of concurrency) sequential algorithm. The difference in computation performed by the parallel program and the best serial program counts as excessive computation overhead incurred by the parallel program.

A parallel algorithm based on the best serial algorithm may still perform more aggregate computation than the serial algorithm. An example of such an algorithm is fast Fourier transform algorithm. In its serial version, the results of certain computations can be reused. However, in the parallel version, these results cannot be reused because they are generated by different processors. Therefore, some computations are performed multiple times on different processors.

Quantifying Parallel Overheads

The overheads incurred by a parallel program are encapsulated into a single expression referred to as the *overhead function* We define *total overhead* or the *overhead function* of a parallel system as the part of its cost (processor-time product) that is not incurred by the fastest known serial algorithm on a sequential computer. It is the total time collectively spent by all the processors in addition to that required by the fastest known sequential algorithm for solving the same problem on a single processor. We denote the overhead function of a parallel system by the symbol T_o. T_o is a function of W and p, and we often write it as $T_o(W, p)$.

The cost of solving a problem of size W on p processors, or the total time spent in solving a problem summed over all processors, is pT_P. W units of this time are spent performing useful work, and the remainder is overhead. Therefore, the relation between cost (pT_P), problem size (W), and the overhead function (T_o) is given by

$$T_o \quad = \quad pT_P - W. \tag{2.2}$$

2.2.2 Scaling Characteristics of Parallel Programs

The efficiency of a parallel program is given by

$$E = \frac{W}{pT_P}$$

Using the expression for parallel overhead (Equation 2.2), we can rewrite this expression as

$$E = \frac{1}{1 + \frac{T_o}{W}} \tag{2.3}$$

. The total overhead function T_o is an increasing function of p. This is because every program must contain some serial component. If this serial component of the program takes time t_{serial}, then during this time, all the other processors must be idle. This corresponds to a total overhead function of $(p - 1) \times t_{serial}$. Therefore, the total overhead function T_o grows atleast linearly with p. In addition, due to communication, idling and excess computation, this function may grow superlinearly in the number of processors. Equation 2.3 gives us several interesting insights into the scaling of parallel programs. First, for a given problem size (i.e. the value of W remains constant), as we increase the number of processors, T_o increases. In such a scenario, it is clear from Equation 2.3 that the overall efficiency of the parallel program goes down. This characteristic of decreasing efficiency with increasing number of processors for a given problem size is common to all parallel programs and is illustrated in Figure **??**(a).

Let us now investigate the effect of increasing the problem size keeping the number of processors constant. We know that the total overhead function T_o is a function of both problem size W and the number of processors p. In many cases, T_o grows sub-linearly with respect to W.

2.2.3 The Isoefficiency Metric of Scalability

The class of scalable parallel programs demonstrate some very interesting properties; namely, increasing p keeping w fixed decreases E, and increasing W keeping p fixed increasing E. So by increasing p and W simultaneously, it is possible to keep E constant. We use this property to formalize the notion of a scalable system.

A scalable parallel system is one in which the efficiency can be kept fixed as the number of processors is increased, provided that the problem size is also increased. It is useful to determine the rate at which the problem size must increase with respect to the number of processors to keep the efficiency fixed. For different parallel systems, the problem size must increase at different rates in order to maintain a fixed efficiency as the number of processors is increased. This rate determines the degree of scalability of the parallel system.

Let us now investigate metrics for quantitatively determining the degree of scalability of a parallel system. However, before we do that, we must define the notion of *problem size* precisely.

Problem Size When analyzing parallel systems, we frequently encounter the notion of the size of the problem being solved. So far, we have used the term *problem size* informally, without giving a precise definition. A naive way to express problem size is as a parameter of the input size; for instance, n in case of a matrix operation involving $n \times n$ matrices. A drawback of this definition is that the interpretation of problem size changes from one problem to another. For example, doubling the input size results in an eight-fold increase in the execution time for matrix multiplication and a four-fold increase for matrix addition (assuming that the conventional $\Theta(n^3)$ algorithm is the best matrix multiplication algorithm, and disregarding more complicated algorithms with better asymptotic complexities).

A consistent definition of the size or the magnitude of the problem should be such that, regardless of the problem, doubling the problem size always means performing twice the amount of computation. Therefore, we choose to express problem size in terms of the total number of basic operations required to solve the problem. By this definition, the problem size is $\Theta(n^3)$ for $n \times n$ matrix multiplication (assuming the conventional algorithm) and $\Theta(n^2)$ for $n \times n$ matrix addition. In order to keep it unique for a given problem, we define *problem size* as the number of basic computation steps in the best sequential algorithm to solve the problem on a single processor, where the best sequential algorithm is defined as in Section 2.1.2. Because it is defined in terms of sequential time complexity, the problem size is a function of the size of the input. The symbol we use to denote problem size is W.

In the remainder of this chapter, we assume that it takes unit time to perform one basic computation step of an algorithm. This assumption does not affect the analysis of any parallel system because the other hardware-related constants, such as message startup time, per-word transfer time, and per-hop time, can be normalized with respect to the time taken by a basic computation step. With this assumption, the problem size W is equal to the serial run time T_S of the fastest known algorithm to solve the problem on a sequential computer.

2.2.4 The Isoefficiency Function

Parallel execution time can be expressed as a function of problem size, overhead function, and the number of processors. Rewriting Equation 2.2, we get the following expressions for parallel run time:

$$T_P = \frac{W + T_o(W, p)}{p} \tag{2.4}$$

The resulting expression for speedup is

$$S = \frac{W}{T_P}$$

$$= \frac{Wp}{W + T_o(W, p)}. \tag{2.5}$$

Finally, we write the expression for efficiency as

$$\begin{aligned} E &= \frac{S}{p} \\ &= \frac{W}{W + T_o(W, p)} \\ &= \frac{1}{1 + T_o(W, p)/W}. \end{aligned} \tag{2.6}$$

In Equation 2.6, if the problem size is kept constant and p is increased, the efficiency decreases because the total overhead T_o increases with p. If W is increased keeping the number of processors fixed, then for scalable parallel systems, the efficiency increases. This is because T_o grows slower than $\Theta(W)$ for a fixed p. For these parallel systems, efficiency can be maintained at a desired value (between 0 and 1) for increasing p, provided W is also increased.

For different parallel systems, W must be increased at different rates with respect to p in order to maintain a fixed efficiency. For instance, in some cases, W might need to grow as an exponential function of p to keep the efficiency from dropping as p increases. Such parallel systems are poorly scalable. The reason is that on these parallel systems it is difficult to obtain good speedups for a large number of processors unless the problem size is enormous. On the other hand, if W needs to grow only linearly with respect to p, then the parallel system is highly scalable. That is because it can easily deliver speedups proportional to the number of processors for reasonable problem sizes.

For scalable parallel systems, efficiency can be maintained at a fixed value (between 0 and 1) if the ratio T_o/W in Equation 2.6 is maintained at a constant value. For a desired value E of efficiency,

$$\begin{aligned} E &= \frac{1}{1 + T_o(W, p)/W}, \\ \frac{T_o(W, p)}{W} &= \frac{1 - E}{E}, \\ W &= \frac{E}{1 - E} T_o(W, p). \end{aligned} \tag{2.7}$$

Let $K = E/(1 - E)$ be a constant depending on the efficiency to be maintained. Since T_o is a function of W and p, Equation 2.7 can be rewritten as

$$W = K T_o(W, p). \tag{2.8}$$

From Equation 2.8, the problem size W can usually be obtained as a function of p by algebraic manipulations. This function dictates the growth rate of W required to keep the efficiency fixed as p increases. We call this function the *isoefficiency function* of the parallel system. The isoefficiency function determines the ease with which a parallel system can maintain a constant efficiency and hence achieve speedups increasing in proportion to the number of processors. A small isoefficiency function means that small increments in the problem size are sufficient for the efficient utilization of an increasing number of processors, indicating that the parallel system is highly scalable. However, a large isoefficiency function indicates a poorly scalable parallel system. The isoefficiency function does not exist for unscalable parallel systems, because in such systems the efficiency cannot be kept at any constant value as p increases, no matter how fast the problem size is increased.

In the simple example of adding n numbers, the overhead due to communication (hereafter referred to as the *communication overhead*) is a function of p only. In general, communication overhead can depend on both the problem size and the number of processors. A typical overhead function can have several distinct terms of different orders of magnitude with respect to p and W. In such a case, it can be cumbersome (or even impossible) to obtain the isoefficiency function as a closed function of p. For example, consider a hypothetical parallel system for which $T_o = p^{3/2} + p^{3/4}W^{3/4}$. For this overhead function, Equation 2.8 can be rewritten as $W = Kp^{3/2} + Kp^{3/4}W^{3/4}$. It is hard to solve this equation for W in terms of p.

Recall that the condition for constant efficiency is that the ratio T_o/W remains fixed. As p and W increase, the efficiency is nondecreasing as long as none of the terms of T_o grows faster than W. If T_o has multiple terms, we balance W against each term of T_o and compute the respective isoefficiency functions for individual terms. The component of T_o that requires the problem size to grow at the highest rate with respect to p determines the overall asymptotic isoefficiency function of the parallel system. Example 2.1 and Section **??** further illustrate the technique of isoefficiency analysis in detail.

Example 2.1 Isoefficiency Function of a Parallel System with a Complex Overhead Function

Consider a parallel system for which $T_o = p^{3/2} + p^{3/4}W^{3/4}$. Using only the first term of T_o in Equation 2.8, we get

$$W = Kp^{3/2}. \tag{2.9}$$

Using only the second term, Equation 2.8 yields the following relation between W and p:

$$\begin{aligned} W &= Kp^{3/4}W^{3/4} \\ W^{1/4} &= Kp^{3/4} \\ W &= K^4p^3 \end{aligned} \tag{2.10}$$

To ensure that the efficiency does not decrease as the number of processors increases, the first and second terms of the overhead function require the problem size to grow as $\Theta(p^{3/2})$ and $\Theta(p^3)$, respectively. The asymptotically higher of the two rates, $\Theta(p^3)$, gives the overall asymptotic isoefficiency function of this parallel system, since it subsumes the rate dictated by the other term. ■

In a single expression, the isoefficiency function captures the characteristics of a parallel algorithm as well as the parallel architecture on which it is implemented. After performing the isoefficiency analysis, we can test the performance of a parallel program on a few processors and then predict its performance on a larger number of processors. However, the utility of isoefficiency analysis is not limited to predicting the impact on performance of an increasing number of processors. Section 2.3.3 shows how the isoefficiency function characterizes the amount of parallelism inherent in a parallel algorithm. We will see in later chapters (Chapter **??**, for example) that isoefficiency analysis can be used also to study the behavior of a parallel system with respect to changes in hardware parameters such as the speed of processors and communication channels.

2.3 Asymptotic Analysis of Parallel Programs

At this point, we have accumulated an arsenal of powerful tools for quantifying the performance and scalability of an algorithm. Let us illustrate the use of these tools for evaluating a set of parallel programs for solving a given problem. Often, we ignore constants and concern ourselves with the asymptotic behaviour or quantities. In many cases, this can yield a clearer picture of relative merits and demerits of various parallel programs.

Consider the problem of sorting a list of n numbers. The fastest serial programs for this problem run in time $O(n \log n)$. Let us look at four different parallel algorithms A1, A2, A3, and A4, for sorting a given list. The parallel runtime of the four algorithms along with the number of processors they can use is given in Table 2.1.

The objective of this exercise is to determine which of these four algorithms is the best. Perhaps the simplest metric is one of speed; the algorithm with the lowest T_P is the best. By this metric, algorithm A1 is the best, followed by A3, A4, and A2. This is also reflected in the fact that the speedups of the set of algorithms are also in this order.

However, in practical situations, we will rarely have n^2 processors as are required by algorithm A1. Furthermore, resource utilization is an important aspect of practical program design. So let us look at how efficient each of these algorithms are. This metric of evalu-

Table 2.1 Comparison of four different algorithms for sorting a given list of numbers. Table shows number of processors, parallel runtime, speedup, efficiency and the pT_P product.

Algorithm	A1	A2	A3	A4
p	n^2	$\log n$	n	$\sqrt{n}$
T_P	1	n	$\sqrt{n}$	$\sqrt{n} \log n$
S	$n \log n$	$\log n$	$\sqrt{n} \log n$	$\sqrt{n}$
E	$\frac{\log n}{n}$	1	$\frac{\log n}{\sqrt{n}}$	1
pT_P	n^2	$n \log n$	$n^{1.5}$	$n \log n$

ating the algorithm presents a starkly different image. Algorithms A2 and A4 are the best, followed by A3 and A1. Since efficiency and cost are realistic measures, let us formalize the notion of *cost* of a parallel algorithm.

We define the *cost* of solving a problem on a parallel system as the product of parallel run time and the number of processors used. Cost reflects the sum of the time that each processor spends solving the problem. The cost of solving a problem on a single processor is the execution time of the fastest known sequential algorithm. A parallel system is said to be *cost-optimal* if the cost of solving a problem on a parallel computer is proportional to the execution time of the fastest-known sequential algorithm on a single processor. Since efficiency is the ratio of sequential cost to parallel cost, a cost-optimal parallel system has an efficiency of $\Theta(1)$.

Cost is sometimes referred to as **work** or *processor-time product*, and a cost-optimal system is also known as a pT_P-optimal system. The last row of Table 2.1 presents the cost of the four algorithms. From this row, it is easy to see that the costs of algorithms A1 and A3 are higher than the serial runtime of $n \log n$ and therefore neither of these algorithms is cost optimal. However, algorithms A2 and A4 are cost optimal.

Cost optimality is a very important practical concept although it is defined in terms of asymptotics. To illustrate this, let us consider a fifth sorting algorithm A5 which uses n processors to sort the list in $(\log n)^2$ time. The speedup and efficiency of this algorithm are given by $n/\log n$ and $1/\log n$ respectively. The pT_P product of this algorithm is $n(\log n)^2$. Therefore, this algorithm is not cost optimal but only by a factor of $\log n$. Let us consider a realistic scenario in which the number of processors p is much less than n. If there were no gains from aggregation of n subtasks into p aggregates, the runtime for p processors is given by $n(\log n)^2/p$. This follows from the fact that if n processors take $(\log n)^2$ time, then one processor would take time $n(\log n)^2$ time or better; and p processors would take time $n(\log n)^2/p$ or better. In the absence of improvements due to aggregation, the time would infact be of the order of $n(\log n)^2/p$. The corresponding speedup of this formulation is $p/\log n$. Consider the problem of sorting 1024 numbers ($n = 1024$, $\log n = 10$) on 32 processors. The speedup expected is only $p/\log n$ or 3.2. This number gets worse as n increases. For $n = 1\text{M}$, $\log n = 20$ and the speedup is only 1.6. Clearly, there is a very significant cost associated with not being cost-optimal even by a very small factor. (Note that a factor of $\log p$ is smaller than even $\sqrt{p}$.) This emphasizes the practical importance of cost-optimality.

2.3.1 Cost-Optimality and the Isoefficiency Function

In Section **??**, we stated that 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 fastest known sequential algorithm on a single processor. In other words, a parallel system is cost-optimal if and only if

$$pT_P = \Theta(W). \tag{2.11}$$

Substituting the expression for T_P from the right-hand side of Equation 2.4, we get the following:

$$
\begin{aligned}
W + T_o(W, p) &= \Theta(W) \\
T_o(W, p) &= O(W) && (2.12) \\
W &= \Omega(T_o(W, p)) && (2.13)
\end{aligned}
$$

Equations 2.12 and 2.13 suggest that a parallel system is cost-optimal if and only if its overhead function does not asymptotically exceed the problem size. This is very similar to the condition given by Equation 2.8 for maintaining a fixed efficiency while increasing the number of processors in a parallel system. If Equation 2.8 yields an isoefficiency function $f(p)$, then it follows from Equation 2.13 that the relation $W = \Omega(f(p))$ must be satisfied to ensure the cost-optimality of a parallel system as it is scaled up. The following example further illustrates the relationship between cost-optimality and the isoefficiency function.

Example 2.2 Relationship Between Cost-Optimality and Isoefficiency Function

Consider the non-cost-optimal parallel system described in Example **??**. Since the algorithm spends $\Theta((n/p)\log p)$ time in communication, its overhead function is $\Theta(n \log p)$. The problem size W for adding n numbers is $\Theta(n)$. The isoefficiency function for this parallel system does not exist, because Equation 2.8 cannot be satisfied for any K (hence, any E); T_o is always greater than W. Thus, this parallel system is unscalable.

Now consider the cost-optimal solution to the problem of adding n numbers on a hypercube presented in Example **??**. For this parallel system, $W \approx n$, and $T_o = \Theta(p \log p)$. From Equation 2.8, its isoefficiency function is $\Theta(p \log p)$; that is, the problem size must increase as $\Theta(p \log p)$ to maintain a constant efficiency. In Example **??** we derived the condition for cost-optimality as $W = \Omega(p \log p)$.

■

2.3.2 A Lower Bound on the Isoefficiency Function

We discussed earlier that a smaller isoefficiency function indicates higher scalability. Accordingly, an ideally-scalable parallel system must have the lowest possible isoefficiency function. For a problem consisting of W units of work, no more than W processors can be used cost-optimally; additional processors will be idle. 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 for an ideal parallel system with no communication or other overhead, the efficiency will drop because processors added beyond $p = W$ will be idle. Thus, asymptotically, the problem size must increase at least as fast as $\Theta(p)$ to maintain fixed efficiency; hence, $\Omega(p)$ is the asymptotic lower bound on the isoefficiency function. It follows that the isoefficiency function of an ideally scalable parallel system is $\Theta(p)$.

2.3.3 The Degree of Concurrency and the Isoefficiency Function

A lower bound of $\Omega(p)$ is imposed on the isoefficiency function of a parallel system by the number of operations that can be performed concurrently. The maximum number of tasks that can be executed simultaneously at any time in a parallel algorithm is called its *degree of concurrency*. The degree of concurrency is a measure of the number of operations that an algorithm can perform in parallel for a problem of size W; it is independent of the parallel architecture. If $C(W)$ is the degree of concurrency of a parallel algorithm, then for a problem of size W, no more than $C(W)$ processors can be employed effectively.

Example 2.3 Effect of Concurrency on Isoefficiency Function
Consider solving a system of n equations in n variables by using Gaussian elimination (Section **??**). The total amount of computation is $\Theta(n^3)$. But the n variables must 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 any time. Since $W = \Theta(n^3)$ for this problem, the degree of concurrency $C(W)$ is $\Theta(W^{2/3})$ and at most $\Theta(W^{2/3})$ processors can be used efficiently. On the other hand, given p processors, the problem size should be at least $\Omega(p^{3/2})$ to use them all. Thus, the isoefficiency function of this computation due to concurrency is $\Theta(p^{3/2})$.

■

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

2.4 Other Scalability Metrics and Bibliographic Remarks

To use today's massively parallel computers effectively, larger problems must be solved as more processors are added. However, when the problem size is fixed, the objective is to attain the best compromise between efficiency and parallel run time. Performance issues for fixed-size problems have been addressed by several researchers [**?**, **?**, **?**, **?**, **?**, **?**]. In most situations, additional computing power derived from increasing the number of processors can be used to solve bigger problems. In some situations, however, different ways of increasing the problem size may apply, and a variety of constraints may guide the scaling up of the workload with respect to the number of processors [**?**].

For many problem domains, it is appropriate to increase the problem size with the number of processors so that the total parallel execution time remains fixed. An example is the domain of weather forecasting, in which the size of the problem can be increased arbitrarily provided that it is solved within a specified time (it does not make sense to take more than 24 hours to forecast the next day's weather). This is known as *time-constrained scaling* and has been explored by Gustafson et al. [**?**, **?**, **?**], Sun and Ni [**?**], and Worley [**?**] (Problem **??**).

Another method for scaling problem size is to solve the largest problem that fits the available memory. This is called *memory-constrained scaling* and has been investigated by Gustafson et al. [**?**, **?**], Sun and Ni [**?**, **?**], and Worley [**?**, **?**, **?**]. Since the total memory of a parallel computer increases with the number of processors, it is possible to solve bigger problems on parallel computers with a larger p. Any problem whose memory requirement exceeds the total available memory cannot be solved on the system.

An important scenario is one in which we want to make the most efficient use of the parallel system; in other words, we want the overall performance of the parallel system to increase linearly with p. This is possible only for scalable parallel systems, which are exactly those for which a fixed efficiency can be maintained for arbitrarily large p by simply increasing the problem size. For such systems, it is natural to use the isoefficiency function or related metrics [**?**, **?**, **?**, **?**]. Isoefficiency analysis has been found to be very useful in characterizing the scalability of a variety of parallel algorithms [**?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**]. Gupta and Kumar [**?**, **?**] have demonstrated the relevance of the isoefficiency function in the fixed time case as well. They have shown that if the isoefficiency function is greater than $\Theta(p)$, then the problem size cannot be increased indefinitely while maintaining a fixed execution time, no matter how many processors are used. A number of other researchers have analyzed the performance of parallel systems with concern for overall efficiency [**?**, **?**, **?**, **?**, **?**, **?**, **?**].

Kruskal, Rudolph, and Snir [**?**] define the concept of *parallel efficient (PE)* problems. Their definition is related to the concept of isoefficiency function. Problems in the class PE have algorithms with a polynomial isoefficiency function at some efficiency. The class PE makes an important distinction between algorithms with polynomial isoefficiency functions and those with worse isoefficiency functions. Kruskal et al. proved the invariance of the class PE over a variety of parallel computational models and interconnection schemes. An important consequence of this result is that an algorithm with a polynomial isoefficiency on one architecture will have a polynomial isoefficiency on many other architectures as well. There can be exceptions, however; for instance, Gupta and Kumar [**?**] show that the fast Fourier transform algorithm has a polynomial isoefficiency on a hypercube but an exponential isoefficiency on a mesh.

Vitter and Simons [**?**] define a class of problems called *PC**. PC* includes problems with efficient parallel algorithms on a PRAM. A problem in class P (the polynomial-time class) is in PC* if it has a parallel algorithm on a PRAM that can use a polynomial (in terms of input size) number of processors and achieve a minimal efficiency ϵ. Any problem in PC* has at least one parallel algorithm such that, for an efficiency ϵ, its isoefficiency function exists and is a polynomial.

A discussion of various scalability and performance measures can be found in the survey by Kumar and Gupta [**?**]. Besides the ones cited so far, a number of other metrics of performance and scalability of parallel systems have been proposed [**?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**].

Flatt and Kennedy [**?**, **?**] show that if the overhead function satisfies certain mathematical properties, then there exists a unique value p_0 of the number of processors for which T_P is minimum for a given W. A property of T_o on which their analysis depends heavily is that $T_o > \Theta(p)$. Gupta and Kumar [**?**] show that there exist parallel systems that do not obey this condition, and in such cases the point of peak performance is determined by the degree of concurrency of the algorithm being used.

Marinescu and Rice [**?**] develop a model to describe and analyze a parallel computation on an MIMD computer in terms of the number of threads of control p into which the computation is divided and the number events $g(p)$ as a function of p. They consider the case where each event is of a fixed duration θ and hence $T_o = \theta g(p)$. Under these assumptions on T_o, they conclude that with increasing number of processors, the speedup saturates at some value if $T_o = \Theta(p)$, and it asymptotically approaches zero if $T_o = \Theta(p^m)$, where $m \geq 2$. Gupta and Kumar [**?**] generalize these results for a wider class of overhead functions. They show that the speedup saturates at some maximum value if $T_o \leq \Theta(p)$, and the speedup attains a maximum value and then drops monotonically with p if $T_o > \Theta(p)$.

Eager et al. [**?**] and Tang and Li [**?**] have proposed a criterion of optimality of a parallel system so that a balance is struck between efficiency and speedup. They propose that a good choice of operating point on the execution time verses efficiency curve is that where the incremental benefit of adding processors is roughly $1/2$ per processor or, in other words, efficiency is 0.5. They conclude that for $T_o = \Theta(p)$, this is also equivalent to operating at a point where the ES product is maximum or $p(T_P)^2$ is minimum. This conclusion is a special case of the more general case presented by Gupta and Kumar [**?**].

Belkhale and Banerjee [**?**], Leuze et al. [**?**], Ma and Shea [**?**], and Park and Dowdy [**?**] address the important problem of optimal partitioning of the processors of a parallel computer among several applications of different scalabilities executing simultaneously.