UNIT - IV

4. Analytical Modelling of Parallel Program

Using twice as many hardware resources, one can reasonably expect a program to run twice as fast. However, in typical parallel programs, this is rarely the case, due to a variety of overheads associated with parallelism. An accurate quantification of these overheads is critical to the understanding of parallel program performance.

A typical execution profile of a parallel program is illustrated in Figure 5.1. In addition to performing essential computation (i.e., computation that would be performed by the serial program for solving the same problem instance), a parallel program may also spend time in interprocess communication, idling, and excess computation (computation not performed by the serial formulation).

**Figure 5.1.** The execution profile of a hypothetical parallel program executing on eight processing elements. Profile indicates times spent performing computation (both essential and excess), communication, and idling.

Interprocess Interaction Any nontrivial parallel system requires its processing elements to interact and communicate data (e.g., intermediate results). The time spent communicating data between processing elements is usually the most significant source of parallel processing overhead.

Idling Processing elements in a parallel system may become idle due to many reasons such as load imbalance, synchronization, and presence of serial components in a program. In many parallel applications (for example, when task generation is dynamic), it is impossible (or at least difficult) to predict the size of the subtasks assigned to various processing elements. Hence, the problem cannot be subdivided statically among the processing elements while maintaining uniform workload. If different processing elements have different workloads, some processing elements may be idle during part of the time that others are working on the problem. In some parallel programs, processing elements must synchronize at certain points during parallel program execution. If all processing elements 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. Parts of an algorithm may be unparallelizable, allowing only a single processing element to work on it. While one processing element works on the serial part, all the other processing elements must wait.
**Excess 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 is the excess 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 a computation is the 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 processing elements. Therefore, some computations are performed multiple times on different processing elements. Chapter 13 discusses these algorithms in detail.

Since different parallel algorithms for solving the same problem incur varying overheads, it is important to quantify these overheads with a view to establishing a figure of merit for each algorithm.

### 5.2 Performance Metrics for Parallel Systems

It is important to study the performance of parallel programs with a view to determining the best algorithm, evaluating hardware platforms, and examining the benefits from parallelism. A number of metrics have been used based on the desired outcome of performance analysis.

#### 5.2.1 Execution Time

The serial runtime of a program is the time elapsed between the beginning and the end of its execution on a sequential computer. The **parallel runtime** is the time that elapses from the moment a parallel computation starts to the moment the last processing element finishes execution. We denote the serial runtime by $T_S$ and the parallel runtime by $T_P$.

#### 5.2.2 Total Parallel Overhead

The overheads incurred by a parallel program are encapsulated into a single expression referred to as the **overhead function**. We define overhead function or total overhead of a parallel system as the total time collectively spent by all the processing elements over and above that required by the fastest known sequential algorithm for solving the same problem on a single processing element. We denote the overhead function of a parallel system by the symbol $T_o$.

The total time spent in solving a problem summed over all processing elements is $pT_P + T_S$ units of this time are spent performing useful work, and the remainder is overhead. Therefore, the overhead function ($T_o$) is given by

**Equation 5.1**

$$T_o = pT_P - T_S.$$
5.2.3 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 processing element to the time required to solve the same problem on a parallel computer with \( p \) identical processing elements. We denote speedup by the symbol \( S \).

**Example 5.1 Adding \( n \) numbers using \( n \) processing elements**

Consider the problem of adding \( n \) numbers by using \( n \) processing elements. Initially, each processing element is assigned one of the numbers to be added and, at the end of the computation, one of the processing elements stores the sum of all the numbers. Assuming that \( n \) is a power of two, we can perform this operation in \( \log n \) steps by propagating partial sums up a logical binary tree of processing elements. **Figure 5.2** illustrates the procedure for \( n = 16 \). The processing elements are labeled from 0 to 15. Similarly, the 16 numbers to be added are labeled from 0 to 15. The sum of the numbers with consecutive labels from \( i \) to \( j \) is denoted by \( \Sigma_i^j \).

**Figure 5.2. Computing the globalsum of 16 partial sums using 16 processing elements.** \( \Sigma_i^j \) denotes the sum of numbers with consecutive labels from \( i \) to \( j \).
Each step shown in Figure 5.2 consists of one addition and the communication of a single word. The addition can be performed in some constant time, say $t_c$, and the communication of a single word can be performed in time $t_s + t_w$. Therefore, the addition and communication operations take a constant amount of time. Thus,

**Equation 5.2**

$$T_p = \Theta(\log n).$$

Since the problem can be solved in $O(n)$ time on a single processing element, its speedup is

**Equation 5.3**

$$S = \Theta\left(\frac{n}{\log n}\right).$$

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 processing element. Sometimes, the asymptotically fastest sequential algorithm to solve a problem is not known, or its runtime has a large constant that makes it impractical to implement. In such cases, we 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 runtime of the best sequential algorithm for solving a problem to the time taken by the parallel algorithm to solve the same problem on $p$ processing elements. The $p$ processing elements used by the parallel algorithm are assumed to be identical to the one used by the sequential algorithm.

**Example 5.2 Computing speedups of parallel programs**

Consider the example of parallelizing bubble sort (Section 9.3.1). Assume that a serial version of bubble sort of $10^5$ records takes 150 seconds and a serial quicksort can sort the same list in 30 seconds. If a parallel version of bubble sort, also called odd-even sort, takes 40 seconds on four processing elements, it would appear that the parallel odd-even sort algorithm results in a speedup of $150/40$ or 3.75. However, this conclusion is misleading, as in reality the parallel algorithm results in a speedup of $30/40$ or 0.75 with respect to the best serial algorithm.

Theoretically, speedup can never exceed the number of processing elements, $p$. If the best sequential algorithm takes $T_S$ units of time to solve a given problem on a single processing element, then a speedup of $p$ can be obtained on $p$ processing elements if none of the processing elements spends more than time $T_S/p$. A speedup greater than $p$ is possible only if each processing element spends less than time $T_S/p$ solving the problem. In this case, a single processing element could emulate the $p$ processing elements 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 runtime of the algorithm, then the problem cannot be solved in less than time $T_S$ on a single processing element.

In practice, a speedup greater than $p$ is sometimes observed (a phenomenon known as superlinear speedup). This usually happens when the work performed by a serial algorithm is greater than its parallel formulation or due to hardware features that put the serial implementation at a disadvantage. For example, the data for a problem might be too large to fit into the cache of a single processing element, thereby degrading its performance due to the use of slower memory elements. But when partitioned among several processing elements, the individual data-partitions would be small enough to fit into their respective processing
elements' caches. In the remainder of this book, we disregard superlinear speedup due to hierarchical memory.

**Example 5.3 Superlinearity effects from caches**

Consider the execution of a parallel program on a two-processor parallel system. The program attempts to solve a problem instance of size $W$. With this size and available cache of 64 KB on one processor, the program has a cache hit rate of 80%. Assuming the latency to cache of 2 ns and latency to DRAM of 100 ns, the effective memory access time is $2 \times 0.8 + 100 \times 0.2$, or 21.6 ns. If the computation is memory bound and performs one FLOP/memory access, this corresponds to a processing rate of 46.3 MFLOPS. Now consider a situation when each of the two processors is effectively executing half of the problem instance (i.e., size $W/2$). At this problem size, the cache hit ratio is expected to be higher, since the effective problem size is smaller. Let us assume that the cache hit ratio is 90%, 8% of the remaining data comes from local DRAM, and the other 2% comes from the remote DRAM (communication overhead). Assuming that remote data access takes 400 ns, this corresponds to an overall access time of $2 \times 0.9 + 100 \times 0.08 + 400 \times 0.02$, or 17.8 ns. The corresponding execution rate at each processor is therefore 56.18, for a total execution rate of 112.36 MFLOPS. The speedup in this case is given by the increase in speed over serial formulation, i.e., $112.36/46.3$ or 2.43! Here, because of increased cache hit ratio resulting from lower problem size per processor, we notice superlinear speedup.

**Example 5.4 Superlinearity effects due to exploratory decomposition**

Consider an algorithm for exploring leaf nodes of an unstructured tree. Each leaf has a label associated with it and the objective is to find a node with a specified label, in this case 'S'. Such computations are often used to solve combinatorial problems, where the label 'S' could imply the solution to the problem (Section 11.6). In Figure 5.3, we illustrate such a tree. The solution node is the rightmost leaf in the tree. A serial formulation of this problem based on depth-first tree traversal explores the entire tree, i.e., all 14 nodes. If it takes time $t_c$ to visit a node, the time for this traversal is $14t_c$. Now consider a parallel formulation in which the left subtree is explored by processing element 0 and the right subtree by processing element 1. If both processing elements explore the tree at the same speed, the parallel formulation explores only the shaded nodes before the solution is found. Notice that the total work done by the parallel algorithm is only nine node expansions, i.e., $9t_c$. The corresponding parallel time, assuming the root node expansion is serial, is $5t_c$ (one root node expansion, followed by four node expansions by each processing element). The speedup of this two-processor execution is therefore $14t_c/5t_c$, or 2.8!

**Figure 5.3.** Searching an unstructured tree for a node with a given label, 'S', on two processing elements using depth-first traversal. The two-processor version with processor 0 searching the left subtree and processor 1 searching the right subtree expands only the shaded nodes before the solution is found. The corresponding serial formulation expands the entire tree. It is clear that the serial algorithm does more work than the parallel algorithm.
The cause for this superlinearity is that the work performed by parallel and serial algorithms is different. Indeed, if the two-processor algorithm was implemented as two processes on the same processing element, the algorithmic superlinearity would disappear for this problem instance. Note that when exploratory decomposition is used, the relative amount of work performed by serial and parallel algorithms is dependent upon the location of the solution, and it is often not possible to find a serial algorithm that is optimal for all instances. Such effects are further analyzed in greater detail in Chapter 11.

5.2.4 Efficiency

Only an ideal parallel system containing p processing elements can deliver a speedup equal to p. In practice, ideal behavior is not achieved because while executing a parallel algorithm, the processing elements cannot devote 100% of their time to the computations of the algorithm. As we saw in Example 5.1, part of the time required by the processing elements to compute the sum of n numbers is spent idling (and communicating in real systems). Efficiency is a measure of the fraction of time for which a processing element is usefully employed; it is defined as the ratio of speedup to the number of processing elements. 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 effectiveness with which the processing elements are utilized. We denote efficiency by the symbol E. Mathematically, it is given by

\[ E = \frac{S}{p}. \]

Example 5.5 Efficiency of adding n numbers on n processing elements

From Equation 5.3 and the preceding definition, the efficiency of the algorithm for adding n numbers on n processing elements is

\[ E = \Theta \left( \frac{n}{\log n} \right) \]

We also illustrate the process of deriving the parallel runtime, speedup, and efficiency while preserving various constants associated with the parallel platform.

Example 5.6 Edge detection on images

Given an n x n pixel image, the problem of detecting edges corresponds to applying a 3x3 template to each pixel. The process of applying the template corresponds to multiplying pixel values with corresponding template values and summing across the template (a convolution operation). This process is illustrated in Figure 5.4(a) along with typical templates (Figure 5.4(b)). Since we have nine multiply-add operations for each pixel, if each multiply-add takes time t_c, the entire operation takes time 9t_c n^2 on a serial computer.
A simple parallel algorithm for this problem partitions the image equally across the processing elements and each processing element applies the template to its own subimage. Note that for applying the template to the boundary pixels, a processing element must get data that is assigned to the adjoining processing element. Specifically, if a processing element is assigned a vertically sliced subimage of dimension \( n \times (n/p) \), it must access a single layer of \( n \) pixels from the processing element to the left and a single layer of \( n \) pixels from the processing element to the right (note that one of these accesses is redundant for the two processing elements assigned the subimages at the extremities). This is illustrated in Figure 5.4(c).

On a message passing machine, the algorithm executes in two steps: (i) exchange a layer of \( n \) pixels with each of the two adjoining processing elements; and (ii) apply template on local subimage. The first step involves two \( n \)-word messages (assuming each pixel takes a word to communicate RGB data). This takes time \( 2(t_s + t_w n) \). The second step takes time \( 9t_c n^2/p \). The total time for the algorithm is therefore given by:

\[
T_P = \frac{9t_c n^2}{p} + 2(t_s + t_w n)
\]

The corresponding values of speedup and efficiency are given by:

\[
S = \frac{9t_c n^2}{9t_c n^2/p + 2(t_s + t_w n)}
\]

and

\[
E = \frac{1}{1 + \frac{2p(t_s + t_w n)}{9t_c n^2}}.
\]

5.2.5 Cost
We define the cost of solving a problem on a parallel system as the product of parallel runtime and the number of processing elements used. Cost reflects the sum of the time that each processing element spends solving the problem. Efficiency can also be expressed as the ratio of the execution time of the fastest known sequential algorithm for solving a problem to the cost of solving the same problem on $p$ processing elements.

The cost of solving a problem on a single processing element 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 has the same asymptotic growth (in $\Theta$ terms) as a function of the input size as the fastest-known sequential algorithm on a single processing element. 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.

**Example 5.7 Cost of adding $n$ numbers on $n$ processing elements**

The algorithm given in Example 5.1 for adding $n$ numbers on $n$ processing elements has a processor-time product of $\Theta(n \log n)$. Since the serial runtime of this operation is $\Theta(n)$, the algorithm is not cost optimal.

Cost optimality is a very important practical concept although it is defined in terms of asymptotics. We illustrate this using the following example.

**Example 5.8 Performance of non-cost optimal algorithms**

Consider a sorting algorithm that uses $n$ processing elements to sort the list in time $(\log n)^2$. Since the serial runtime of a (comparison-based) sort is $n \log n$, 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 processing elements $p$ is much less than $n$. An assignment of these $n$ tasks to $p < n$ processing elements gives us a parallel time less than $n(\log n)^2/p$. This follows from the fact that if $n$ processing elements take time $(\log n)^2$, then one processing element would take time $n(\log n)^2/p$; and $p$ processing elements would take time $n(\log n)^2/p$. The corresponding speedup of this formulation is $p/\log n$. Consider the problem of sorting 1024 numbers ($n = 1024$, $\log n = 10$) on 32 processing elements. The speedup expected is only $p/\log n$ or 3.2. This number gets worse as $n$ increases. For $n = 10^5$, $\log n = 20$ and the speedup is only 1.6. Clearly, there is a 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.

**5.3 The Effect of Granularity on Performance**

Example 5.7 illustrated an instance of an algorithm that is not cost-optimal. The algorithm discussed in this example uses as many processing elements as the number of inputs, which is excessive in terms of the number of processing elements. In practice, we assign larger pieces of input data to processing elements. This corresponds to increasing the granularity of computation on the processing elements. Using fewer than the maximum possible number of processing elements to execute a parallel algorithm is called scaling.
down a parallel system in terms of the number of processing elements. A naive way to scale down a parallel system is to design a parallel algorithm for one input element per processing element, and then use fewer processing elements to simulate a large number of processing elements. If there are \( n \) inputs and only \( p \) processing elements \((p < n)\), we can use the parallel algorithm designed for \( n \) processing elements by assuming \( n \) virtual processing elements and having each of the \( p \) physical processing elements simulate \( n/p \) virtual processing elements.

As the number of processing elements decreases by a factor of \( n/p \), the computation at each processing element increases by a factor of \( n/p \) because each processing element now performs the work of \( n/p \) processing elements. If virtual processing elements are mapped appropriately onto physical processing elements, the overall communication time does not grow by more than a factor of \( n/p \). The total parallel runtime increases, at most, by a factor of \( n/p \), and the processor-time product does not increase. Therefore, if a parallel system with \( n \) processing elements is cost-optimal, using \( p \) processing elements \((p < n)\) to simulate \( n \) processing elements preserves cost-optimality.

A drawback of this naive method of increasing computational granularity is that if a parallel system is not cost-optimal to begin with, it may still not be cost-optimal after the granularity of computation increases. This is illustrated by the following example for the problem of adding \( n \) numbers.

**Example 5.9 Adding \( n \) numbers on \( p \) processing elements**

Consider the problem of adding \( n \) numbers on \( p \) processing elements such that \( p < n \) and both \( n \) and \( p \) are powers of 2. We use the same algorithm as in Example 5.1 and simulate \( n \) processing elements on \( p \) processing elements. The steps leading to the solution are shown in Figure 5.5 for \( n = 16 \) and \( p = 4 \). Virtual processing element \( i \) is simulated by the physical processing element labeled \( i \mod p \); the numbers to be added are distributed similarly. The first \( \log p \) of the \( \log n \) steps of the original algorithm are simulated in \((n/p) \log p \) steps on \( p \) processing elements. In the remaining steps, no communication is required because the processing elements that communicate in the original algorithm are simulated by the same processing element; hence, the remaining numbers are added locally. The algorithm takes \( \Theta((n/p) \log p) \) time in the steps that require communication, after which a single processing element is left with \( n/p \) numbers to add, taking time \( \Theta(n/p) \). Thus, the overall parallel execution time of this parallel system is \( \Theta((n/p) \log p) \). Consequently, its cost is \( \Theta(n \log p) \), which is asymptotically higher than the \( \Theta(n) \) cost of adding \( n \) numbers sequentially. Therefore, the parallel system is not cost-optimal.

**Figure 5.5. Four processing elements simulating 16 processing elements to compute the sum of 16 numbers (first two steps).** \( \sum_{i}^{j} \) denotes the sum of numbers with consecutive labels from \( i \) to \( j \). Four processing elements simulating 16 processing elements to compute the sum of 16 numbers (last three steps).
HIGH PERFORMANCE COMPUTING  
BACHLOR OF ENGINEERING [BE COMP]  

Substep 1

Substep 3

(a) Four processors simulating the first communication step of 16 processors

Substep 1

Substep 3

(b) Four processors simulating the second communication step of 16 processors

Substep 1

Substep 2

(c) Simulation of the third step in two substeps

(d) Simulation of the fourth step  
(e) Final result
Example 5.1 showed that $n$ numbers can be added on an $n$-processor machine in time $\Theta((\log n)$. When using $p$ processing elements to simulate $n$ virtual processing elements ($p < n$), the expected parallel runtime is $\Theta((n/p) \log n)$. However, in Example 5.9 this task was performed in time $\Theta((n/p) \log p)$ instead. The reason is that every communication step of the original algorithm does not have to be simulated; at times, communication takes place between virtual processing elements that are simulated by the same physical processing element. For these operations, there is no associated overhead. For example, the simulation of the third and fourth steps (Figure 5.5(c) and (d)) did not require any communication. However, this reduction in communication was not enough to make the algorithm cost-optimal. Example 5.10 illustrates that the same problem (adding $n$ numbers on $p$ processing elements) can be performed cost-optimally with a smarter assignment of data to processing elements.

Example 5.10 Adding $n$ numbers cost-optimally

An alternate method for adding $n$ numbers using $p$ processing elements is illustrated in Figure 5.6 for $n = 16$ and $p = 4$. In the first step of this algorithm, each processing element locally adds its $n/p$ numbers in time $\Theta(n/p)$. Now the problem is reduced to adding the $p$ partial sums on $p$ processing elements, which can be done in time $\Theta(\log p)$ by the method described in Example 5.1. The parallel runtime of this algorithm is

Equation 5.5

$$T_p = \Theta(n/p + \log p),$$

and its cost is $\Theta(n + p \log p)$. As long as $n = \Theta(p \log p)$, the cost is $\Theta(n)$, which is the same as the serial runtime. Hence, this parallel system is cost-optimal.

Figure 5.6. A cost-optimal way of computing the sum of 16 numbers using four processing elements.
These simple examples demonstrate that the manner in which the computation is mapped onto processing elements may determine whether a parallel system is cost-optimal. Note, however, that we cannot make all non-cost-optimal systems cost-optimal by scaling down the number of processing elements.

5.4 Scalability of Parallel Systems

Very often, programs are designed and tested for smaller problems on fewer processing elements. However, the real problems these programs are intended to solve are much larger, and the machines contain larger number of processing elements. Whereas code development is simplified by using scaled-down versions of the machine and the problem, their performance and correctness (of programs) is much more difficult to establish based on scaled-down systems. In this section, we will investigate techniques for evaluating the scalability of parallel programs using analytical tools.

Example 5.11 Why is performance extrapolation so difficult?

Consider three parallel algorithms for computing an $n$-point Fast Fourier Transform (FFT) on 64 processing elements. Figure 5.7 illustrates speedup as the value of $n$ is increased to 18 K. Keeping the number of processing elements constant, at smaller values of $n$, one would infer from observed speedups that binary exchange and 3-D transpose algorithms are the best. However, as the problem is scaled up to 18 K points or more, it is evident from Figure 5.7 that the 2-D transpose algorithm yields best speedup. (These algorithms are discussed in greater detail in Chapter 13.)

Figure 5.7. A comparison of the speedups obtained by the binary-exchange, 2-D transpose and 3-D transpose algorithms on 64 processing elements with $t_c = 2$, $t_w = 4$, $t_s = 25$, and $t_h = 2$ (see Chapter 13 for details).

Similar results can be shown relating to the variation in number of processing elements as the problem size is held constant. Unfortunately, such parallel performance traces are the norm as opposed to the exception, making performance prediction based on limited observed data very difficult.

5.4.1 Scaling Characteristics of Parallel Programs
The efficiency of a parallel program can be written as:

\[ E = \frac{S}{p} = \frac{T_S}{pT_P} \]

Using the expression for parallel overhead (Equation 5.1), we can rewrite this expression as

**Equation 5.6**

\[ E = \frac{1}{1 + \frac{T_o}{T_S}}. \]

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 processing elements must be idle. This corresponds to a total overhead function of \((p - 1) \times t_{serial}\). Therefore, the total overhead function \( T_o \) grows at least linearly with \( p \). In addition, due to communication, idling, and excess computation, this function may grow superlinearly in the number of processing elements. **Equation 5.6** gives us several interesting insights into the scaling of parallel programs. First, for a given problem size (i.e. the value of \( T_S \) remains constant), as we increase the number of processing elements, \( T_o \) increases. In such a scenario, it is clear from **Equation 5.6** that the overall efficiency of the parallel program goes down. This characteristic of decreasing efficiency with increasing number of processing elements for a given problem size is common to all parallel programs.

**Example 5.12 Speedup and efficiency as functions of the number of processing elements**

Consider the problem of adding \( n \) numbers on \( p \) processing elements. We use the same algorithm as in **Example 5.10**. However, to illustrate actual speedups, we work with constants here instead of asymptotics. Assuming unit time for adding two numbers, the first phase (local summations) of the algorithm takes roughly \( \frac{n}{p} \) time. The second phase involves \( \log p \) steps with a communication and an addition at each step. If a single communication takes unit time as well, the time for this phase is \( 2 \log p \). Therefore, we can derive parallel time, speedup, and efficiency as:

**Equation 5.7**

\[ T_P = \frac{n}{p} + 2 \log p \]

**Equation 5.8**

\[ S = \frac{n}{\frac{n}{p} + 2 \log p} \]
Equation 5.9

\[
E = \frac{1}{1 + \frac{2p \log p}{n}}
\]

These expressions can be used to calculate the speedup and efficiency for any pair of \( n \) and \( p \). Figure 5.8 shows the \( S \) versus \( p \) curves for a few different values of \( n \) and \( p \). Table 5.1 shows the corresponding efficiencies.

**Figure 5.8. Speedup versus the number of processing elements for adding a list of numbers.**

![Figure 5.8](image)

Figure 5.8 and Table 5.1 illustrate that the speedup tends to saturate and efficiency drops as a consequence of Amdahl’s law (Problem 5.1). Furthermore, a larger instance of the same problem yields higher speedup and efficiency for the same number of processing elements, although both speedup and efficiency continue to drop with increasing \( p \).

Let us investigate the effect of increasing the problem size keeping the number of processing elements constant. We know that the total overhead function \( T_o \) is a function of both problem size \( T_S \) and the number of processing elements \( p \). In many cases, \( T_o \) grows sublinearly with respect to \( T_S \). In such cases, we can see that efficiency increases if the problem size is increased keeping the number of processing elements constant. For such algorithms, it should be possible to keep the efficiency fixed by increasing both the size of the problem and the number of processing elements simultaneously. For instance, in Table 5.1, the efficiency of adding 64 numbers using four processing elements is 0.80. If the number of processing elements is increased to 8 and the size of the problem is scaled up to add 192 numbers, the efficiency remains 0.80. Increasing \( p \) to 16 and \( n \) to 512 results in the same efficiency. This ability to maintain efficiency at a fixed value by simultaneously increasing the number of processing elements and the size of the problem is exhibited by many parallel systems. We call such systems *scalable* parallel systems. The *scalability* of a parallel system is a measure of its capacity to increase speedup in proportion to the number of processing elements. It reflects a
parallel system’s ability to utilize increasing processing resources effectively.

Table 5.1. Efficiency as a function of \( n \) and \( p \) for adding \( n \) numbers on \( p \) processing elements.

<table>
<thead>
<tr>
<th>( n )</th>
<th>( p = 1 )</th>
<th>( p = 4 )</th>
<th>( p = 8 )</th>
<th>( p = 16 )</th>
<th>( p = 32 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>1.0</td>
<td>0.80</td>
<td>0.57</td>
<td>0.33</td>
<td>0.17</td>
</tr>
<tr>
<td>192</td>
<td>1.0</td>
<td>0.92</td>
<td>0.80</td>
<td>0.60</td>
<td>0.38</td>
</tr>
<tr>
<td>320</td>
<td>1.0</td>
<td>0.95</td>
<td>0.87</td>
<td>0.71</td>
<td>0.50</td>
</tr>
<tr>
<td>512</td>
<td>1.0</td>
<td>0.97</td>
<td>0.91</td>
<td>0.80</td>
<td>0.62</td>
</tr>
</tbody>
</table>

Recall from Section 5.2.5 that a cost-optimal parallel system has an efficiency of \( \Theta(1) \). Therefore, scalability and cost-optimality of parallel systems are related. A scalable parallel system can always be made cost-optimal if the number of processing elements and the size of the computation are chosen appropriately. For instance, Example 5.10 shows that the parallel system for adding \( n \) numbers on \( p \) processing elements is cost-optimal when \( n = \Theta(p \log p) \). Example 5.13 shows that the same parallel system is scalable if \( n \) is increased in proportion to \( \Theta(p \log p) \) as \( p \) is increased.

Example 5.13 Scalability of adding \( n \) numbers

For the cost-optimal addition of \( n \) numbers on \( p \) processing elements \( n = \Theta(p \log p) \). As shown in Table 5.1, the efficiency is 0.80 for \( n = 64 \) and \( p = 4 \). At this point, the relation between \( n \) and \( p \) is \( n = 8p \log p \). If the number of processing elements is increased to eight, then \( 8p \log p = 192 \). Table 5.1 shows that the efficiency is indeed 0.80 with \( n = 192 \) for eight processing elements. Similarly, for \( p = 16 \), the efficiency is 0.80 for \( n = 8p \log p = 512 \). Thus, this parallel system remains cost-optimal at an efficiency of 0.80 if \( n \) is increased as \( 8p \log p \).

5.4.2 The Isoefficiency Metric of Scalability

We summarize the discussion in the section above with the following two observations:

1. For a given problem size, as we increase the number of processing elements, the overall efficiency of the parallel system goes down. This phenomenon is common to all parallel systems.
2. In many cases, the efficiency of a parallel system increases if the problem size is increased while keeping the number of processing elements constant.

These two phenomena are illustrated in Figure 5.9(a) and (b), respectively. Following from these two observations, we define a scalable parallel system as one in which the efficiency can be kept constant as the number of processing elements 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 processing elements 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 processing elements is increased. This rate determines the degree of scalability of the parallel system. As we shall show, a lower rate is more desirable than a higher growth rate in problem size. 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.

Figure 5.9. Variation of efficiency: (a) as the number of processing elements is increased for a given
Problem Size

When analyzing parallel systems, we frequently encounter the notion of the size of the problem being solved. Thus 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 processing element, where the best sequential algorithm is defined as in Section 5.2.3. 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 impact 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 runtime \( T_S \) of the fastest known algorithm to solve the problem on a sequential computer.

The Isoefficiency Function

Parallel execution time can be expressed as a function of problem size, overhead function, and the number of processing elements. We can write parallel runtime as:

\[
T_P = \frac{W + T_o(W, p)}{p}
\]
The resulting expression for speedup is

\textbf{Equation 5.11}

\[ S = \frac{W}{T_p} \]
\[ = \frac{W_p}{W + T_o(W, p)}. \]

Finally, we write the expression for efficiency as

\textbf{Equation 5.12}

\[ E = \frac{S}{p} \]
\[ = \frac{W}{W + T_o(W, p)} \]
\[ = \frac{1}{1 + T_o(W, p)/W}. \]

In \textbf{Equation 5.12}, 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 processing elements fixed, then for scalable parallel systems, the efficiency increases. This is because \( T_o \) grows slower than \( \Box(W) \) 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 processing elements 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 processing elements 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 \textbf{Equation 5.12} is maintained at a constant value. For a desired value \( E \) of efficiency,

\textbf{Equation 5.13}
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 5.13 can be rewritten as

**Equation 5.14**

\[
W = K T_o(W, p).
\]

From **Equation 5.14**, 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 processing elements. A small isoefficiency function means that small increments in the problem size are sufficient for the efficient utilization of an increasing number of processing elements, 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.

### Example 5.14 Isoefficiency function of adding numbers

The overhead function for the problem of adding \( n \) numbers on \( p \) processing elements is approximately \( 2p \log p \), as given by Equations 5.9 and 5.1. Substituting \( T_o \) by \( 2p \log p \) in **Equation 5.14**, we get

**Equation 5.15**

\[
W = K 2p \log p.
\]

Thus, the asymptotic isoefficiency function for this parallel system is \( \Theta(p \log p) \). This means that, if the number of processing elements is increased from \( p \) to \( p' \), the problem size (in this case, \( n \)) must be increased by a factor of \( (p' \log p')/(p \log p) \) to get the same efficiency as on \( p \) processing elements. In other words, increasing the number of processing elements by a factor of \( p'/p \) requires that \( n \) be increased by a factor of \( (p' \log p')/(p \log p) \) to increase the speedup by a factor of \( p'/p \).

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 processing elements. 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 5.14 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 $T_o$ grow 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 5.15 further illustrates this technique of isoefficiency analysis.

**Example 5.15 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 5.14, we get

**Equation 5.16**

$$W = Kp^{3/2}.\)**

Using only the second term, Equation 5.14 yields the following relation between $W$ and $p$:

**Equation 5.17**

$$W = Kp^{3/4} W^{3/4}\text{ (Equation 5.17)}$$

$$W^{1/4} = Kp^{3/4}\text{ (Equation 5.17)}$$

$$W = K^4 p^3\text{ (Equation 5.17)}$$

To ensure that the efficiency does not decrease as the number of processing elements 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. The reader may indeed verify that if the problem size is increased at this rate, the efficiency is $\Theta(1)$ and that any rate lower than this causes the efficiency to fall with increasing $p$. ■

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 isoefficiency analysis, we can test the performance of a parallel program on a few processing elements and then predict its performance on a larger number of processing elements. However, the utility of isoefficiency analysis is not limited to predicting the impact on performance of an increasing number of processing elements. Section 5.4.5 shows how the isoefficiency function characterizes the amount of parallelism inherent in a parallel algorithm. We will see in Chapter 13 that isoefficiency analysis can also be used to study the behavior of a parallel system with respect to changes in hardware parameters such as the speed of processing elements and communication channels. Chapter 11 illustrates how isoefficiency analysis can be used even for parallel algorithms for which...
we cannot derive a value of parallel runtime.

5.4.3 Cost-Optimality and the Isoefficiency Function

In Section 5.2.5, we stated that a parallel system is cost-optimal if the product of the number of processing elements and the parallel execution time is proportional to the execution time of the fastest known sequential algorithm on a single processing element. In other words, a parallel system is cost-optimal if and only if

Equation 5.18

\[ p T_P = \Theta(W). \]

Substituting the expression for \( T_P \) from the right-hand side of Equation 5.10, we get the following:

Equation 5.19

\[
W + T_o(W, p) = \Theta(W) \\
T_o(W, p) = O(W)
\]

Equation 5.20

\[ W = \Omega(T_o(W, p)) \]

Equations 5.19 and 5.20 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 5.14 for maintaining a fixed efficiency while increasing the number of processing elements in a parallel system. If Equation 5.14 yields an isoefficiency function \( f(p) \), then it follows from Equation 5.20 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 5.16 Relationship between cost-optimality and isoefficiency

Consider the cost-optimal solution to the problem of adding \( n \) numbers on \( p \) processing elements, presented in Example 5.10. For this parallel system, \( W \approx n \), and \( T_o = \Theta(p \log p) \). From Equation 5.14, its isoefficiency function is \( \Omega(p \log p) \); that is, the problem size must increase as \( \Omega(p \log p) \) to maintain a constant efficiency. In Example 5.10 we also derived the condition for cost-optimality as \( W = \Theta(p \log p) \).

5.4.4 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 \( W \) processing elements can be used cost-optimally; additional processing elements will be idle. If the problem size grows at a rate slower than \( \Omega(p) \) as the number of
processing elements increases, then the number of processing elements will eventually exceed $W$. Even for an ideal parallel system with no communication, or other overhead, the efficiency will drop because processing elements added beyond $p = W$ will be idle. Thus, asymptotically, the problem size must increase at least as fast as $\square(p)$ to maintain fixed efficiency; hence, $\square(p)$ is the asymptotic lower bound on the isoefficiency function. It follows that the isoefficiency function of an ideally scalable parallel system is $\square(p)$.

### 5.4.5 The Degree of Concurrency and the Isoefficiency Function

A lower bound of $\square(p)$ 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 $C(W)$ processing elements can be employed effectively.

**Example 5.17 Effect of concurrency on isoefficiency function**

Consider solving a system of $n$ equations in $n$ variables by using Gaussian elimination (Section 8.3.1). The total amount of computation is $\square(n^3)$. But then variables must be eliminated one after the other, and eliminating each variable requires $\square(n^2)$ computations. Thus, at most $\square(n^2)$ processing elements can be kept busy at any time. Since $W = \square(n^3)$ for this problem, the degree of concurrency $C(W)$ is $\square(W^{2/3})$ and at most $\square(W^{2/3})$ processing elements can be used efficiently. On the other hand, given $p$ processing elements, the problem size should be at least $\square(p^{3/2})$ to use them all. Thus, the isoefficiency function of this computation due to concurrency is $\square(p^{3/2})$.

The isoefficiency function due to concurrency is optimal (that is, $\square(p)$) only if the degree of concurrency of the parallel algorithm is $\square(W)$. If the degree of concurrency of an algorithm is less than $\square(W)$, then the isoefficiency function due to concurrency is worse (that is, greater) than $\square(p)$. 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 overheads.

### 5.5 Minimum Execution Time and Minimum Cost-Optimal Execution Time

We are often interested in knowing how fast a problem can be solved, or what the minimum possible execution time of a parallel algorithm is, provided that the number of processing elements is not a constraint. As we increase the number of processing elements for a given problem size, either the parallel runtime continues to decrease and asymptotically approaches a minimum value, or it starts rising after attaining a minimum value (Problem 5.12). We can determine the minimum parallel runtime $T_p^{\text{min}}$ for a given $W$ by differentiating the expression for $T_p$ with respect to $p$ and equating the derivative to zero (assuming that the function $T_p(W, p)$ is differentiable with respect to $p$). The number of processing elements for which $T_p$ is minimum is determined by the following equation:

**Equation 5.21**

\[
\frac{d}{dp} T_p = 0
\]
Let $p_0$ be the value of the number of processing elements that satisfies Equation 5.21. The value of $T_p^{\text{min}}$ can be determined by substituting $p_0$ for $p$ in the expression for $T_p$. In the following example, we derive the expression for $T_p^{\text{min}}$ for the problem of adding $n$ numbers.

**Example 5.18 Minimum execution time for adding $n$ numbers**

Under the assumptions of Example 5.12, the parallel run time for the problem of adding $n$ numbers on $p$ processing elements can be approximated by

**Equation 5.22**

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

Equating the derivative with respect to $p$ of the right-hand side of Equation 5.22 to zero we get the solutions for $p$ as follows:

**Equation 5.23**

$$-\frac{n}{p^2} + \frac{2}{p} = 0$$

$$-n + 2p = 0$$

$$p = \frac{n}{2}$$

Substituting $p = n/2$ in Equation 5.22, we get

**Equation 5.24**

$$T_p^{\text{min}} = 2 \log n.$$ 

In Example 5.18, the processor-time product for $p = p_0$ is $Q(n \log n)$, which is higher than the $Q(n)$ serial complexity of the problem. Hence, the parallel system is not cost-optimal for the value of $p$ that yields minimum parallel runtime. We now derive an important result that gives a lower bound on parallel runtime if the problem is solved cost-optimally.

Let $T_p^{\text{cost-opt}}$ be the minimum time in which a problem can be solved by a cost-optimal parallel system. From the discussion regarding the equivalence of cost-optimality and the isoefficiency function in Section 5.4.3, we conclude that if the isoefficiency function of a parallel system is $Q(f(p))$, then a problem of size $W$ can be solved cost-optimally if and only if $W = W(f(p))$. In other words, given a problem of size $W$, a cost-optimal solution requires that $p = O(f(W))$. Since the parallel runtime is $Q(W/p)$ for a cost-optimal parallel system (Equation 5.18), the lower bound on the parallel runtime for solving a problem of size $W$ cost-optimally is
Equation 5.25

\[ T_p^{\text{cost-opt}} = \Omega \left( \frac{W}{f^{-1}(W)} \right). \]

Example 5.19 Minimum cost-optimal execution time for adding n numbers

As derived in Example 5.14, the isoefficiency function \( f(p) \) of this parallel system is \( Q(p \log p) \).

If \( W = n = f(p) = p \log p \), then \( \log(n) = \log p + \log \log p \). Ignoring the double logarithmic term, \( \log(n) \approx \log p \).

If \( n = f(p) = p \log p \), then \( p = f^{-1}(n) = n/\log p \approx n/\log n \). Hence, \( f^{-1}(W) = Q(n/\log n) \). As a consequence of the relation between cost-optimality and the isoefficiency function, the maximum number of processing elements that can be used to solve this problem cost-optimally is \( Q(n/\log n) \). Using \( p = n/\log n \) in Equation 5.2, we get

Equation 5.26

\[ T_p^{\text{cost-opt}} = \log n + \log \left( \frac{n}{\log n} \right) \approx 2 \log n - \log \log n. \]

It is interesting to observe that both \( T_p^{\text{min}} \) and \( T_p^{\text{cost-opt}} \) for adding \( n \) numbers are \( Q(\log n) \) (Equations 5.24 and 5.26). Thus, for this problem, a cost-optimal solution is also the asymptotically fastest solution. The parallel execution time cannot be reduced asymptotically by using a value of \( p \) greater than that suggested by the isoefficiency function for a given problem size (due to the equivalence between cost-optimality and the isoefficiency function). This is not true for parallel systems in general, however, and it is quite possible that \( T_p^{\text{cost-opt}} > \Theta(T_p^{\text{min}}) \). The following example illustrates such a parallel system.

Example 5.20 A parallel system with

Consider the hypothetical parallel system of Example 5.15, for which

Equation 5.27

\[ T_o = p^{3/2} + p^{3/4} W^{3/4}. \]

From Equation 5.10, the parallel runtime for this system is

Equation 5.28

\[ T_p = \frac{W}{p} + p^{1/2} + \frac{W^{3/4}}{p^{1/4}}. \]
Using the methodology of Example 5.18,
\[
\frac{d}{dp} T_P = -\frac{W}{p^2} + \frac{1}{2p^{1/2}} - \frac{W^{3/4}}{4p^{5/4}} = 0,
\]
\[-W + \frac{1}{4} p^{3/2} - \frac{1}{4} W^{3/4} p^{3/4} = 0,
\]
\[p^{3/4} = \frac{1}{4} W^{3/4} \pm \left(\frac{1}{16} W^{3/2} + 2W\right)^{1/2}
\]
\[= \Theta(W^{3/4}),
\]
\[p = \Theta(W).
\]

From the preceding analysis, \(p_0 = Q(W)\). Substituting \(p\) by the value of \(p_0\) in Equation 5.28, we get

**Equation 5.29**

\[T_P^{\min} = \Theta(W^{1/2}).\]

According to Example 5.15, the overall isoefficiency function for this parallel system is \(Q(p^3)\), which implies that the maximum number of processing elements that can be used cost-optimally is \(Q(W^{1/3})\). Substituting \(p = Q(W^{1/3})\) in Equation 5.28, we get

**Equation 5.30**

\[T_P^{\text{cost-opt}} = \Theta(W^{2/3}).\]

A comparison of Equations 5.29 and 5.30 shows that \(T_P^{\text{cost-opt}}\) is asymptotically greater than \(T_P^{\min}\).

In this section, we have seen examples of both types of parallel systems: those for which \(T_P^{\text{cost-opt}}\) is asymptotically equal to \(T_P^{\min}\), and those for which \(T_P^{\text{cost-opt}}\) is asymptotically greater than \(T_P^{\min}\). Most parallel systems presented in this book are of the first type. Parallel systems for which the runtime can be reduced by an order of magnitude by using an asymptotically higher number of processing elements than indicated by the isoefficiency function are rare.

While deriving the minimum execution time for any parallel system, it is important to be aware that the maximum number of processing elements that can be utilized is bounded by the degree of concurrency \(C(W)\) of the parallel algorithm. It is quite possible that \(p_0\) is greater than \(C(W)\) for a parallel system (Problems 5.13 and 5.14). In such cases, the value of \(p_0\) is meaningless, and \(T_P^{\min}\) is given by
5.6 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 behavior of quantities. In many cases, this can yield a clearer picture of relative merits and demerits of various parallel programs.

Table 5.2. Comparison of four different algorithms for sorting a given list of numbers. The table shows number of processing elements, parallel runtime, speedup, efficiency and the \(pT_P\) product.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>A1</th>
<th>A2</th>
<th>A3</th>
<th>A4</th>
</tr>
</thead>
<tbody>
<tr>
<td>(p)</td>
<td>(n^2)</td>
<td>(\log n)</td>
<td>(n)</td>
<td>(\sqrt{n})</td>
</tr>
<tr>
<td>(T_P)</td>
<td>1</td>
<td>(n)</td>
<td>(\sqrt{n})</td>
<td>(\sqrt{n} \log n)</td>
</tr>
<tr>
<td>(S)</td>
<td>(n \log n)</td>
<td>(\log n)</td>
<td>(\sqrt{n} \log n)</td>
<td>(\sqrt{n})</td>
</tr>
<tr>
<td>(E)</td>
<td>(\log n)</td>
<td>1</td>
<td>(\log n)</td>
<td>1</td>
</tr>
<tr>
<td>(pT_P)</td>
<td>(n^2)</td>
<td>(n \log n)</td>
<td>(n^{1.5})</td>
<td>(n \log n)</td>
</tr>
</tbody>
</table>

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 processing elements they can use is given in Table 5.2. 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 \(n^2\) processing elements 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 evaluating the algorithm presents a starkly different image. Algorithms A2 and A4 are the best, followed by A3 and A1. The last row of Table 5.2 presents the cost of the four algorithms. From this row, it is evident 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.

This set of algorithms illustrate that it is important to first understand the objectives of parallel algorithm analysis and to use appropriate metrics. This is because use of different metrics may often result in contradictory outcomes.

5.7 Other Scalability Metrics

A number of other metrics of scalability of parallel systems have been proposed. These metrics are specifically suited to different system requirements. For example, in real time applications, the objective is to scale up a system to accomplish a task in a specified time bound. One such application is multimedia...
decompression, where MPEG streams must be decompressed at the rate of 25 frames/second. Consequently, a parallel system must decode a single frame in 40 ms (or with buffering, at an average of 1 frame in 40 ms over the buffered frames). Other such applications arise in real-time control, where a control vector must be generated in real-time. Several scalability metrics consider constraints on physical architectures. In many applications, the maximum size of a problem is constrained not by time, efficiency, or underlying models, but by the memory available on the machine. In such cases, metrics make assumptions on the growth function of available memory (with number of processing elements) and estimate how the performance of the parallel system changes with such scaling. In this section, we examine some of the related metrics and how they can be used in various parallel applications.

**Scaled Speedup** This metric is defined as the speedup obtained when the problem size is increased linearly with the number of processing elements. If the scaled-speedup curve is close to linear with respect to the number of processing elements, then the parallel system is considered scalable. This metric is related to isoefficiency if the parallel algorithm under consideration has linear or near-linear isoefficiency function. In this case the scaled-speedup metric provides results very close to those of isoefficiency analysis, and the scaled-speedup is linear or near-linear with respect to the number of processing elements. For parallel systems with much worse isoefficiencies, the results provided by the two metrics may be quite different. In this case, the scaled-speedup versus number of processing elements curve is sublinear.

Two generalized notions of scaled speedup have been examined. They differ in the methods by which the problem size is scaled up with the number of processing elements. In one method, the size of the problem is increased to fill the available memory on the parallel computer. The assumption here is that aggregate memory of the system increases with the number of processing elements. In the other method, the size of the problem grows with $p$ subject to an upper-bound on execution time.

**Example 5.21 Memory and time-constrained scaled speedup for matrix-vector products**

The serial runtime of multiplying a matrix of dimension $n \times n$ with a vector is $t_c n^2$, where $t_c$ is the time for a single multiply-add operation. The corresponding parallel runtime using a simple parallel algorithm is given by:

$$T_P = t_c \frac{n^2}{p} + t_s \log p + t_w n$$

and the speedup $S$ is given by:

**Equation 5.32**

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

The total memory requirement of the algorithm is $Q(n^2)$. Let us consider the two cases of problem scaling. In the case of memory constrained scaling, we assume that the memory of the parallel system grows linearly with the number of processing elements, i.e., $m = Q(p)$. This is a reasonable assumption for most current parallel platforms. Since $m = Q(n^2)$, we have $n^2 = c \times p$, for some constant $c$. Therefore, the scaled speedup $S'$ is given by:
\[ S' = \frac{t_c c \times p}{t_c \frac{c \times p}{p} + t_s \log p + t_w \sqrt{c \times p}} \]

or

\[ S' = \frac{c_1 p}{c_2 + c_3 \log p + c_4 \sqrt{p}}. \]

In the limiting case, \( S' = O(\sqrt{p}) \).

In the case of time constrained scaling, we have \( T_P = O(n^2/p) \). Since this is constrained to be constant, \( n^2 = O(p) \). We notice that this case is identical to the memory constrained case. This happened because the memory and runtime of the algorithm are asymptotically identical.

**Example 5.22 Memory and time-constrained scaled speedup for matrix-matrix products**

The serial runtime of multiplying two matrices of dimension \( n \times n \) is \( t_c n^3 \), where \( t_c \), as before, is the time for a single multiply-add operation. The corresponding parallel runtime using a simple parallel algorithm is given by:

\[ T_P = t_c \frac{n^3}{p} + t_s \log p + 2t_w \frac{n^2}{\sqrt{p}} \]

and the speedup \( S \) is given by:

**Equation 5.33**

\[ S = \frac{t_c n^3}{t_c \frac{n^3}{p} + t_s \log p + 2t_w \frac{n^2}{\sqrt{p}}} \]

The total memory requirement of the algorithm is \( Q(n^2) \). Let us consider the two cases of problem scaling. In the case of memory constrained scaling, as before, we assume that the memory of the parallel system grows linearly with the number of processing elements, i.e., \( m = Q(p) \). Since \( m = Q(n^2) \), we have \( n^2 = c \times p \), for some constant \( c \). Therefore, the scaled speedup \( S' \) is given by:

\[ S' = \frac{t_c (c \times p)^{1.5}}{t_c \frac{(c \times p)^{1.5}}{p} + t_s \log p + 2t_w \frac{c \times p}{\sqrt{p}}} = O(p) \]

In the case of time constrained scaling, we have \( T_P = O(n^3/p) \). Since this is constrained to be
constant, \( n^3 = O(p) \), or \( n^3 = c \times p \) (for some constant \( c \)).

Therefore, the time-constrained speedup \( S'' \) is given by:

\[
S'' = \frac{t_c c \times p}{t_c \frac{c \times p}{p} + t_s \log p + 2t_w \frac{(c \times p)^{2/3}}{\sqrt{p}}} = O(p^{5/6})
\]

This example illustrates that memory-constrained scaling yields linear speedup, whereas time-constrained speedup yields sublinear speedup in the case of matrix multiplication.

**Serial Fraction**

The experimentally determined serial fraction \( f \) can be used to quantify the performance of a parallel system on a fixed-size problem. Consider a case when the serial runtime of a computation can be divided into a totally parallel and a totally serial component, i.e.,

\[
W = T_{ser} + T_{par}.
\]

Here, \( T_{ser} \) and \( T_{par} \) correspond to totally serial and totally parallel components. From this, we can write:

\[
T_p = T_{ser} + \frac{T_{par}}{p}.
\]

Here, we have assumed that all of the other parallel overheads such as excess computation and communication are captured in the serial component \( T_{ser} \). From these equations, it follows that:

**Equation 5.34**

\[
T_p = T_{ser} + \frac{W - T_{ser}}{p}
\]

The serial fraction \( f \) of a parallel program is defined as:

\[
f = \frac{T_{ser}}{W}.
\]

Therefore, from Equation 5.34, we have:

\[
T_p = f \times W + \frac{W - f \times W}{p}
\]

\[
\frac{T_p}{W} = f + \frac{1 - f}{p}
\]

Since \( S = W/T_p \), we have
\[ \frac{1}{S} = f + \frac{1-f}{p}. \]

Solving for \( f \), we get:

**Equation 5.35**

\[ f = \frac{1/S - 1/p}{1 - 1/p}. \]

It is easy to see that smaller values of \( f \) are better since they result in higher efficiencies. If \( f \) increases with the number of processing elements, then it is considered as an indicator of rising communication overhead, and thus an indicator of poor scalability.

**Example 5.23 Serial component of the matrix-vector product**

From Equations 5.35 and 5.32, we have

**Equation 5.36**

\[ f = \frac{\frac{t_c n^2}{p} + t_s \log p + t_w n}{1 - 1/p}. \]

Simplifying the above expression, we get

\[ f = \frac{t_s p \log p + t_w n p}{t_c n^2} \times \frac{1}{p - 1} \]

\[ f \approx \frac{t_s \log p + t_w n}{t_c n^2} \]

It is useful to note that the denominator of this equation is the serial runtime of the algorithm and the numerator corresponds to the overhead in parallel execution.
Dense Matrix Algorithms

Algorithms involving matrices and vectors are applied in several numerical and non-numerical contexts. This chapter discusses some key algorithms for dense or full matrices that have no or few known usable zero entries. We deal specifically with square matrices for pedagogical reasons, but the algorithms in this chapter, wherever applicable, can easily be adapted for rectangular matrices as well.

Due to their regular structure, parallel computations involving matrices and vectors readily lend themselves to data-decomposition (Section 3.2.2). Depending on the computation at hand, the decomposition may be induced by partitioning the input, the output, or the intermediate data. Section 3.4.1 describes in detail the various schemes of partitioning matrices for parallel computation. The algorithms discussed in this chapter use one- and two-dimensional block, cyclic, and block-cyclic partitionings. For the sake of brevity, we will henceforth refer to one- and two-dimensional partitionings as 1-D and 2-D partitionings, respectively.

Another characteristic of most of the algorithms described in this chapter is that they use one task per process. As a result of a one-to-one mapping of tasks to processes, we do not usually refer to the tasks explicitly and decompose or partition the problem directly into processes.

8.1 Matrix-Vector Multiplication

This section addresses the problem of multiplying a dense \( n \times n \) matrix \( A \) with an \( n \times 1 \) vector \( x \) to yield the \( n \times 1 \) result vector \( y \). Algorithm 8.1 shows a serial algorithm for this problem. The sequential algorithm requires \( n^2 \) multiplications and additions. Assuming that a multiplication and addition pair takes unit time, the sequential run time is

Equation 8.1

\[ W = n^2. \]

At least three distinct parallel formulations of matrix-vector multiplication are possible, depending on whether rowwise 1-D, columnwise 1-D, or a 2-D partitioning is used.

Algorithm 8.1 A serial algorithm for multiplying an \( n \times n \) matrix \( A \) with an \( n \times 1 \) vector \( x \) to yield an \( n \times 1 \) product vector \( y \).

1. procedure \( \text{MAT\_VECT} \ (A, x, y) \)
2. begin
3. for \( i := 0 \) to \( n - 1 \) do
4. begin
5. \( y[i] := 0; \)
6. for \( j := 0 \) to \( n - 1 \) do
7. \( y[i] := y[i] + A[i, j] \times x[j]; \)
8. endfor;
9. end \( \text{MAT\_VECT} \)

8.1.1 Rowwise 1-D Partitioning

This section details the parallel algorithm for matrix-vector multiplication using rowwise block 1-D partitioning. The parallel algorithm for columnwise block 1-D partitioning is similar (Problem 8.2) and has a similar expression for parallel run time. Figure 8.1 describes the distribution and movement of data for matrix-vector multiplication with block 1-D partitioning.
Figure 8.1. Multiplication of an \( n \times n \) matrix with an \( n \times 1 \) vector using rowwise block 1-D partitioning.

For the one-row-per-process case, \( p = n \).

(a) Initial partitioning of the matrix and the starting vector \( x \)

(b) Distribution of the full vector among all the processes by all-to-all broadcast

(c) Entire vector distributed to each process after the broadcast

(d) Final distribution of the matrix and the result vector \( y \)

One Row Per Process

First, consider the case in which the \( n \times n \) matrix is partitioned among \( n \) processes so that each process stores one complete row of the matrix. The \( n \times 1 \) vector \( x \) is distributed such that each process owns one of its elements. The initial distribution of the matrix and the vector for rowwise block 1-D partitioning is shown in Figure 8.1(a). Process \( P_i \) initially owns \( x[i] \) and \( A[i, 0], A[i, 1], \ldots, A[i, n-1] \) and is responsible for computing \( y[i] \). Vector \( x \) is multiplied with each row of the matrix (Algorithm 8.1); hence, every process needs the entire vector. Since each process starts with only one element of \( x \), an all-to-all broadcast is required to distribute all the elements to all the processes. Figure 8.1(b) illustrates this communication step. After the vector \( x \) is distributed among the processes (Figure 8.1(c)), process \( P_i \) computes

\[
y[i] = \sum_{j=0}^{n-1} (A[i, j] \times x[j])
\]

(lines 6 and 7 of Algorithm 8.1). As Figure 8.1(d) shows, the result vector \( y \) is stored exactly the way the starting vector \( x \) was stored.

Parallel Run Time

Starting with one vector element per process, the all-to-all broadcast of the vector elements among \( n \) processes requires time \( O(n) \) on any architecture (Table 4.1). The multiplication of a single row of \( A \) with \( x \) is also performed by each process in time \( O(n) \). Thus, the entire procedure is completed by \( n \) processes in time \( O(n^2) \), resulting in a process-time product of \( O(n^3) \). The parallel algorithm is cost-optimal because the complexity of the serial algorithm is \( O(n^2) \).

Using Fewer than \( n \) Processes

Consider the case in which \( p \) processes are used such that \( p < n \), and the matrix is partitioned among the processes by using block 1-D partitioning. Each process initially stores \( n/p \) complete rows of the matrix and a portion of the vector of size \( n/p \). Since the vector \( x \) must be multiplied with each row of the matrix, every process needs the entire vector (that is, all the portions belonging to separate processes). This again requires an all-to-all broadcast as shown in Figure 8.1(b) and (c). The all-to-all broadcast takes
place among \( p \) processes and involves messages of size \( n/p \). After this communication step, each process multiplies its \( n/p \) rows with the vector \( x \) to produce \( n/p \) elements of the result vector. Figure 8.1(d) shows that the result vector \( y \) is distributed in the same format as that of the starting vector \( x \).

**Parallel Run Time** According to Table 4.1, an all-to-all broadcast of messages of size \( n/p \) among \( p \) processes takes time \( t_s \log p + t_w(n/p)(p - 1) \). For large \( p \), this can be approximated by \( t_s \log p + t_w n \). After the communication, each process spends time \( n^2/p \) multiplying its \( n/p \) rows with the vector. Thus, the parallel run time of this procedure is

**Equation 8.2**

\[
T_P = \frac{n^2}{p} + t_s \log p + t_w n.
\]

The process-time product for this parallel formulation is \( n^2 + t_s p \log p + t_w np \). The algorithm is cost-optimal for \( p = O(n) \).

**Scalability Analysis** We now derive the isoefficiency function for matrix-vector multiplication along the lines of the analysis in Section 5.4.2 by considering the terms of the overhead function one at a time. Consider the parallel run time given by Equation 8.2 for the hypercube architecture. The relation \( T_o = pT_P - W \) gives the following expression for the overhead function of matrix-vector multiplication on a hypercube with block 1-D partitioning:

**Equation 8.3**

\[
T_o = t_s p \log p + t_w np.
\]

Recall from Chapter 5 that the central relation that determines the isoefficiency function of a parallel algorithm is \( W = KT_o \) (Equation 5.14), where \( K = E/(1 - E) \) and \( E \) is the desired efficiency. Rewriting this relation for matrix-vector multiplication, first with only the \( t_s \) term of \( T_o \),

**Equation 8.4**

\[
W = K t_s p \log p.
\]

Equation 8.4 gives the isoefficiency term with respect to message startup time. Similarly, for the \( t_w \) term of the overhead function,

\[
W = K t_w np.
\]

Since \( W = n^2 \) (Equation 8.1), we derive an expression for \( W \) in terms of \( p, K, \) and \( t_w \) (that is, the isoefficiency function due to \( t_w \)) as follows:
Equation 8.5

\[ \begin{align*}
    n^2 & = Kt_wnp, \\
    n & = Kt_wp, \\
    n^2 & = K^2t_wp^2, \\
    W & = K^2t_wp^2.
\end{align*} \]

Now consider the degree of concurrency of this parallel algorithm. Using 1-D partitioning, a maximum of \( n \) processes can be used to multiply an \( n \times n \) matrix with an \( n \times 1 \) vector. In other words, \( p \) is \( O(n) \), which yields the following condition:

Equation 8.6

\[ \begin{align*}
    n & = \Omega(p), \\
    n^2 & = \Omega(p^2), \\
    W & = \Omega(p^2).
\end{align*} \]

The overall asymptotic isoeficiency function can be determined by comparing Equations 8.4, 8.5, and 8.6. Among the three, Equations 8.5 and 8.6 give the highest asymptotic rate at which the problem size must increase with the number of processes to maintain a fixed efficiency. This rate of \( Q(p^2) \) is the asymptotic isoeficiency function of the parallel matrix-vector multiplication algorithm with 1-D partitioning.

8.1.2 2-D Partitioning

This section discusses parallel matrix-vector multiplication for the case in which the matrix is distributed among the processes using a block 2-D partitioning. Figure 8.2 shows the distribution of the matrix and the distribution and movement of vectors among the processes.

Figure 8.2. Matrix-vector multiplication with block 2-D partitioning. For the one-element-per-process case, \( p = n^2 \) if the matrix size is \( n \times n \).
One Element Per Process

We start with the simple case in which an $n \times n$ matrix is partitioned among $n^2$ processes such that each process owns a single element. The $n \times 1$ vector $x$ is distributed only in the last column of $n$ processes, each of which owns one element of the vector. Since the algorithm multiplies the elements of the vector $x$ with the corresponding elements in each row of the matrix, the vector must be distributed such that the $i$th element of the vector is available to the $i$th element of each row of the matrix. The communication steps for this are shown in Figure 8.2(a) and (b). Notice the similarity of Figure 8.2 to Figure 8.1. Before the multiplication, the elements of the matrix and the vector must be in the same relative locations as in Figure 8.1(c). However, the vector communication steps differ between various partitioning strategies. With 1-D partitioning, the elements of the vector cross only the horizontal partition-boundaries (Figure 8.1), but for 2-D partitioning, the vector elements cross both horizontal and vertical partition-boundaries (Figure 8.2).

As Figure 8.2(a) shows, the first communication step for the 2-D partitioning aligns the vector $x$ along the principal diagonal of the matrix. Often, the vector is stored along the diagonal instead of the last column, in which case this step is not required. The second step copies the vector elements from each diagonal process to all the processes in the corresponding column. As Figure 8.2(b) shows, this step consists of $n$ simultaneous one-to-all broadcast operations, one in each column of processes. After these two communication steps, each process multiplies its matrix element with the corresponding element of $x$. To obtain the result vector $y$, the products computed for each row must be added, leaving the sums in the last column of processes. Figure 8.2(c) shows this step, which requires an all-to-one reduction (Section 4.1) in each row with the last process of the row as the destination. The parallel matrix-vector multiplication is complete after the reduction step.

Parallel Run Time Three basic communication operations are used in this algorithm: one-to-one communication to align the vector along the main diagonal, one-to-all broadcast of each vector element among the $n$ processes of each column, and all-to-one reduction in each row. Each of these operations takes time $O(\log n)$. Since each process performs a single multiplication in constant time, the overall parallel run time of this algorithm is $O(n)$. The cost (process-time product) is $O(n^2 \log n)$; hence, the algorithm is not cost-optimal.
Using Fewer than $n^2$ Processes

A cost-optimal parallel implementation of matrix-vector multiplication with block 2-D partitioning of the matrix can be obtained if the granularity of computation at each process is increased by using fewer than $n^2$ processes.

Consider a logical two-dimensional mesh of $p$ processes in which each process owns an $(n/\sqrt{p}) \times (n/\sqrt{p})$ block of the matrix. The vector is distributed in portions of $n/\sqrt{p}$ elements in the last process-column only. Figure 8.2 also illustrates the initial data-mapping and the various communication steps for this case. The entire vector must be distributed on each row of processes before the multiplication can be performed. First, the vector is aligned along the main diagonal. For this, each process in the rightmost column sends its $n/\sqrt{p}$ vector elements to the diagonal process in its row. Then a columnwise one-to-all broadcast of these $n/\sqrt{p}$ elements takes place. Each process then performs $n^2/p$ multiplications and locally adds the $n/\sqrt{p}$ sets of products. At the end of this step, as shown in Figure 8.2(c), each process has $n/\sqrt{p}$ partial sums that must be accumulated along each row to obtain the result vector. Hence, the last step of the algorithm is an all-to-one reduction of the $n/\sqrt{p}$ values in each row, with the rightmost process of the row as the destination.

**Parallel Run Time** The first step of sending a message of size $n/\sqrt{p}$ from the rightmost process of a row to the diagonal process (Figure 8.2(a)) takes time $t_s + t_u n/\sqrt{p}$. We can perform the columnwise one-to-all broadcast in at most time $(t_s + t_u n/\sqrt{p}) \log(\sqrt{p})$ by using the procedure described in Section 4.1.3. Ignoring the time to perform additions, the final rowwise all-to-one reduction also takes the same amount of time. Assuming that a multiplication and addition pair takes unit time, each process spends approximately $n^2/p$ time in computation. Thus, the parallel run time for this procedure is as follows:

**Equation 8.7**

$$T_p = \frac{n^2}{p} + t_s + t_u n/\sqrt{p} + \frac{(t_s + t_u n/\sqrt{p}) \log(\sqrt{p})}{(t_s + t_u n/\sqrt{p}) \log(\sqrt{p})}$$

$$\approx \frac{n^2}{p} + t_s \log p + t_u \frac{n}{\sqrt{p}} \log p$$

**Scalability Analysis** By using Equations 8.1 and 8.7, and applying the relation $T_o = p T_p - W$ (Equation 5.1), we get the following expression for the overhead function of this parallel algorithm:

**Equation 8.8**

$$T_o = t_s p \log p + t_u n \sqrt{p} \log p.$$
Equation 8.9

\[ W = K t_s p \log p. \]

Equation 8.9 gives the isoefficiency term with respect to the message startup time. We can obtain the isoefficiency function due to \( t_w \) by balancing the term \( t_w n \sqrt{p} \log p \) with the problem size \( n^2 \). Using the isoefficiency relation of Equation 5.14, we get the following:

Equation 8.10

\[
\begin{align*}
W &= n^2 = K t_w n \sqrt{p} \log p, \\
n &= K t_w \sqrt{p} \log p, \\
n^2 &= K^2 t_w^2 p \log^2 p, \\
W &= K^2 t_w^2 p \log^2 p.
\end{align*}
\]

Finally, considering that the degree of concurrency of 2-D partitioning is \( n^2 \) (that is, a maximum of \( n^2 \) processes can be used), we arrive at the following relation:

Equation 8.11

\[
\begin{align*}
p &= O(n^2), \\
n^2 &= \Omega(p), \\
W &= \Omega(p).
\end{align*}
\]

Among Equations 8.9, 8.10, and 8.11, the one with the largest right-hand side expression determines the overall isoefficiency function of this parallel algorithm. To simplify the analysis, we ignore the impact of the constants and consider only the asymptotic rate of the growth of problem size that is necessary to maintain constant efficiency. The asymptotic isoefficiency term due to \( t_w \) (Equation 8.10) clearly dominates the ones due to \( t_s \) (Equation 8.9) and due to concurrency (Equation 8.11). Therefore, the overall asymptotic isoefficiency function is given by \( Q(p \log^2 p) \).

The isoefficiency function also determines the criterion for cost-optimality (Section 5.4.3). With an isoefficiency function of \( Q(p \log^2 p) \), the maximum number of processes that can be used cost-optimally for a given problem size \( W \) is determined by the following relations:

Equation 8.12

\[
\begin{align*}
p \log^2 p &= O(n^2), \\
\log p + 2 \log \log p &= O(\log n).
\end{align*}
\]
Ignoring the lower-order terms,

$$\log p = O(\log n).$$

Substituting $\log n$ for $\log p$ in Equation 8.12,

**Equation 8.13**

$$p \log^2 n = O(n^2),$$

$$p = O\left(\frac{n^2}{\log^2 n}\right).$$

The right-hand side of Equation 8.13 gives an asymptotic upper bound on the number of processes that can be used cost-optimally for an $n \times n$ matrix-vector multiplication with a 2-D partitioning of the matrix.

**Comparison of 1-D and 2-D Partitionings**

A comparison of Equations 8.2 and 8.7 shows that matrix-vector multiplication is faster with block 2-D partitioning of the matrix than with block 1-D partitioning for the same number of processes. If the number of processes is greater than $n$, then the 1-D partitioning cannot be used. However, even if the number of processes is less than or equal to $n$, the analysis in this section suggests that 2-D partitioning is preferable.

Among the two partitioning schemes, 2-D partitioning has a better (smaller) asymptotic isoefficiency function. Thus, matrix-vector multiplication is more scalable with 2-D partitioning; that is, it is more scalable.

**Matrix-Matrix Multiplication**

This section discusses parallel algorithms for multiplying two $n \times n$ dense, square matrices $A$ and $B$ to yield the product matrix $C = A \times B$. All parallel matrix multiplication algorithms in this chapter are based on the conventional serial algorithm shown in Algorithm 8.2. If we assume that an addition and multiplication pair (line 8) takes unit time, then the sequential run time of this algorithm is $n^3$. Matrix multiplication algorithms with better asymptotic sequential complexities are available, for example Strassen's algorithm. However, for the sake of simplicity, in this book we assume that the conventional algorithm is the best available serial algorithm. Problem 8.5 explores the performance of parallel matrix multiplication regarding Strassen's method as the base algorithm.

**Algorithm 8.2 The conventional serial algorithm for multiplication of two $n \times n$ matrices.**

1. procedure MAT_MULT ($A$, $B$, $C$)
2. begin
3. for $i := 0$ to $n - 1$ do
4. for $j := 0$ to $n - 1$ do
5. begin
6. $C[i, j] := 0$;
7. for $k := 0$ to $n - 1$ do
Algorithm 8.3 The block matrix multiplication algorithm for $n \times n$ matrices with a block size of $(n/q) \times (n/q)$.

1. procedure BLOCK_MAT_MULT $(A, B, C)$
2. begin
3. for $i := 0$ to $q - 1$ do
4. for $j := 0$ to $q - 1$ do
5. begin
6. Initialize all elements of $C_{ij}$ to zero;
7. for $k := 0$ to $q - 1$ do
8. $C_{ij} := C_{ij} + A_{ik} \times B_{kj};$
9. endfor;
10. end BLOCK_MAT_MULT

A concept that is useful in matrix multiplication as well as in a variety of other matrix algorithms is that of block matrix operations. We can often express a matrix computation involving scalar algebraic operations on all its elements in terms of identical matrix algebraic operations on blocks or submatrices of the original matrix. Such algebraic operations on the submatrices are called block matrix operations. For example, an $n \times n$ matrix $A$ can be regarded as a $q \times q$ array of blocks $A_{ij}$ $(0 \leq i, j < q)$ such that each block is an $(n/q) \times (n/q)$ submatrix. The matrix multiplication algorithm in Algorithm 8.2 can then be rewritten as Algorithm 8.3, in which the multiplication and addition operations on line 8 are matrix multiplication and matrix addition, respectively. Not only are the final results of Algorithm 8.2 and 8.3 identical, but so are the total numbers of scalar additions and multiplications performed by each. Algorithm 8.2 performs $n^3$ additions and multiplications, and Algorithm 8.3 performs $q^3$ matrix multiplications, each involving $(n/q) \times (n/q)$ matrices and requiring $(n/q)^3$ additions and multiplications. We can use $p$ processes to implement the block version of matrix multiplication in parallel by choosing $q = \sqrt{p}$ and computing a distinct $C_{ij}$ block at each process.

In the following sections, we describe a few ways of parallelizing Algorithm 8.3. Each of the following parallel matrix multiplication algorithms uses a block 2-D partitioning of the matrices.

8.2.1 A Simple Parallel Algorithm

Consider two $n \times n$ matrices $A$ and $B$ partitioned into $p$ blocks $A_{ij}$ and $B_{ij}$ $0 \leq k < \sqrt{p}$ of size $(n/\sqrt{p}) \times (n/\sqrt{p})$ each. These blocks are mapped onto a $\sqrt{p} \times \sqrt{p}$ logical mesh of processes. The processes are labeled from $P_{0,0}$ to $P_{\sqrt{p}-1, \sqrt{p}-1}$. Process $P_{i,j}$ initially stores $A_{i,j}$ and $B_{i,j}$ and computes block $C_{i,j}$ of the result matrix. Computing submatrix $C_{i,j}$ requires all submatrices $A_{i,k}$ and $B_{k,j}$ for $0 \leq i, j < \sqrt{p}$. To acquire all the required blocks, an all-to-all broadcast of matrix $A$'s blocks is performed in each row of processes, and an all-to-all broadcast of matrix $B$'s blocks is performed in each column. After $P_{i,j}$ acquires $A_{i,0}, A_{i,1}, \ldots, A_{i,\sqrt{p}-1}$ and $B_{0,j}, B_{1,j}, \ldots, B_{\sqrt{p}-1,j}$, it performs the submatrix multiplication and addition step of lines 7 and 8 in Algorithm 8.3.

Performance and Scalability Analysis The algorithm requires two all-to-all broadcast steps (each consisting of $\sqrt{p}$ concurrent broadcasts in all rows and columns of the process mesh) among groups of $\sqrt{p}$ processes. The messages consist of submatrices of $n^2/p$ elements. From Table 4.1, the total communication time is $2(t_s \log(\sqrt{p}) + t_u(n^2/p)(\sqrt{p} - 1))$. After the communication step, each process computes a submatrix $C_{i,j}$, which requires $\sqrt{p}$ multiplications of $(n/\sqrt{p}) \times (n/\sqrt{p})$ submatrices (lines 7 and 8 of Algorithm 8.3 with $q = \sqrt{p}$). This takes a total of time $\sqrt{p} \times (n/\sqrt{p})^3 = n^3/p$. Thus, the parallel run time is approximately

Equation 8.14
The process-time product is \( T_p = \frac{n^3}{p} + t_s \log p + 2t_w \frac{n^2}{\sqrt{p}} \), and the parallel algorithm is cost-optimal for \( p = O(n^2) \).

The isoefficiency functions due to \( t_s \) and \( t_w \) are \( t_s p \log p \) and \( 8(t_w) p^{3/2} \), respectively. Hence, the overall isoefficiency function due to the communication overhead is \( Q(p^{3/2}) \). This algorithm can use a maximum of \( n^2 \) processes; hence, \( p \leq n^2 \) or \( n^3 \geq p^{3/2} \).

Therefore, the isoefficiency function due to concurrency is also \( Q(p^{3/2}) \).

A notable drawback of this algorithm is its excessive memory requirements. At the end of the communication phase, each process has \( \sqrt{p} \) blocks of both matrices \( A \) and \( B \). Since each block requires \( Q(n^2/p) \) memory, each process requires \( Q(n^2/\sqrt{p}) \) memory. The total memory requirement over all the processes is \( Q(n^2/\sqrt{p}) \), which is \( \sqrt{p} \) times the memory requirement of the sequential algorithm.

### 8.2.2 Cannon's Algorithm

Cannon's algorithm is a memory-efficient version of the simple algorithm presented in Section 8.2.1. To study this algorithm, we again partition matrices \( A \) and \( B \) into \( p \) square blocks. We label the processes from \( P_{ij} \) to \( P_{\sqrt{p} - 1, \sqrt{p} - 1} \), and initially assign submatrices \( A_{ij} \) and \( B_{ij} \) to process \( P_{ij} \). Although every process in the \( i \) th row requires all \( \sqrt{p} \) submatrices \( A_{ik} \) (\( 0 \leq k < \sqrt{p} \)), it is possible to schedule the computations of the \( \sqrt{p} \) processes of the \( i \)th row such that, at any given time, each process is using a different \( A_{ik} \). These blocks can be systematically rotated among the processes after every submatrix multiplication so that every process gets a fresh \( A_{ik} \) after each rotation. If an identical schedule is applied to the columns, then no process holds more than one block of each matrix at any time, and the total memory requirement of the algorithm over all the processes is \( Q(n^2) \). Cannon's algorithm is based on this idea. The scheduling for the multiplication of submatrices on separate processes in Cannon's algorithm is illustrated in Figure 8.3 for 16 processes.

**Figure 8.3. The communication steps in Cannon's algorithm on 16 processes.**
The first communication step of the algorithm aligns the blocks of \( A \) and \( B \) in such a way that each process multiplies its local submatrices. As Figure 8.3(a) shows, this alignment is achieved for matrix \( A \) by shifting all submatrices \( A_{ij} \) to the left (with wraparound) by \( i \) steps. Similarly, as shown in Figure 8.3(b), all submatrices \( B_{ij} \) are shifted up (with wraparound) by \( j \) steps.

These are circular shift operations (Section 4.6) in each row and column of processes, which leave process \( P_{ij} \) with submatrices 

\[
\begin{align*}
A_{0,0} & \sim A_{0,1} \sim A_{0,2} \sim A_{0,3} \\
B_{0,0} & \sim B_{1,1} \sim B_{2,2} \sim B_{3,3}
\end{align*}
\]

Figure 8.3(c) shows the blocks of \( A \) and \( B \) after the initial alignment, when each process is ready for the first submatrix multiplication. After a submatrix multiplication step, each block of \( A \) moves one step left and each block of \( B \) moves one step up (again with wraparound), as shown in Figure 8.3(d). A sequence of \( \sqrt{p} \) such submatrix multiplications and single-step shifts pairs up each \( A_{ij} \) and \( B_{ij} \) for \( k \) \( (0 \leq k < \sqrt{p}) \) at \( P_{ij} \). This completes the multiplication of
matrices $A$ and $B$.

**Performance Analysis** The initial alignment of the two matrices ([Figure 8.3(a) and (b)]) involves a rowwise and a columnwise circular shift. In any of these shifts, the maximum distance over which a block shifts is $\sqrt{p} - 1$. The two shift operations require a total of time $2(t_s + t_w n^2/p)$ (Table 4.1). Each of the $\sqrt{p}$ single-step shifts in the compute-and-shift phase of the algorithm takes time $t_s + t_w n^2/p$. Thus, the total communication time (for both matrices) during this phase of the algorithm is $2(t_s + t_w n^2/p)\sqrt{p}$.

For large enough $p$ on a network with sufficient bandwidth, the communication time for the initial alignment can be disregarded in comparison with the time spent in communication during the compute-and-shift phase.

Each process performs $\sqrt{p}$ multiplications of $(n/\sqrt{p}) \times (n/\sqrt{p})$ submatrices. Assuming that a multiplication and addition pair takes unit time, the total time that each process spends in computation is $n^3/p$. Thus, the approximate overall parallel run time of this algorithm is

\[ T_P = \frac{n^3}{p} + 2\sqrt{p} t_s + 2t_w \frac{n^2}{\sqrt{p}}. \]

The cost-optimality condition for Cannon’s algorithm is identical to that for the simple algorithm presented in Section 8.2.1. As in the simple algorithm, the isoefficiency function of Cannon's algorithm is $Q(p^{3/2})$.

### 8.2.3 The DNS Algorithm

The matrix multiplication algorithms presented so far use block 2-D partitioning of the input and the output matrices and use a maximum of $n^2$ processes for $n \times n$ matrices. As a result, these algorithms have a parallel run time of $W(n)$ because there are $Q(n^3)$ operations in the serial algorithm. We now present a parallel algorithm based on partitioning intermediate data that can use up to $n^3$ processes and that performs matrix multiplication in time $Q(\log n)$ by using $W(n^3/\log n)$ processes. This algorithm is known as the DNS algorithm because it is due to Dekel, Nassimi, and Sahni.

We first introduce the basic idea, without concern for inter-process communication. Assume that $n^3$ processes are available for multiplying two $n \times n$ matrices. These processes are arranged in a three-dimensional $n \times n \times n$ logical array. Since the matrix multiplication algorithm performs $n^3$ scalar multiplications, each of the $n^3$ processes is assigned a single scalar multiplication. The processes are labeled according to their location in the array, and the multiplication $A[i, k] \times B[k, j]$ is assigned to process $P_{ijk}$ ($0 \leq i, j, k < n$). After each process performs a single multiplication, the contents of $P_{ijk}$, $P_{i,j,1}$, ..., $P_{i,j,n-1}$ are added to obtain $C[i, j]$. The additions for all $C[i, j]$ can be carried out simultaneously in $\log n$ steps each. Thus, it takes one step to multiply and $\log n$ steps to add; that is, it takes time $Q(\log n)$ to multiply the $n \times n$ matrices by this algorithm.

We now describe a practical parallel implementation of matrix multiplication based on this idea. As Figure 8.4 shows, the process arrangement can be visualized as $n$ planes of $n \times n$ processes each. Each plane corresponds to a different value of $k$. Initially, as shown in [Figure 8.4(a)], the matrices are distributed among the $n^2$ processes of the plane corresponding to $k = 0$ at the base of the three-dimensional process array. Process $P_{i,j,0}$ initially owns $A[i, j]$ and $B[i, j]$.

**Figure 8.4.** The communication steps in the DNS algorithm while multiplying $4 \times 4$ matrices $A$ and $B$ on 64 processes. The shaded processes in part (c) store elements of the first row of $A$ and the shaded processes in part (d) store elements of the first column of $B$. 
The vertical column of processes $P_{ij,*}$ computes the dot product of row $A[i,\ast]$ and column $B[\ast,j]$. Therefore, rows of $A$ and columns of $B$ need to be moved appropriately so that each vertical column of processes $P_{ij,*}$ has row $A[i,\ast]$ and column $B[\ast,j]$. More precisely, process $P_{i,j,k}$ should have $A[i,k]$ and $B[k,j]$.

The communication pattern for distributing the elements of matrix $A$ among the processes is shown in Figure 8.4(a)-(c). First, each column of $A$ moves to a different plane such that the $j$th column occupies the same position in the plane corresponding to $k=j$ as it initially did in the plane corresponding to $k=0$. The distribution of $A$ after moving $A[i,j]$ from $P_{i,j,0}$ to $P_{i,j,j}$ is shown in Figure 8.4(b). Now all the columns of $A$ are replicated $n$ times in their respective planes by a parallel one-to-all broadcast along the $j$ axis. The result of this step is shown in Figure 8.4(c), in which the $n$ processes $P_{i,0,j}, P_{i,1,j},..., P_{i,n-1,j}$ receive a copy of $A[i,j]$ from $P_{i,j,j}$. At this point, each vertical column of processes $P_{i,j,*}$ has row $A[i,\ast]$. More precisely, process $P_{i,j,k}$ has $A[i,k]$.

For matrix $B$, the communication steps are similar, but the roles of $i$ and $j$ in process subscripts are switched. In the first one-to-one communication step, $B[i,j]$ is moved from $P_{i,j,0}$ to $P_{i,j,j}$. Then it is broadcast from $P_{i,j,j}$ among $P_{0,j,j}, P_{1,j,j},..., P_{n-1,j,j}$. The distribution of $B$ after this one-to-all broadcast along the $i$ axis is shown in Figure 8.4(d). At this point, each vertical column of processes $P_{i,j,*}$ has column $B[\ast,j]$. Now process $P_{i,j,k}$ has $B[k,j]$, in addition to $A[i,k]$.

After these communication steps, $A[i,k]$ and $B[k,j]$ are multiplied at $P_{i,j,k}$. Now each element $C[i,j]$ of the product matrix is
obtained by an all-to-one reduction along the $k$ axis. During this step, process $P_{i,j,0}$ accumulates the results of the multiplication from processes $P_{i,j,1}, ..., P_{i,j,n-1}$. Figure 8.4 shows this step for $C[0,0]$.

The DNS algorithm has three main communication steps: (1) moving the columns of $A$ and the rows of $B$ to their respective planes, (2) performing one-to-all broadcast along the $j$ axis for $A$ and along the $i$ axis for $B$, and (3) all-to-one reduction along the $k$ axis. All these operations are performed within groups of $n$ processes and take time $Q(\log n)$. Thus, the parallel run time for multiplying two $n \times n$ matrices using the DNS algorithm on $n^3$ processes is $Q(\log n)$.

DNS Algorithm with Fewer than $n^3$ Processes

The DNS algorithm is not cost-optimal for $n^3$ processes, since its process-time product of $Q(n^3 \log n)$ exceeds the $Q(n^3)$ sequential complexity of matrix multiplication. We now present a cost-optimal version of this algorithm that uses fewer than $n^3$ processes. Another variant of the DNS algorithm that uses fewer than $n^3$ processes is described in Problem 8.6.

Assume that the number of processes $p$ is equal to $q^3$ for some $q < n$. To implement the DNS algorithm, the two matrices are partitioned into blocks of size $(n/q) \times (n/q)$. Each matrix can thus be regarded as a $q \times q$ two-dimensional square array of blocks. The implementation of this algorithm on $q^3$ processes is very similar to that on $n^3$ processes. The only difference is that now we operate on blocks rather than on individual elements. Since $1 \leq q \leq n$, the number of processes can vary between 1 and $n^3$.

Performance Analysis

The first one-to-one communication step is performed for both $A$ and $B$, and takes time $t_s + t_w (n/q)^2$ for each matrix. The second step of one-to-all broadcast is also performed for both matrices and takes time $t_s \log q + t_w (n/q)^2 \log q$ for each matrix. The final all-to-one reduction is performed only once (for matrix $C$) and takes time $t_s \log q + t_w (n/q)^2 \log q$. The multiplication of $(n/q) \times (n/q)$ submatrices by each process takes time $(n/q)^3$. We can ignore the communication time for the first one-to-one communication step because it is much smaller than the communication time of one-to-all broadcasts and all-to-one reduction. We can also ignore the computation time for addition in the final reduction phase because it is of a smaller order of magnitude than the computation time for multiplying the submatrices. With these assumptions, we get the following approximate expression for the parallel run time of the DNS algorithm:

$$T_p \approx \left(\frac{n}{q}\right)^3 + 3t_s \log q + 3t_w \left(\frac{n}{q}\right)^2 \log q$$

Since $q = p^{1/3}$, we get

Equation 8.16

$$T_p = \frac{n^3}{p} + t_s \log p + t_w \frac{n^2}{p^{2/3}} \log p.$$  

The total cost of this algorithm is $n^3 + t_s p \log p + t_w n^2 p^{1/3} \log p$. The isoefficiency function is $Q(p(\log p)^3)$. The algorithm is cost-optimal for $n^3 = W(p(\log p)^3)$, or $p = O(n^3/(\log n)^3)$.

can deliver the same efficiency on more processes with 2-D partitioning than with 1-D partitioning.