CHAPTER

7

Performance Analysis

The highest and best form of efficiency is
the spontaneous cooperation of a free people.
Woodrow Wilson

7.1 INTRODUCTION

Being able to accurately predict the performance of a parallel algorithm you have designed can help you decide whether to actually go to the trouble of coding and debugging it. Being able to analyze the execution time exhibited by a parallel program can help you understand the barriers to higher performance and predict how much improvement can be realized by increasing the number of processors. This chapter will help you develop both of these skills.

We begin by deriving a general formula for the speedup achievable by a parallel program. We then look at well-known performance prediction formulas: Amdahl's Law, Gustafson-Barsis's Law, the Karp-Flatt metric, and the isoefficiency metric. Amdahl's Law can help you decide whether a program merits parallelization. Gustafson-Barsis's Law is a way to evaluate the performance of a parallel program. The Karp-Flatt metric can help you decide whether the principal barrier to speedup is the amount of inherently sequential code or parallel overhead. The isoefficiency metric is a way to evaluate the scalability of a parallel algorithm executing on a parallel computer. It can help you choose the design that will achieve higher performance when the number of processors increases.

7.2 SPEEDUP AND EFFICIENCY

We design and implement parallel programs in the hope that they will run faster than their sequential counterparts. **Speedup** is the ratio between sequential

execution time and parallel execution time:

$$\text{Speedup} = \frac{\text{Sequential execution time}}{\text{Parallel execution time}}$$

In the case studies we have worked through, we have discovered that the operations performed by parallel algorithm can be put into three categories:

- Computations that must be performed sequentially
- Computations that can be performed in parallel
- Parallel overhead (communication operations and redundant computations)

With these categories in mind, we can produce a simple model of speedup. Let $\psi(n, p)$ denote the speedup achieved solving a problem of size n on p processors, $\sigma(n)$ denote the inherently sequential (serial) portion of the computation, $\varphi(n)$ denote the portion of the computation that can be executed in parallel, and $\kappa(n, p)$ denote the time required for parallel overhead.

A sequential program, executing on a single processor, can only perform one computation at a time. Hence it requires time $\sigma(n) + \varphi(n)$ to execute the required computations. A sequential program requires no interprocessor communications, so the expression for sequential execution time does not have the $\kappa(n, p)$ term.

Now let's consider the best possible parallel execution time. The inherently sequential portion of the computation cannot benefit from parallelization. It contributes $\sigma(n)$ to the execution time of the parallel program, no matter how many processors are available. In the best case the portion of the computation that can be executed in parallel divides up perfectly among the p processors. In this case the time needed to perform these operations is $\varphi(n)/p$. Finally, we must add in time $\kappa(n, p)$ for the interprocessor communication required for the parallel program.

We have made the optimistic assumption that the parallel portion of the computation can be divided perfectly among the processors. If this is not the case, the parallel execution time will be larger, and the speedup will be smaller. Hence actual speedup will be less than or equal to the ratio between sequential execution time and parallel execution time as we have just defined. Here, then, is our completed expression for speedup:

$$\psi(n, p) \leq \frac{\sigma(n) + \varphi(n)}{\sigma(n) + \varphi(n)/p + \kappa(n, p)}$$

Adding processors reduces the computation time (by dividing the work among more processors) but increases the communication time. At some point the communication time increase is larger than the computation time decrease (see Figure 7.1). At this point the execution time begins to increase. Since speedup is inversely proportional to execution time, the speedup curve "elbows" and begins to decline.

The **efficiency** of a parallel program is a measure of processor utilization. We define efficiency to be speedup divided by the number of processors used:

$$\text{Efficiency} = \frac{\text{Sequential execution time}}{\text{Processors used} \times \text{Parallel execution time}}$$

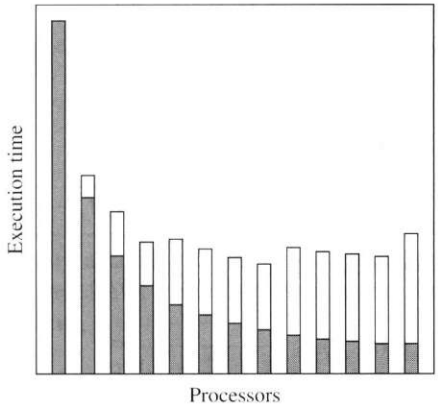

Figure 7.1 Nontrivial parallel algorithms have a computation component (black bars) that is a decreasing function of the number of processors used and a communication component (gray bars) that is an increasing function of the number of processors. For any fixed problem size there is an optimum number of processors that minimizes overall execution time.

More formally, let $\varepsilon(n, p)$ denote the efficiency of a parallel computation solving a problem of size n on p processors. Building on our earlier definition of speedup

$$\varepsilon(n, p) \leq \frac{\sigma(n) + \varphi(n)}{p\big(\sigma(n) + \varphi(n)/p + \kappa(n, p)\big)}$$

$$\Rightarrow \varepsilon(n, p) \leq \frac{\sigma(n) + \varphi(n)}{p\sigma(n) + \varphi(n) + p\kappa(n, p)}$$

Since all terms are greater than or equal to zero, $0 \leq \varepsilon(n, p) \leq 1$.

7.3 AMDAHL'S LAW

Consider the expression for speedup we have just derived.

$$\psi(n, p) \leq \frac{\sigma(n) + \varphi(n)}{\sigma(n) + \varphi(n)/p + \kappa(n, p)}$$

Since $\kappa(n, p) > 0$,

$$\psi(n, p) \leq \frac{\sigma(n) + \varphi(n)}{\sigma(n) + \varphi(n)/p + \kappa(n, p)} \leq \frac{\sigma(n) + \varphi(n)}{\sigma(n) + \varphi(n)/p}$$

Let f denote the inherently sequential portion of the computation. In other words, $f = \sigma(n)/(\sigma(n) + \varphi(n))$. Then

$$\psi(n, p) \leq \frac{\sigma(n) + \varphi(n)}{\sigma(n) + \varphi(n)/p}$$

$$\Rightarrow \psi(n, p) \leq \frac{\sigma(n)/f}{\sigma(n) + \sigma(n)(1/f - 1)/p}$$

$$\Rightarrow \psi(n, p) \leq \frac{1/f}{1 + (1/f - 1)/p}$$

$$\Rightarrow \psi(n, p) \leq \frac{1}{f + (1 - f)/p}$$

Amdahl's Law [2]
Let f be the fraction of operations in a computation that must be performed sequentially, where $0 \leq f \leq 1$. The maximum speedup ψ achievable by a parallel computer with p processors performing the computation is

$$\psi \leq \frac{1}{f + (1 - f)/p}$$

Amdahl's Law is based on the assumption that we are trying to solve a problem of fixed size as quickly as possible. It provides an upper bound on the speedup achievable by applying a certain number of processors to solve the problem in parallel. It can also be used to determine the asymptotic speedup achievable as the number of processors increases.

EXAMPLE 1

Suppose we are trying to determine whether it is worthwhile to develop a parallel version of a program solving a particular problem. Benchmarking reveals that 90 percent of the execution time is spent inside functions that we believe we can execute in parallel. The remaining 10 percent of the execution time is spent in functions that must be executed on a single processor. What is the maximum speedup that we could expect from a parallel version of the program executing on eight processors?

■ Solution
By Amdahl's Law

$$\psi \leq \frac{1}{0.1 + (1 - 0.1)/8} \approx 4.7$$

We should expect a speedup of 4.7 or less.

EXAMPLE 2

If 25 percent of the operations in a parallel program must be performed sequentially, what is the maximum speedup achievable?

■ **Solution**

The maximum achievable speedup is

$$\lim_{p \to \infty} \frac{1}{0.25 + (1 - 0.25)/p} = 4$$

EXAMPLE 3

Suppose we have implemented a parallel version of a sequential program with time complexity $\Theta(n^2)$, where n is the size of the dataset. Assume the time needed to input the dataset and output the result is

$$(18000 + n)\ \mu\text{sec}$$

This constitutes the sequential portion of the program. The computational portion of the program can be executed in parallel; it has execution time

$$(n^2/100)\ \mu\text{sec}$$

What is the maximum speedup achievable by this parallel program on a problem of size 10,000?

■ **Solution**

By Amdahl's Law

$$\psi \leq \frac{(28{,}000 + 1{,}000{,}000)\ \mu\text{sec}}{(28{,}000 + 1{,}000{,}000/p)\ \mu\text{sec}}$$

The dashed line in Figure 7.2 is the upper bound on speedup derived from Amdahl's Law.

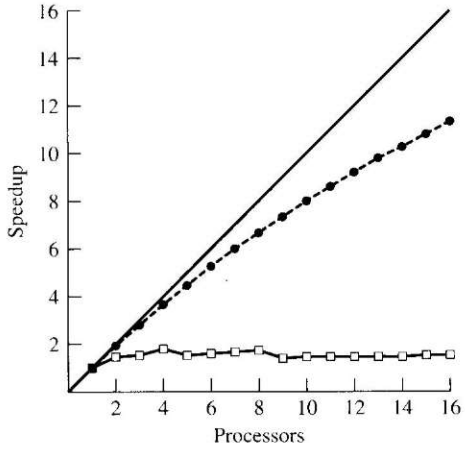

Figure 7.2 Speedup predicted by Amdahl's Law (dashed line) is higher than speedup prediction that takes communication overhead into account (solid line).

7.3.1 Limitations of Amdahl's Law

Amdahl's Law ignores overhead associated with the introduction of parallelism. Let's return to our previous example. Suppose the parallel version of the program has $\lceil \log n \rceil$ communication points. At each of these points, the communication time is

$$10{,}000\lceil \log p \rceil + (n/10)\ \mu\text{sec}$$

For a problem of size 10,000, the total communication time is

$$14(10{,}000\lceil \log p \rceil + 1{,}000)\ \mu\text{sec}$$

Now we have taken into account all of the factors included in our formula for speedup: $\sigma(n)$, $\varphi(n)$, and $\kappa(n, p)$. Our prediction for the speedup achievable by the parallel program solving a problem of size 10,000 on p processors is

$$\psi \leq \frac{(28{,}000 + 1{,}000{,}000)\ \mu\text{sec}}{(42{,}000 + 1{,}000{,}000/p + 140{,}000\lceil \log p \rceil)\ \mu\text{sec}}$$

The solid line in Figure 7.2 plots a new upper bound on speedup predicted by this more comprehensive formula. Taking communication time into account gives us a more realistic prediction of the parallel program's performance.

7.3.2 The Amdahl Effect

Typically, $\kappa(n, p)$ has lower complexity than $\varphi(n)$. That is the case with the hypothetical problem we have been considering: $\kappa(n, p) = \Theta(n \log n + n \log p)$, while $\varphi(n) = \Theta(n^2)$. Increasing the size of the problem increases the computation time faster than it increases the communication time. Hence for a fixed number of processors, speedup is usually an increasing function of the problem size. This is called the **Amdahl effect** [42]. Figure 7.3 illustrates the Amdahl effect by plotting expected speedup for our hypothetical problem. As problem size n increases, so does the height of the speedup curve.

7.4 GUSTAFSON-BARSIS'S LAW

Amdahl's Law assumes that minimizing execution time is the focus of parallel computing. It treats the problem size as a constant and demonstrates how increasing processors can reduce time.

Often, however, the goal of applying parallelism is to increase the accuracy of the solution that can be computed in a fixed amount of time. For example, an engineer studying airflow around the body of a hypersonic aircraft may want her computer to determine the solution to a problem in an hour (e.g., the length of a lunch break). If she has access to a computer with more processors, it is better for her to get a more detailed answer than to get the same results more quickly.

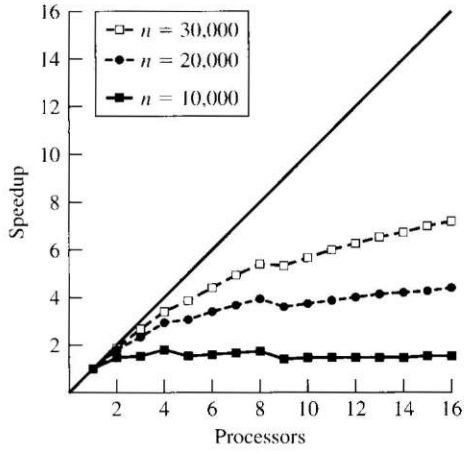

Figure 7.3 For any fixed number of processors, speedup is usually an increasing function of the problem size. This is called the **Amdahl effect.**

What happens if we treat time as a constant and let the problem size increase with the number of processors? The inherently sequential fraction of a computation typically decreases as problem size increases (the Amdahl effect). Increasing the number of processors enables us to increase the problem size, decreasing the inherently sequential fraction of a computation, and increasing the quotient between serial execution time and parallel execution time (speedup).

Consider the expression for speedup we have derived. Since $\kappa(n, p) \geq 0$,

$$\psi(n, p) \leq \frac{\sigma(n) + \varphi(n)}{\sigma(n) + \varphi(n)/p}$$

Let s denote the fraction of time spent in the *parallel* computation performing inherently sequential operations. The fraction of time spent in the parallel computation performing parallel operations is what remains, or $(1 - s)$. Mathematically,

$$s = \frac{\sigma(n)}{\sigma(n) + \varphi(n)/p}$$

$$(1 - s) = \frac{\varphi(n)/p}{\sigma(n) + \varphi(n)/p}$$

Hence

$$\sigma(n) = \big(\sigma(n) + \varphi(n)/p\big)s$$

$$\varphi(n) = \big(\sigma(n) + \varphi(n)/p\big)(1 - s)p$$

Therefore

$$\psi(n, p) \leq \frac{\sigma(n) + \varphi(n)}{\sigma(n) + \varphi(n)/p}$$

$$\Rightarrow \psi(n, p) \leq \frac{(\sigma(n) + \varphi(n)/p)(s + (1 - s)p)}{\sigma(n) + \varphi(n)/p}$$

$$\Rightarrow \psi(n, p) \leq s + (1 - s)p$$

$$\Rightarrow \psi(n, p) \leq p + (1 - p)s$$

Gustafson-Barsis's Law [46]
Given a parallel program solving a problem of size n using p processors, let s denote the fraction of total execution time spent in serial code. The maximum speedup ψ achievable by this program is

$$\psi \leq p + (1 - p)s$$

While Amdahl's Law determines speedup by taking a serial computation and predicting how quickly that computation could execute on multiple processors, Gustafson-Barsis's Law does just the opposite. It begins with a parallel computation and estimates how much faster the parallel computation is than the same computation executing on a single processor.

In many cases, assuming a single processor is only p times slower than p processors is overly optimistic. For example, imagine solving a problem on a parallel computer with 16 processors, each with one gigabyte of local memory. Suppose the dataset occupies 15 gigabytes, and the aggregate memory of the parallel computer is barely large enough to hold the dataset and multiple copies of the program. If we tried to solve the same problem on a single processor, the entire dataset would not fit in primary memory. If the working set of the executing program exceeded one gigabyte, it would begin to thrash, taking much more than 16 times as long to execute the parallel portion of the program as the group of 16 processors.

That is why we say that in Gustafson-Barsis's Law, speedup is the time required by a parallel computation divided into the time that *would* be required to solve the same problem on a single CPU, *if* it had sufficient memory. We refer to the speedup predicted by Gustafson-Barsis's Law as **scaled speedup,** because by using the parallel computation as the starting point, rather than the sequential computation, it allows the problem size to be an increasing function of the number of processors.

EXAMPLE 1

An application executing on 64 processors requires 220 seconds to run. Benchmarking reveals that 5 percent of the time is spent executing serial portions of the computation on a single processor. What is the scaled speedup of the application?

■ **Solution**

Since $s = 0.05$, the scaled speedup on 64 processors is

$$\psi = 64 + (1 - 64)(0.05) = 64 - 3.15 = 60.85$$

EXAMPLE 2

Vicki plans to justify her purchase of a $30 million Gadzooks supercomputer by demonstrating its 16,384 processors can achieve a scaled speedup of 15,000 on a problem of great importance to her employer. What is the maximum fraction of the parallel execution time that can be devoted to inherently sequential operations if her application is to achieve this goal?

■ **Solution**

Using Gustafson-Barsis's Law:

$$\begin{aligned} 15{,}000 &= 16{,}384 - 16{,}383s \\ \Rightarrow \quad s &= 1{,}384/16{,}383 \\ \Rightarrow \quad s &= 0.084 \end{aligned}$$

7.5 THE KARP-FLATT METRIC

Because Amdahl's Law and Gustafson-Barsis's Law ignore $\kappa(n, p)$, the parallel overhead term, they can overestimate speedup or scaled speedup. Karp and Flatt have proposed another metric, called the experimentally determined serial fraction, which can provide valuable performance insights [59].

Recall that we have represented the execution time of a parallel program executing on p processors as

$$T(n, p) = \sigma(n) + \varphi(n)/p + \kappa(n, p)$$

where $\sigma(n)$ is the inherently serial component of the computation, $\varphi(n)$ is the portion of the computation that may be executed in parallel, and $\kappa(n, p)$ is overhead resulting from processor communication and synchronization, and redundant computations. The serial program does not have any interprocessor communication or synchronization overhead, so its execution time is

$$T(n, 1) = \sigma(n) + \varphi(n)$$

We define the **experimentally determined serial fraction** e of the parallel computation is the total amount of idle and overhead time scaled by two factors: the number of processors (less one) and the sequential execution time.

$$e = \frac{(p - 1)\sigma(n) + p\kappa(n, p)}{(p - 1)T(n, 1)}$$

Hence

$$e = p\frac{T(n, p) - T(n, 1)}{(p-1)T(n, 1)}$$

We may now rewrite the parallel execution time as

$$T(n, p) = T(n, 1)e + T(n, 1)(1 - e)/p$$

Let's use ψ as a shorthand for $\psi(n, p)$. Since speedup $\psi = T(n, 1)/T(n, p)$, we have $T(n, 1) = T(n, p)\psi$. Hence

$$\begin{aligned} & T(n, p) = T(n, p)\psi e + T(n, p)\psi(1 - e)/p \\ \Rightarrow \quad & 1 = \psi e + \psi(1 - e)/p \\ \Rightarrow \quad & 1/\psi = e + (1 - e)/p \\ \Rightarrow \quad & 1/\psi = e + 1/p - e/p \\ \Rightarrow \quad & 1/\psi = e(1 - 1/p) + 1/p \\ \Rightarrow \quad & e = \frac{1/\psi - 1/p}{1 - 1/p} \end{aligned}$$

The Karp-Flatt Metric [59]

Given a parallel computation exhibiting speedup ψ on p processors, where $p > 1$, the experimentally determined serial fraction e is defined to be

$$e = \frac{1/\psi - 1/p}{1 - 1/p}$$

The experimentally determined serial fraction is a useful metric for two reasons. First, it takes into account parallel overhead [the $\kappa(n, p)$ term] that Amdahl's Law and Gustafson-Barsis's Law ignore. Second, it can help us detect other sources of overhead or inefficiency that are ignored in our simple model of parallel execution time. For example, we assume that p processors execute the parallelizable portion of the computation p times as quickly as a single processor. That is why the $\varphi(n)$ term in $T(n, 1)$ becomes the $\varphi(n)/p$ term in $T(n, p)$. This assumption ignores the fact that the amount of work to be done may not divide evenly among the processors. For example, suppose we have 19 equal and undividable pieces of work, each of which takes one unit of time to complete. If six processors are available, one processor must take four pieces while the other processors take three. The parallel execution time is 4, not 19/6.

For a problem of fixed size, the efficiency of a parallel computation typically decreases as the number of processors increases. By using the experimentally determined serial fraction, we can determine whether this efficiency decrease is due to (1) limited opportunities for parallelism or (2) increases in algorithmic or architectural overhead.

Page 168, line 2: "$\sigma(n) + \kappa(n, p) = T(n, 1)e$" should be

$$e = \frac{pT(n, p) - T(n, 1)}{(p-1)T(n, 1)}$$

EXAMPLE 1

Benchmarking a parallel program on 1, 2, . . . , 8 processors produces the following speedup results:

p	2	3	4	5	6	7	8
ψ	1.82	2.50	3.08	3.57	4.00	4.38	4.71

What is the primary reason for the parallel program achieving a speedup of only 4.71 on eight processors?

■ Solution

Using the formula we have developed, we can compute the experimentally determined serial fraction e corresponding to each data point:

p	2	3	4	5	6	7	8
ψ	1.82	2.50	3.08	3.57	4.00	4.38	4.71
e	0.10	0.10	0.10	0.10	0.10	0.10	0.10

Since the experimentally determined serial fraction is not increasing with the number of processors, the primary reason for the poor speedup is the limited opportunity for parallelism—that is, the large fraction of the computation that is inherently sequential.

EXAMPLE 2

Benchmarking a parallel program on 1, 2, . . . , 8 processors produces the following speedup results:

p	2	3	4	5	6	7	8
ψ	1.87	2.61	3.23	3.73	4.14	4.46	4.71

What is the primary reason for the parallel program achieving a speedup of only 4.71 on eight processors?

■ Solution

We begin by computing the experimentally determined serial fraction for each of these program runs:

p	2	3	4	5	6	7	8
ψ	1.87	2.61	3.23	3.73	4.14	4.46	4.71
e	0.070	0.075	0.080	0.085	0.090	0.095	0.1

Since the experimentally determined serial fraction is steadily increasing as the number of processors increases, the principal reason for the poor speedup is parallel overhead. This could be time spent in process startup, communication, or synchronization, or it could be an architectural constraint.

7.6 THE ISOEFFICIENCY METRIC

Let's refer to a parallel program executing on a parallel computer as a **parallel system.** The **scalability** of a parallel system is a measure of its ability to increase performance as the number of processors increases.

As we have already seen, speedup (and hence efficiency) is typically an increasing function of the problem size, because the communication complexity is usually lower than the computational complexity. We call this the Amdahl effect. In order to maintain the same level of efficiency when processors are added, we can increase the problem size.

These ideas are formalized by the **isoefficiency relation.** To derive the isoefficiency relation, we return to our original definition of speedup:

$$\psi(n, p) \leq \frac{\sigma(n) + \varphi(n)}{\sigma(n) + \varphi(n)/p + \kappa(n, p)}$$

$$\Rightarrow \psi(n, p) \leq \frac{p\big(\sigma(n) + \varphi(n)\big)}{p\sigma(n) + \varphi(n) + p\kappa(n, p)}$$

$$\Rightarrow \psi(n, p) \leq \frac{p\big(\sigma(n) + \varphi(n)\big)}{\sigma(n) + \varphi(n) + (p-1)\sigma(n) + p\kappa(n, p)}$$

We define $T_o(n, p)$ to be the total amount of time spent by all processes doing work not done by the sequential algorithm. One component of this time is the time $p - 1$ processes spend executing inherently sequential code. The other component of this time is the time all p processes spend performing interprocessor communications and redundant computations. Hence $T_o(n, p) = (p-1)\sigma(n) + p\kappa(n, p)$. Substituting $T_o(n, p)$ into our previous equation, we get:

$$\Rightarrow \psi(n, p) \leq \frac{p\big(\sigma(n) + \varphi(n)\big)}{\sigma(n) + \varphi(n) + T_o(n, p)}$$

Since efficiency equals speedup divided by p:

$$\varepsilon(n, p) \leq \frac{\sigma(n) + \varphi(n)}{\sigma(n) + \varphi(n) + T_o(n, p)}$$

$$\Rightarrow \varepsilon(n, p) \leq \frac{1}{1 + \frac{T_o(n, p)}{\sigma(n) + \varphi(n)}}$$

Recalling that $T(n, 1)$ represents sequential execution time:

$$\Rightarrow \varepsilon(n, p) \leq \frac{1}{1 + T_o(n, p)/T(n, 1)}$$

$$\Rightarrow \frac{T_o(n, p)}{T(n, 1)} \leq \frac{1 - \varepsilon(n, p)}{\varepsilon(n, p)}$$

$$\Rightarrow T(n, 1) \geq \frac{\varepsilon(n, p)}{1 - \varepsilon(n, p)} T_o(n, p)$$

If we wish to maintain a constant level of efficiency, the fraction

$$\frac{\varepsilon(n, p)}{1 - \varepsilon(n, p)}$$

is a constant, and the formula simplifies to

$$T(n, 1) \geq CT_o(n, p)$$

Isoefficiency Relation [43]
Suppose a parallel system exhibits efficiency $\varepsilon(n, p)$. Define $C = \varepsilon(n, p)/(1 - \varepsilon(n, p))$ and $T_o(n, p) = (p - 1)\sigma(n) + p\kappa(n, p)$. In order to maintain the same level of efficiency as the number of processors increases, n must be increased so that the following inequality is satisfied:

$$T(n, 1) \geq CT_o(n, p)$$

We can use a parallel system's isoefficiency relation to determine the range of processors for which a particular level of efficiency can be maintained. Since parallel overhead increases when the number of processors increases, the way to maintain efficiency when increasing the number of processors is to increase the size of the problem being solved. The algorithms we are designing assume that the data structures we manipulate fit in primary memory. The maximum problem size we can solve is limited by the amount of primary memory that is available. For this reason we need to treat space as the limiting factor when we perform this analysis.

Suppose a parallel system has isoefficiency relation $n \geq f(p)$. If $M(n)$ denotes the amount of memory required to store a problem of size n, the relation $M^{-1}(n) \geq f(p)$ indicates how the amount of memory used must increase as a function of p in order to maintain a constant level of efficiency. We can rewrite this relation as $n \geq M(f(p))$. The total amount of memory available is a linear function of the number of processors used. Hence the function $M(f(p))/p$ indicates how the amount of memory used *per processor* must increase as a function of p in order to maintain the same level of efficiency. We call $M(f(p))/p$ the **scalability function.**

The complexity of $M(f(p))/p$ determines the range of processors for which a constant level of efficiency can be maintained, as illustrated in Figure 7.4. If $M(f(p))/p = \Theta(1)$, memory requirements per processor are constant, and the parallel system is perfectly scalable. If $M(f(p))/p = \Theta(p)$, memory requirements per processor increase linearly with the number of processors p. While memory is available, it is possible to maintain the same level of efficiency by increasing the problem size. However, since the memory used per processor increases linearly with p, at some point this value will reach the memory capacity of the system. Efficiency cannot be maintained when the number of processors increases beyond this point.

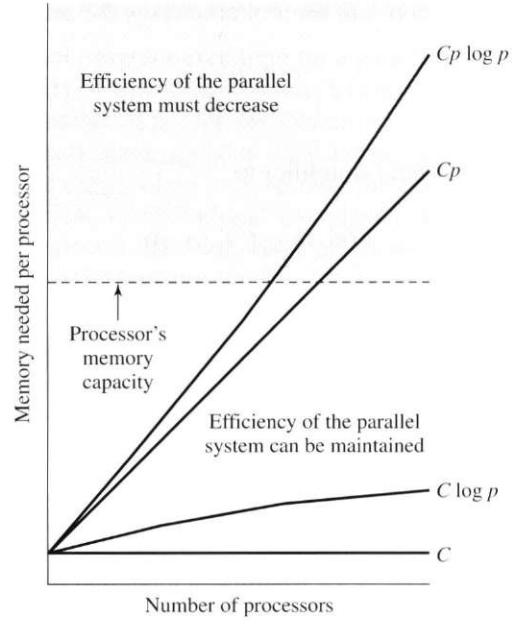

Figure 7.4 The way to maintain efficiency when increasing the number of processors is to increase the size of the problem being solved. The maximum problem size is limited by the amount of memory that is available, which is a linear function of the number of processors. Starting with the isoefficiency relation, and taking into account memory requirements as a function of n, we can determine how the amount of memory used per processor must increase as a function of p to maintain efficiency. The lower the complexity of this function, the more scalable the parallel system.

Similar arguments hold for the cases where $M(f(p))/p = \Theta(\log p)$ and $M(f(p))/p = \Theta(p \log p)$. While constants of proportionality must be taken into account, in general we say that the lower the complexity of $M(f(p))/p$, the higher the scalability of the parallel system.

EXAMPLE 1 Reduction

In Chapter 3 we developed a parallel reduction algorithm. The computational complexity of the sequential reduction algorithm is $\Theta(n)$. The reduction step has time complexity $\Theta(\log p)$. Every processor participates in this step, so $T_0(n, p) = \Theta(p \log p)$. Big-Oh notation ignores constants, but we can assume they are folded into the efficiency constant C.

Hence the isoefficiency relation for the reduction algorithm is

$$n \geq Cp \log p$$

The sequential algorithm reduces n values, so $M(n) = n$. Therefore,

$$M(Cp \log p)/p = Cp \log p/p = C \log p$$

We can mentally confirm that this result makes sense. Suppose we are sum-reducing n values on p processors. Each processor adds about n/p values, then participates in a reduction that has $\lceil \log p \rceil$ steps. If we double the number of processors and double the value of n, each processor's share is still about the same: n/p values. So the time each processor spends adding is the same. However, the number of steps needed to perform the reduction has increased slightly, from $\lceil \log p \rceil$ to $\lceil \log(2p) \rceil$. Hence the efficiency has dropped a bit. In order to maintain the same level of efficiency, we must more than double the value of n when we double the number of processors. The scalability function confirms this, when it shows that the problem size per processor must grow as $\Theta(\log p)$.

Floyd's Algorithm — EXAMPLE 2

Let's determine the isoefficiency function for the parallel implementation of Floyd's algorithm we developed in Chapter 6. The sequential algorithm has time complexity $\Theta(n^3)$. Each of the p processors executing the parallel algorithm spends $\Theta(n^2 \log p)$ time performing communications. Hence the isoefficiency relation is

$$n^3 \geq C(pn^2 \log p) \Rightarrow n \geq Cp \log p$$

This looks like the same relation we had in the previous example, but we have to be careful to consider the memory requirements associated with the problem size n. In the case of Floyd's algorithm the amount of storage needed to represent a problem of size n is n^2; that is, $M(n) = n^2$. The scalability function for this system is:

$$M(Cp \log p)/p = C^2 p^2 \log^2 p/p = C^2 p \log^2 p$$

This parallel system has poor scalability compared to parallel reduction.

Finite Difference Method — EXAMPLE 3

Consider a parallel algorithm implementing a finite difference method to solve a partial differential equation. (We'll consider these algorithms in more detail in Chapter 13.) The problem is represented by a $n \times n$ grid. Each processor is responsible for a subgrid of size $(n/\sqrt{p}) \times (n/\sqrt{p})$ (Figure 7.5). During each iteration of the algorithm every processor sends boundary values to its four neighbors; the time needed to perform these communications is $\Theta(n/\sqrt{p})$ per iteration.

The time complexity of the sequential algorithm solving this problem is $\Theta(n^2)$ per iteration.

The isoefficiency relation for this parallel system is

$$n^2 \geq Cp(n/\sqrt{p}) \Rightarrow n \geq C\sqrt{p}$$

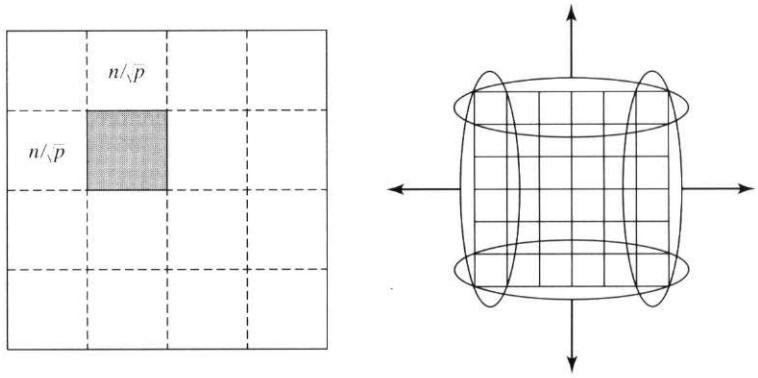

Figure 7.5 Partitioning for a parallel finite difference algorithm. (a) Each process is responsible for an $(n \times \sqrt{p}) \times (n \times \sqrt{p})$ block of the $n \times n$ matrix. (b) During each iteration each process sends *n/sqrt p* boundary values to each of its four neighboring processes.

When we say a problem has size n, we mean the grid has n^2 elements. Hence $M(n) = n^2$ and

$$M(C\sqrt{p})/p = (C\sqrt{p})^2)/p = C^2p/p = C^2$$

The scalability function is $\Theta(1)$, meaning the parallel system is perfectly scalable.

7.7 SUMMARY

Our goal in parallel computing is to use p processors to execute a program p times faster than it executes on a single processor. The ratio of the sequential execution time to parallel execution time is called speedup.

$$\text{Speedup} = \frac{\text{Sequential execution time}}{\text{Parallel execution time}}$$

The efficiency of the parallel computation (also called processor utilization) is the speedup divided by the number of processors:

$$\text{Efficiency} = \frac{\text{Speedup}}{p}$$

To achieve a speedup of p, the parallel execution time must be $1/p$ that of the sequential program. Since there are only p processors, that means each processor must do an equal share of the work, all of the processors must be busy during the entire parallel execution, and there can be no extra operations introduced when the algorithm is made parallel. In other words, speedup equals p if and only if utilization equals 100 percent. In reality, this is rarely the case.

Why? First, there is usually some portion of the algorithm that cannot be performed on multiple processors. This is called serial code, and it prevents us from keeping all the processors busy all the time.

Second, virtually all parallel programs require at least some interactions among the processors. These communications operations do not exist in the sequential program. Hence they represent extra operations introduced when the algorithm is made parallel.

We have developed a general formula for speedup that takes into account the inherently sequential portion of the computation, the parallelizable portion of the computation, and parallel overhead (communication operations and redundant computations). We have also discussed four different lenses for analyzing the performance of parallel programs.

The first lens, Amdahl's Law, is forward looking. It relies upon an evaluation of the sequential program to predict an upper limit to the speedup that can be achieved by using multiple processors to speed the execution of the parallelizable portion of the program.

The second lens, Gustafson-Barsis's Law, is backward looking. It relies upon benchmarking of a parallel program to predict how long the program would take to run on a single processor, if that processor had enough memory. We say that Gustafson-Barsis's Law provides an estimate of scaled speedup, since the size of the problem is allowed to increase with the number of processors.

The third lens, the Karp-Flatt metric, examines the speedup achieved by a parallel program solving a problem of fixed size. The experimentally determined serial fraction can be used to support hypotheses about the performance of the program on larger numbers of processors.

The fourth and final lens, the isoefficiency metric, is used to determine the scalability of a parallel system. A parallel system is perfectly scalable if the same level of efficiency can be sustained as processors are added by increasing the size of the problem being solved. The scalability function, derived from the isoefficiency relation, identifies how the problem size must grow as a function of the number of processors in order to maintain the same level of efficiency.

7.8 KEY TERMS

Amdahl effect
Amdahl's Law
efficiency
experimentally determined serial fraction
Gustafson-Barsis's Law
isoefficiency relation
Karp-Flatt metric
parallel system
scalability
scalability function
scaled speedup
speedup

7.9 BIBLIOGRAPHIC NOTES

The seminal paper of Gustafson, Montry, and Benner [47] not only introduces the notion of scaled speedup, but also is the first to report scaled speedups in excess of 1000. It provides a fascinating glimpse into strategies for extracting maximum

speedup from a massively parallel computer (in their case a 1024-CPU nCUBE multicomputer). The authors won the Gordon Bell Award and the Karp Prize for this work.

You can find a much more detailed introduction to the isoefficiency metric in *Introduction to Parallel Computing* by Grama et al. [44]. Note that I have not adopted their definition of problem size. Grama et al. define problem size to be "the number of basic computation steps in the best sequential algorithm to solve the problem on a single processing element." In other words, when they say "problem size" they mean "sequential time." I believe this definition is counterintuitive, which is why I do not use it in this book.

7.10 EXERCISES

7.1 Using the definition of speedup presented in Section 7.2, prove that there exists a p_0 such that $p > p_0 \Rightarrow \psi(n, p) < \psi(n, p_0)$. Assume $\kappa(n, p) = C \log p$.

7.2 Starting with the definition of efficiency presented in Section 7.2, prove that $p' > p \Rightarrow \varepsilon(n, p') \leq \varepsilon(n, p)$.

7.3 Estimate the speedup achievable by the parallel reduction algorithm developed in Section 3.5 on $1, 2, \ldots, 16$ processors. Assume $n = 1{,}00{,}000$, $\chi = 10$ nanoseconds and $\lambda = 100$ μsec.

7.4 Benchmarking of a sequential program reveals that 95 percent of the execution time is spent inside functions that are amenable to parallelization. What is the maximum speedup we could expect from executing a parallel version of this program on 10 processors?

7.5 For a problem size of interest, 6 percent of the operations of a parallel program are inside I/O functions that are executed on a single processor. What is the minimum number of processors needed in order for the parallel program to exhibit a speedup of 10?

7.6 What is the maximum fraction of execution time that can be spent performing inherently sequential operations if a parallel application is to achieve a speedup of 50 over its sequential counterpart?

7.7 Shauna's parallel program achieves a speedup of 9 on 10 processors. What is the maximum fraction of the computation that may consist of inherently sequential operations?

7.8 Brandon's parallel program executes in 242 seconds on 16 processors. Through benchmarking he determines that 9 seconds is spent performing initializations and cleanup on one processor. During the remaining 233 seconds all 16 processors are active. What is the scaled speedup achieved by Brandon's program?

7.9 Courtney benchmarks one of her parallel programs executing on 40 processors. She discovers that it spends 99 percent of its time inside parallel code. What is the scaled speedup of her program?

7.10 The execution times of six parallel programs, labeled I-VI, have been benchmarked on 1, 2, . . . , 8 processors. The following table presents the speedups achieved by these programs.

	Speedup					
Processors	**I**	**II**	**III**	**IV**	**V**	**VI**
1	1.00	1.00	1.00	1.00	1.00	1.00
2	1.67	1.89	1.89	1.96	1.74	1.94
3	2.14	2.63	2.68	2.88	2.30	2.82
4	2.50	3.23	3.39	3.67	2.74	3.65
5	2.78	3.68	4.03	4.46	3.09	4.42
6	3.00	4.00	4.62	5.22	3.38	5.15
7	3.18	4.22	5.15	5.93	3.62	5.84
8	3.33	4.35	5.63	6.25	3.81	6.50

For each of these programs, choose the statement that *best* describes its likely performance on 16 processors:

A. The speedup achieved on 16 processors will probably be at least 40 percent higher than the speedup achieved on eight processors.

B. The speedup achieved on 16 processors will probably be less than 40 percent higher than the speedup achieved on eight processors, due to the large serial component of the computation.

C. The speedup achieved on 16 processors will probably be less than 40 percent higher than the speedup achieved on eight processors, due to the increase in overhead as processors are added.

7.11 Both Amdahl's Law and Gustafson-Barsis's Law are derived from the same general speedup formula. However, when increasing the number of processors p, the maximum speedup predicted by Amdahl's Law converges on $1/f$, while the speedup predicted by Gustafson-Barsis's Law increases without bound. Explain why this is so.

7.12 Given a problem to be solved and access to all the processors you care to use, can you always solve the problem within a specified time limit? Explain your answer.

7.13 Let $n \geq f(p)$ denote the isoefficiency relation of a parallel system and $M(n)$ denote the amount of memory required to store a problem of size n. Use the scalability function to rank the parallel systems shown below from most scalable to least scalable.

a. $f(p) = Cp$ and $M(n) = n^2$

b. $f(p) = C\sqrt{p}\log p$ and $M(n) = n^2$

c. $f(p) = C\sqrt{p}$ and $M(n) = n^2$

d. $f(p) = Cp\log p$ and $M(n) = n^2$

e. $f(p) = Cp$ and $M(n) = n$

f. $f(p) = p^C$ and $M(n) = n$. Assume $1 < C < 2$.

g. $f(p) = p^C$ and $M(n) = n$. Assume $C > 2$.