Chapter 6 Performance Analysis

> *"The computer scientist's greatest challenge is not to get confused by complexities of his own making."* - E. W. Dijkstra

6.1 Introduction

Sometimes we discover tools that we like so much that we use them even when we should not. Learning how to design and write parallel programs should not be confused with knowing when to do so. Some problems benefit from parallelization and others do not. It is as important to know when to parallelize and understand its costs and benefits as it is to know how to do it. The purpose of this chapter is to give you the means to

- determine how much faster a parallel algorithm performs than a sequential algorithm;
- predict the performance of a parallel algorithm;
- decide whether there are inherent limitations that prevent a parallel algorithm from performing, and what they are; and
- determine how efficient a parallel algorithm can be as the problem size and number of available processors increases.

The theory we describe here can help you avoid the mistake of using a tool that does not fit the job.

6.2 Speedup and Efficiency

Informally, the ***speedup*** of a parallel algorithm is a measure of how much faster it is than a sequential algorithm, but this begs the question, which sequential algorithm, and what does *faster* mean? The second question is the easy one; the speedup of a parallel algorithm over a specific sequential algorithm is the ratio of the execution time of the sequential algorithm to the execution time of the parallel algorithm:

$$\text{Speedup} = \frac{\text{Sequential execution time}}{\text{Parallel execution time}} \tag{6.1}$$

Regarding the first question, for a given problem, more than one sequential algorithm may exist, but not all of these will be suitable for parallelization. On a single processor computer, one tries to use the fastest sequential algorithm that solves the problem. It makes sense to judge the speedup of a parallel algorithm by comparing it to the fastest sequential algorithm for solving the same problem on a single processing element. The problem is that 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 use. In this case, we take the fastest known algorithm that would be a practical choice for a single processor computer as the best sequential algorithm.

There are three components to the execution time that a parallel algorithm would require if it were implemented:

- the part of the computation that must be performed sequentially, called the ***inherently sequential*** part of the computation,
- the part of the computation that can be performed in parallel, called the ***parallelizable part*** of the computation, and

- the overhead that the parallel algorithm has, but that the sequential algorithm does not, which includes:
  - communication overhead
  - computations that are performed redundantly
  - the overhead of creating multiple processes.

We make this more mathematical in order to do some quantitative analysis. To do so, we have to abstract the concept of computation. When we talk about *computations*, we will think of them as being equivalent to time units. If we say, for example that the inherently sequential part of the computation is s, then we think of s as an amount of time. Similarly, if we say the computation in an algorithm that can be performed in parallel is t, then we think of t as being measured in time units. So we use the words *computation* and *time* interchangeably. With this in mind, we introduce some notation.

Definition 1. We let

$\psi(n,p)$ denote the speedup achieved by solving a problem of size n on a computer system with p processors.

$\sigma(n)$ denote the inherently sequential part of the computation of a problem of size n.

$\varphi(n)$ denote the part of the computation of a problem of size n that may be executed in parallel.

$\kappa(n,p)$ denote the parallel overhead of the computation of a problem of size n using p processors.

Observations

1. A sequential program has no parallel overhead, $\kappa(n,p)$, and executes all instructions on a single processor. Therefore the total time spent in a sequential algorithm on a problem of size n is

$$\sigma(n) + \varphi(n) \tag{6.2}$$

.

2. The best possible parallel algorithm still has to execute the inherently sequential portion of the computation sequentially, so that part contributes $\sigma(n)$ time to the parallel execution time. However, this best possible parallel algorithm can execute the parallelizable part of the computation, $\varphi(n)$, on p processors by dividing it up equally among the processors, so that this portion of the computation contributes $\varphi(n)/p$ time to the parallel execution time. Lastly, it still has the parallel overhead, $\kappa(n,p)$. Therefore the total parallel execution time is, in the best case,

$$\sigma(n) + \varphi(n)/p + \kappa(n,p) \tag{6.3}$$

3. If the parallel algorithm is not "the best", it may not be able to uniformly distribute the parallelizable part of the computation among the p processors, and in this case, some processors must necessarily take longer than $\varphi(n)/p$ time to finish their part of the work. This implies that the expression in Eq. 6.3 above is a lower bound on the parallel execution time. Since in the definition of speedup (Eq. 6.1 above), the parallel execution time is in the denominator, the relationship is

$$\psi(n,p) \leq \frac{\sigma(n) + \varphi(n)}{\sigma(n) + \varphi(n)/p + \kappa(n,p)} \tag{6.4}$$

It might seem that throwing more processors at a problem will make the parallel execution time even smaller, but this is not true in general. In most parallel algorithms, the processes need to communicate with each other. As the number of processes increases, the communication costs increase. The objective of agglomeration, you may remember, is to reduce communication overhead by reducing the number of processes that need to exchange data. In general, as p increases, $\varphi(n)/p$ decreases but $\kappa(n,p)$ increases. Therefore, for each n, there is an optimal value of p that minimizes the parallel execution time $\sigma(n) + \varphi(n)/p + \kappa(n,p)$, and consequently maximizes speedup. When p is too large, the speedup is not increased proportionately, and the processors are being wasted.

Suppose that you had to buy time on a parallel machine to run your algorithm, and that the price was based on the total time spent executing on all processors combined. You would be wasting money to run the parallel program with too many processors because you would be paying for the parallel overhead instead of actual computation time. In other words, you would not be utilizing the processors well. Efficiency is a measure of processor utilization.

Definition 2. The ***efficiency*** of a parallel algorithm solving a problem of size n using p processors, denoted $\varepsilon(n,p)$, is defined by

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

From Eq. 6.4 and the above definition, we have

$$\begin{aligned} \varepsilon(n,p) &= \psi(n,p)/p \\ &\leq \frac{\sigma(n)+\varphi(n)}{(\sigma(n)+\varphi(n)/p+\kappa(n,p))\cdot p} \end{aligned}$$

implying

$$\varepsilon(n,p) \leq \frac{\sigma(n)+\varphi(n)}{p\sigma(n)+\varphi(n)+p\kappa(n,p)} \tag{6.5}$$

Also, $p \geq 1$ implies that

$$0 \leq \sigma(n)+\varphi(n) \leq p\sigma(n)+\varphi(n,p) \tag{6.6}$$

.

Since parallel overhead, $\kappa(n,p))$, is non-negative,

$$0 \leq p\kappa(n,p)) \tag{6.7}$$

.

Using (6.6) and (6.7), it follows that

$$0 \leq \sigma(n)+\varphi(n) \leq p\sigma(n)+\varphi(n)+p\kappa(n,p) \tag{6.8}$$

.

Dividing all sides of Eq. 6.8 by $p\sigma(n)+\varphi(n)+p\kappa(n,p)$ shows that $0 \leq \varepsilon(n,p) \leq 1$.

6.3 Amdahl's Law

In 1967, Gene Amdahl, who at the time worked at IBM, argued that there was an inherent limitation to the amount of speedup that could be obtained by performing a computation using more processors [1]. His observation of this fact has come to be called ***Amdahl's Law*** and has been formalized more mathematically than he actually presented it.

Amdahl's Law. 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 for this computation is

$$\psi \leq \frac{1}{f+(1-f)/p} \tag{6.9}$$

Before we prove *Amdahl's Law*, observe that, as p approaches ∞, the upper bound for the speedup approaches $1/f$. In other words, the inverse of the sequential fraction is the most speedup one can ever obtain, so the more inherently sequential computation in an algorithm, the less speedup is possible. For any fixed problem size, the fraction of inherently sequential computation is fixed. *Amdahl's Law* shows that for a fixed problem, the upper bound on speedup is also fixed.

Proof. Starting with Eq. 6.4, we have

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

The fact that $0 \leq \kappa(n,p))$ implies that

$$\psi(n,p) \leq \frac{\sigma(n)+\varphi(n)}{\sigma(n)+\varphi(n)/p+\kappa(n,p)} \leq \frac{\sigma(n)+\varphi(n)}{\sigma(n)+\varphi(n)/p} \tag{6.10}$$

The inherently sequential part of the computation is, by definition, $\sigma(n)$. The total computation is the inherently sequential part plus the parallelizable part, which is $\varphi(n)$. Therefore, the fraction of the computation that is inherently sequential is

$$f = \frac{\sigma(n)}{\sigma(n)+\varphi(n)}.$$

Then

$$1/f = \frac{\sigma(n)+\varphi(n)}{\sigma(n)}$$

implying

$$\sigma(n)+\varphi(n) = \sigma(n)/f \tag{6.11}$$

and

$$\varphi(n) = \sigma(n)/f - \sigma(n) = \sigma(n)(1/f\ -\ 1) \tag{6.12}$$

Combining (6.10), (6.11), and(6.12), we have

$$\begin{aligned}
\psi(n,p) &\leq \frac{\sigma(n)+\varphi(n)}{\sigma(n)+\varphi(n)/p} \\
&= \frac{\sigma(n)/f}{\sigma(n)+\sigma(n)(1/f\ -\ 1)/p} \\
&= \frac{1/f}{1+(1/f\ -\ 1)/p} \\
&= \frac{1}{f+(1-f)/p}
\end{aligned}$$

which completes the proof. □

Example 3. Suppose that we are considering developing a parallel program to improve on an existing sequential program and that we determine that 10% of the execution time of the sequential program is spent in inherently sequential code. (We have to inspect the code to determine this.) The remaining code can be parallelized, although we do not as yet know how many processors would be optimal. What is the maximum possible speedup that could be obtained if we were to develop a parallel version that used ten processors?

Solution In this problem, $f = 0.1$. Applying *Amdahl's Law*, we have

$$\psi \leq \frac{1}{0.1+0.9/10} = \frac{1}{0.1+0.09} \approx 5.26$$

Example 4. We can ask the inverse question. Suppose that we know that fraction of inherently sequential computation is 0.12 in the problem of interest. We know that the upper bound on the speedup with any number of processors is $1/f = 1/0.12 \approx 8.33$. What is the least number of processors that we need to use to obtain a speedup of 5.0?

Solution We solve Eq. 6.9 in terms of p.

$$\begin{aligned}
& \psi && \leq \frac{1}{f + (1-f)/p} \\
\implies & \psi && \leq \frac{p}{pf + (1-f)} \\
\implies & pf\psi + (1-f)\psi && \leq p \\
\implies & (1-f)\psi && \leq p - pf\psi \\
\implies & \frac{(1-f)\psi}{1-f\psi} && \leq p
\end{aligned}$$

This tells us that p must be at least $(1 - 0.12) \cdot 5.0/(1 - 0.12 \cdot 5.0) = 4.40/0.4 = 11$.

6.3.1 Ramifications of Amdahl's Law

Many problems have the property that the inherently sequential part of the solution is a linear function of problem size, n. This is often due to the fact that the time to read the data is linear and the time to output results is either constant or at most linear. In contrast, the time to do the computation is a higher order function, such as n^2 or even n^3. For example, an elementary array sorting algorithm takes time $\Theta(n^2)$ for inputs of size n.

Let us represent the inherently sequential part of some sequential program execution time as a linear function $\sigma(n) = an + b$, for some constants a, $b > 0$, and let us suppose that the parallelizable part is quadratic, i.e., $\varphi(n) = cn^2$ for some constant $c > 0$. To make it concrete, we can let $a = 1000$, $b = 10000$, and $c = 0.25$. This is not unreasonable, because I/O operations take time on the order of milliseconds, and a and b represent primarily I/O operations, whereas c represents the cost of computational instructions, on the order of nanoseconds or microseconds. If we plot the maximum speedup predicted by *Amdahl's Law* with p processors using these functions for $\sigma(n)$ and $\varphi(n)$, for two fixed problem sizes of $n = 20000$ and $n = 1000000$, we have a chart such as the one shown in Figure 6.1.

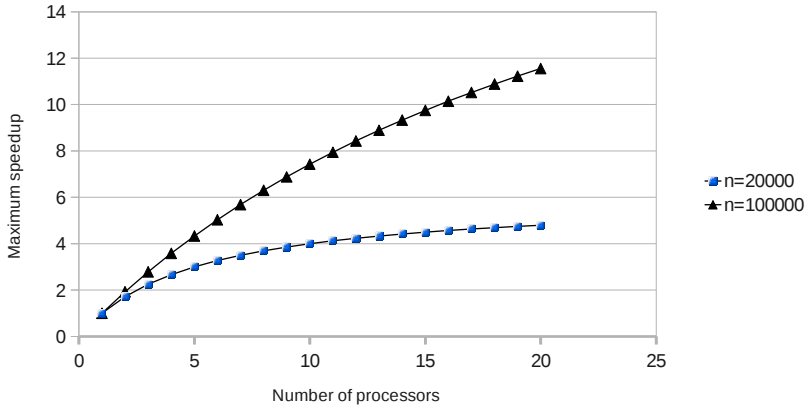

Figure 6.1: Maximum speedup for the functions $\sigma(n) = 1000n + 10000$ and $\varphi(n) = 0.25n^2$, when $n = 20000$ and $n = 1000000$.

Notice that the maximum speedup for a larger problem size is larger. We will discuss this shortly.

Amdahl's Law does not account for the parallel overhead, a large part of which are the communication costs. Suppose we add communication delays to the hypothetical parallel version of the program we just analyzed.

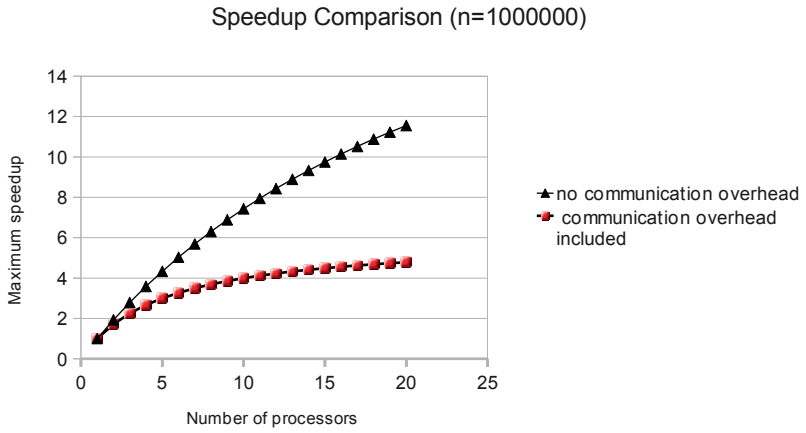

Figure 6.2: Comparison of speedup maximum with and without communication overhead.

In particular, suppose that there is an additional overhead of $\lceil \log n \rceil$ communication points and that each of these points uses

$$10000 \cdot \lceil \log p \rceil + (n/10)$$

of the same unspecified units of time. Then the prediction for speedup taking this into account, for our problem with $\sigma(n) = 1000n + 10000$ and $\varphi(n) = 0.25 \cdot n^2$ is

$$\psi \leq \frac{1000n + 10000 + 0.25 \cdot n^2}{1000n + 10000 + (0.25 \cdot n^2)/p + 10000 \cdot \lceil \log p \rceil + (n/10)}$$

Letting $n = 100000$ as before, the predicted upper bound for a given p is

$$\psi \leq \frac{10001 \times 10^4 + 25 \cdot 10^8}{10002 \times 10^4 + 25 \cdot 10^8/p + 10000 \cdot \lceil \log p \rceil}$$

A graph comparing this function and the one without overhead is shown in Figure 6.2.

6.3.2 The Amdahl Effect

In Section 6.3.1, we observed in one example that as the problem size increased, the fraction of the computation that was inherently sequential decreased. This is often the case. Furthermore, the parallel overhead $\kappa(n, p)$ often has smaller complexity as a function of n than the parallelizable portion of the computation, $\varphi(n)$. A consequence of these facts is that, as the problem size increases, for a fixed number of processors, the maximum possible speedup tends to increase. This relationship has been called the ***Amdahl effect***.

6.4 Gustafson-Barsis's Law

Amdahl's Law states a limit on just how much faster a parallel program can run than a given sequential program *for a problem of a fixed size*. In other words, it starts out with the execution time of solving a problem of fixed size on a single processor, and shows that there is an inherent limit on how much faster the problem can be solved with more processors. It does not take into consideration that adding processors to the parallel computer usually means adding memory to the computer as well, and that the increased memory size makes it possible to solve larger instances of the problem on the computer.

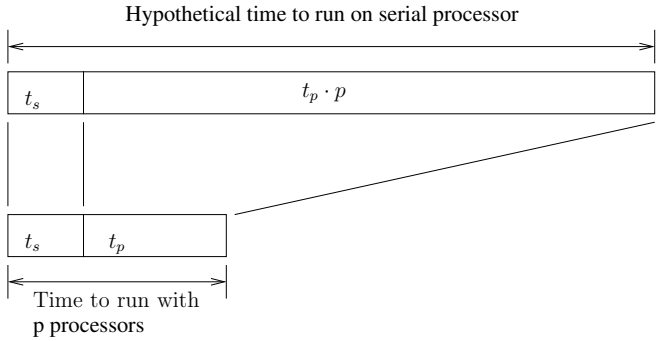

Figure 6.3: Gustafson-Barsis scaled speedup is justified by imagining the parallel program being run on a serial processor. The parallel time expands and the problem size grows in proportion, but the serial time stays fixed.

In 1988, John Gustafson and his colleague Edwin Barsis proposed an alternative way to view the speedup obtained by using a parallel computer [3]. At the time, they worked at *Sandia National Labs* and had found in their work there that parallelizing several programs resulted in speedups greater than *Amdahl's Law* predicted, because the assumptions underlying *Amdahl's Law* did not hold. Gustafson argued that people do not take a fixed-size problem and run it on varying numbers of processors to decrease running time, "except when doing academic research." Instead, he noted, "when given a more powerful processor, the problem generally expands to make use of the increased facilities" [3]. In other words, when the number of processors increases, *people use the larger number of processors to solve a larger problem in the same amount of time.*

There are many examples of this. Someone may be trying to solve a problem like the heat dissipation example that we worked through in Chapter 3. Given a larger number of processors, one would increase the number of grid points to get a more accurate answer, or decrease the time interval and increase the number of simulated time steps. In fact, he argued, the inherently sequential code does not grow at all when grid resolution or time granularity is refined. Therefore, his argument was that, *when the number of processors is increased, the problem size is increased linearly with it and the running time of the parallel version of the program remains fixed, not the problem size.* In essence, when the problem size grows, the parallelizable part of the program grows proportionately and the inherently sequential part tends to diminish, consistent with the Amdahl Effect.

Let us start with a parallel program that is being used to solve a problem of some arbitrary size c. Let t_s denote the amount of time that this parallel program spends executing inherently sequential code, and let t_p denote the amount of time it spends executing the parallel code in the computation with p processors. We ignore the parallel overhead in this analysis, which therefore implies that the total running time of the parallel program for this particular problem size c, using p processors, is $t_s + t_p$.

Let

$$s = \frac{t_s}{t_s + t_p}$$

.

Then s is the fraction of time that the parallel program spends executing inherently sequential code, and

$$(1 - s) = 1 - \frac{t_s}{t_s + t_p} = \frac{t_s + t_p - t_s}{t_s + t_p} = \frac{t_p}{t_s + t_p}$$

is the fraction of time that the parallel program spends executing the parallel part of the code.

Now we turn the problem sideways. Suppose that the reason we used this parallel program on a parallel computer was to increase the problem size. In other words, initially we could only solve a problem of some smaller size, but by using p processors, we were able to solve the larger problem in the same amount of time. Thus, as p increases, c increases linearly with it. We say that problem size ***scales*** with the number

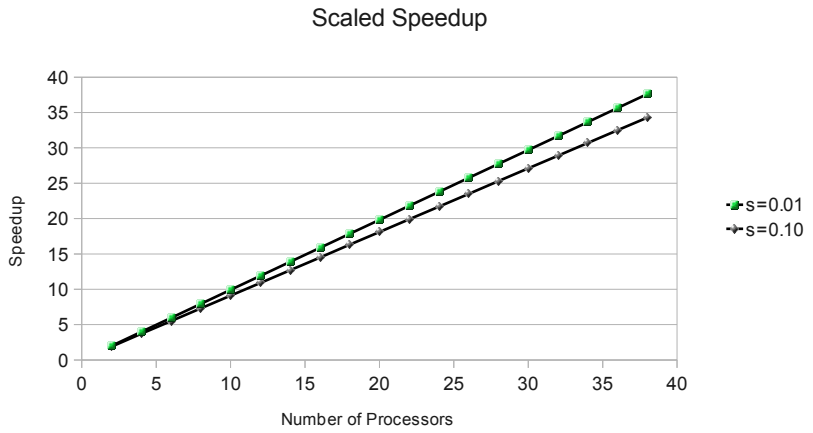

Figure 6.4: Plot of scaled speedup, $S(p)$, with $s = 0.01$ and $s = 0.1$. The graphs are straight lines because this is a linear function of p for each fixed s.

of processors and we define a different notion of speedup, called ***scaled speedup*** (by Gustafson and Barsis), to be the ratio of the amount of time we would have to spend to solve this same problem on a computer with a single processor, to the amount of time that it takes to solve it on our parallel machine with p processors, under the assumption that, as p increases, the parallel running time remains fixed and problem size is increased linearly with p:

$$\text{Scaled speedup} = \frac{\text{Hypothetical time to solve problem on sequential computer}}{\text{Actual parallel execution time}} \tag{6.13}$$

The time to solve this problem on a sequential computer would be the sum of the inherently sequential time, t_s, and the time to execute the parallel code, which would be $t_p \cdot p$, since there is only one processor instead of p, so the total execution time would be $t_s + t_p \cdot p$. This is illustrated in Figure 6.3. Letting $S(p)$ denote the scaled speedup,

$$S(p) = \frac{t_s + t_p p}{t_s + t_p} = \frac{t_s}{t_s + t_p} + \frac{t_p}{t_s + t_p} p = s + (1 - s)p = s + p - ps = p + (1 - p)s \tag{6.14}$$

This is ***Gustafson-Barsis's Law***, which we restate:

Gustafson-Barsis's Law Given a parallel program solving a problem of size n in a given amount of time using p processors, let s denote the fraction of the program's total execution time spent executing inherently sequential code. The scaled speedup achievable by this program, as a function of the number of processors p, is

$$S(p) = p + (1 - p)s$$

Further Observations

1. *Gustafson-Barsis's Law* states the relationship between the left and right-hand sides as an equality, unlike the statement of it in [5]. However, Gustafson intentionally ignores the parallel overhead, $k(n, p)$. As $\kappa(n, p) \geq 0$, the actual execution time that would appear in the denominator of Eq. 6.13, $t_s + t_p + \kappa(p)$ is greater than what is used in Eq. 6.14. This means that the actual scaled speedup is at most what is derived in Eq. 6.13. Hence it is reasonable to state the law as $S(p) \leq p + (1 - p)s$.

2. Quinn uses the function $\psi(n,p)$ in *Gustafson-Barsis's Law*. Here we do not. The problem size n does not appear on the right-hand side of the equation because it is not an independent variable. An assumption underlying this law is that the problem size is dependent on p; therefore, it is not a free variable.

3. The inherently sequential fraction is treated as a constant, not as a free variable. The underlying assumption, as we noted above, is that the initial problem is run on a parallel computer and s is derived from the execution time there. It is based on some initial problem size. As p increases, s remains fixed because it is a fraction derived from a fixed problem size and number of processors.

A plot of the scaled speedup for two fixed values of s is shown in Figure 6.4.

6.5 The *Karp-Flatt Metric*

Neither *Amdahl's Law* nor *Gustafson-Barsis's Law* take into account the parallel overhead, $\kappa(n,p)$. As a result, they both overestimate the maximum possible speedup, because this overhead is never absent. The *Karp-Flatt Metric* [4] provides a means to use empirical data to learn something about this overhead as well as the amount of inherently sequential computation in a parallel program. Like *Gustafson-Barsis's Law*, it starts with the parallel program. The basic idea is that one can collect data from several runs of the program with increasing numbers of processors, and if one knows the running time of the sequential program, one can compute the speedup for these various runs. From the speedup data and the *Karp-Flatt Metric*, one can infer sources of performance degradation. Our derivation of the metric follows the method used by Karp and Flatt [4], rather than that used by Quinn [5], which has errors in the older printings.

To simplify the notation (and to be consistent with Karp and Flatt) we henceforth assume that the problem size is fixed and introduce the following notation:

Notation 5. $T(n,p)$ denotes the execution time of a parallel program solving a problem of size n using p processors. When the meaning is clear, we drop the size argument and write $T(p)$ instead.

Notation 6. $T(n,1)$ denotes the execution time of a sequential program solving a problem of size n. When the meaning is clear, we drop the size argument and write $T(1)$ instead.

Suppose that we let the ***experimentally determined speedup*** of a parallel program solving a problem using p processors be denoted $s(p)$. The way that $s(p)$ is determined is by running the program on p processors and also running the sequential program and computing the fraction

$$s(p) = \frac{T(1)}{T(p)}$$

where $T(p)$ denotes the execution time of a parallel program solving a problem using p processors and $T(1)$ is the execution time of a sequential program. Notice that, here, the speed-up is the *elapsed time* needed by one processor divided by the *elapsed* time needed on p processors. Let f denote the inherently serial fraction of the computation (as it is defined in *Amdahl's Law*). We let T_s denote the time spent in the serial part of the computation (i.e., $\sigma(n)$) and T_p denote the time spent in the parallelizable part of the computation (i.e., $\varphi(n)$). Then

$$f = \frac{T_s}{T_s + T_p} = \frac{T_s}{T(1)}$$

and

$$T_s = f \cdot T(1)$$

and

$$T_p = T(1) - T_s = T(1) - fT(1) = (1-f)T(1)$$

If we ignore the overhead component of the parallel program's running time as well as load-balancing effects, we can write

$$T(p) = T_s + T_p/p \tag{6.15}$$

implying

$$T(p) = f \cdot T(1) + (1-f)T(1)/p$$

and therefore

$$\frac{1}{s(p)} = \frac{T(p)}{T(1)} = \frac{f \cdot T(1) + (1-f)T(1)/p}{T(1)} = f + (1-f)/p$$

Solving for f, we have

$$\begin{aligned} f &= \frac{1}{s(p)} - (1-f)/p) \\ f &= \frac{1}{s(p)} - \frac{1}{p} + \frac{f}{p} \\ f - \frac{f}{p} &= \frac{1}{s(p)} - \frac{1}{p} \\ f(1 - \frac{1}{p}) &= \frac{1}{s(p)} - \frac{1}{p} \end{aligned}$$

implying

$$f = \frac{1/s(p) - 1/p}{1 - 1/p} \tag{6.16}$$

This is what Karp and Flatt call the ***experimentally determined serial fraction*** of a computation. We will henceforth denote it by f_e:

Definition 7. The ***experimentally determined serial fraction*** f_e of a parallel computation of a problem using p processors is defined as

$$f_e = \frac{1/s(p) - 1/p}{1 - 1/p} \tag{6.17}$$

where $s(p)$ is the ***measured speedup*** of the parallel program using p processors. It is important to stress this, because the simplified formula for the parallel execution time in Eq. 6.15 assumes that all processors compute for the same amount of time, i.e., the work is perfectly load balanced. If some processors take longer than others, the measured speed-up will be reduced giving a larger measured serial fraction. In addition, it does not include the overhead of communication.

Load-balancing effects can result in an irregular change in f_e as p increases. For example, if there are 12 tasks that each take the same amount of time, and they are distributed among the processors, one can have perfect load balancing for 2, 3, 4, 6, and 12 processors, but less than perfect load balancing for other values of p. Since a larger load imbalance results in a larger increase in f_e, the value of f_e can be used to identify problems not apparent from speed-up or efficiency measures. For example, the data in Table 6.1, which was obtained by running the Linpack software benchmark on an *Alliance FX/80* with varying numbers of processors, shows that the efficiency ranges from 0.97 for 2 processors, down to 0.749 for 8 processors. This by itself is not so bad. However, the serial fraction ranges from 0.031 up to 0.048 as the number of processors increases. The fact that it is increasing shows that there is some overhead that grows with the number of processors. This is most likely communication overhead, but we cannot know without further analysis, as it could be process startup costs, memory contention, or something else. It could be that the topologies of the interconnection networks on the different machines are different, and that one is more suitable than the other for this software.

On the other hand, the data collected from a run of the same software on a *Cray Y-MP/8* (Figure 6.2) shows that the efficiency drops only slightly from 0.975 for 2 processors to 0.87 for 8 processors, and the

p (number of processors)	2	3	4	5	6	7	8
s (experimental speedup)	1.94	2.79	3.56	4.24	4.89	5.44	5.99
ε (experimental efficiency)	0.970	0.930	0.890	0.848	0.815	0.777	0.749
f_e (experimental serial fraction)	0.031	0.038	0.041	0.045	0.045	0.048	0.048

Table 6.1: Performance data of Linpack running on an *Alliance FX/80* (from [4].)

experimentally determined serial fraction remains constant, supporting the conjecture that on this machine, there is very little overhead. One can use these two tables to infer characteristics of the software that could not be determined from either *Amdahl's Law* or *Gustafson-Barsis's Law*, and as such the *Karp-Flatt Metric* is an important, practical tool.

p (number of processors)	2	3	4	8
s (experimental speedup)	1.95	2.88	3.76	6.96
ε (experimental efficiency)	0.975	0.960	0.940	0.870
f_e (experimental serial fraction)	0.024	0.021	0.021	0.021

Table 6.2: Performance data of Linpack running on a *Cray Y-MP/8* (from [4].)

6.6 The Isoefficiency Relation

In the preceding section, you saw that the same parallel software running on two different parallel computers had very different performance characteristics. This should convince you that it is both the underlying hardware and the software that determine the overall performance. Therefore, rather than discussing the parallel program alone, we discuss ***parallel systems***. We define a ***parallel system*** to be a specific parallel program running on a specific parallel computer.

We introduce a metric in this section that can be used to determine how ***scalable*** a parallel system is. Informally ***scalability*** is the extent to which the efficiency of a parallel system can be maintained as the number of processors and the problem size are increased. If efficiency, and consequently performance, degrades with an increasing number of processors, the system is not scalable. If it stays at least the same, it is scalable.

Recall from the discussion in Section 6.3.2 that we need to increase problem size as the number of processors is increased in order to maintain the same level of efficiency. The *Amdahl Effect* tells us that the inherently serial fraction tends to decrease with increased problem size. Also in Section 6.4, we saw that *Gustafson-Barsis's Law* tells us that the scaled speedup does not have a limiting upper bound when both the problem size and number of processors are increased together. Summarizing, on the one hand, increasing the number of processors alone tends to decrease efficiency, whereas increasing problem size tends to increase efficiency. The question is, at what rate should we increase the problem size with respect to the number of processors to keep the efficiency fixed? The ***isoefficiency relation*** proposed by Grama, Kupta, and Kumar [2] will be the key to this question. It will tell us in a quantitative way the relationship between problem size and the number of processes and the degree of scalability of a parallel system. The word *isoefficiency* means "same efficiency."

6.6.1 Derivation of the Relation

In Section 6.5 we defined $T(n, 1)$ to represent the amount of time that the sequential program executes on a single processor on a problem of size n, and $T(n, p)$ to be the time that the parallel program executes on p processors.

Let $T_0(n, p)$ denote the total amount of time spent by all processes in the parallel program performing work that is not performed in the sequential program. This includes communication costs, process startup costs, contention for shared memory, and so on. But it also includes the fact that in the parallel program, every

process executes the inherently sequential code. For example, if a program has 1000 lines of inherently sequential code that are each executed by 16 processes, then 16,000 lines are executed in total in the parallel system, versus only 1000 lines in the sequential program. Since $T_0(n,p)$ is the total computation done by the parallel program not done by the sequential program, and $T(n,1)$ is the time the sequential program takes to execute, it follows that

$$T(n,1) + T_0(n,p) = p \cdot T(n,p) \tag{6.18}$$

because the parallel program is executed by p processes each spending $T(n,p)$ time[1]. Therefore

$$T(n,p) = \frac{T(n,1) + T_0(n,p)}{p}$$

Since speedup is the ratio of the sequential program's running time divided by the parallel program's running time, we can write it as

$$\psi(n,p) = \frac{T(n,1)}{T(n,p)} = \frac{p \cdot T(n,1)}{T(n,1) + T_0(n,p)}$$

and since efficiency is speedup divided by the number of processors, this leads to

$$\begin{aligned} \varepsilon(n,p) &= \frac{T(n,1)}{T(n,1) + T_0(n,p)} \\ &= \frac{1}{1 + T_0(n,p)/T(n,1)} \end{aligned} \tag{6.19}$$

In other words, the efficiency of the system depends entirely on the ratio between the parallel overhead $T_0(n,p)$ and the serial execution time. If that ratio can be kept constant by increasing n and p appropriately, then efficiency can be held constant. For different parallel systems, we must increase n at different rates with respect to p to maintain a fixed efficiency. For example, n might need to grow as an exponential function of p; such a system would be poorly scalable, as it is difficult to obtain good speedup for a large number of processors on such a system unless the problem size were extremely large, which may not be feasible. In contrast, if n needs to grow only linearly with respect to p, then the system is highly scalable. Thus, it would be useful to have a function of p that could tell us how n has to grow to maintain efficiency, i.e., an isoefficiency function.

Using Eq. 6.19, we have

$$\begin{aligned} &\varepsilon(n,p)\,(1 + T_0(n,p)/T(n,1)) = 1 \\ \Rightarrow\quad &\varepsilon(n,p) + \frac{\varepsilon(n,p)T_0(n,p)}{T(n,1)} = 1 \end{aligned}$$

implying

$$T(n,1)(1 - \varepsilon(n,p)) = \varepsilon(n,p)T_0(n,p)$$

and

$$T(n,1) = \frac{\varepsilon(n,p)}{(1 - \varepsilon(n,p))} T_0(n,p) \tag{6.20}$$

If the goal is that efficiency is held constant as both n and p increase, then the factor $\varepsilon(n,p)/(1-\varepsilon(n,p))$ is treated as remaining a constant for all n and p. Let us write

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

[1] In fact, the right-hand side of Eq. 6.18 is at least as large as the left-hand side, because $T(n,p)$ is the total elapsed time spent by all processes, and some of these processes might have finished their work before others. Therefore, it is more accurate to replace the $=$ relation by $\leq$.

Then C is a constant depending on the efficiency alone, and we can rewrite Eq. 6.20 as

$$T(n,1) = C \cdot T_0(n,p) \tag{6.21}$$

This is the ***isoefficiency function*** defined in [2]. In fact, since Eq. 6.19 really defines an upper bound on the efficiency, the right-hand side of (6.21) is a lower bound (which you can verify from the steps leading to it) and the ***isoefficiency relation*** is

$$T(n,1) \geq C \cdot T_0(n,p)$$

How is this relation used? The following definition is the first step.

Definition 8. Suppose that a parallel system with p processors has efficiency $\varepsilon(n,p)$ for a problem of size n. Define $C = \varepsilon(n,p)/(1-\varepsilon(n,p))$ and let $T_0(n,p)$ be the total amount of time spent by all processes in the parallel program performing work that is not performed in the sequential program. In order to maintain the same level of efficiency as p is increased, n must be increased so that the inequality

$$T(n,1) \geq CT_0(n,p) \tag{6.22}$$

is satisfied.

Notice that the left-hand side of (6.22), which is the running time of the sequential program, is some function $g(n)$. The right-hand side is a function of n and p, say $q(n,p)$. We can solve the inequality $g(n) \geq q(n,p)$ to determine for what range of values of n it will be true. For simplicity we let the constant C equal 1. Assuming that $g(n)$ is invertible, we can write $g^{-1}(g(n)) \geq g^{-1}(q(n,p))$ or $n \geq g^{-1}(q(n,p))$ or $n \geq f(n,p)$ where $f = g^{-1}q$. In other words, the isoefficiency relation can always be expressed in the form $n \geq f(n,p)$ for some suitable function f.

We cannot increase the problem size arbitrarily. Machines have a finite amount of memory and the problem must fit into that memory. The assumption is that the data must be stored in memory[2]. Therefore, the memory size is the limiting factor in how large n may be.

Suppose for simplicity that the isoefficiency relation for a particular parallel system is $n \geq f(p)$, i.e., the overhead does not depend on n. Let $M(n)$ be a function that describes the amount of memory that a problem of size n requires. For example, in Floyd's algorithm, we say that the problem size is n because n is the number of vertices in the graph. The algorithm uses $\Theta(n^2)$ memory, so $M(n) = n^2$. If m is an amount of memory in a parallel computer, then $M^{-1}(m)$ is the size of the largest problem that can be stored in that amount of memory. Therefore the inequality $M^{-1}(m) \geq f(p)$ indicates how the memory must increase to maintain the same level of efficiency as p increases. Thus

$$M(M^{-1}(m)) \geq M(f(p))$$

which implies

$$m \geq M(f(p))$$

must be true. In other words, the amount of available memory must be greater than or equal to $M(f(p))$. In general the amount of memory available on a parallel computer is a linear function of the number of processors, p. Therefore $M(f(p))/p$ is the least amount of memory per processor that must be available to solve the problem at the same level of efficiency as p increases. The function $M(f(p))/p$ is called the ***scalability function***. Figure 6.5 shows how the scalability function's growth rate is a factor in the sizes of the problems that can be solved with the given parallel system.

If $M(f(p))/p$ is $\Theta(1)$ then the amount of memory needed per processor is constant and the parallel system will scale without bound. On the other hand if it is anything faster than constant, it will eventually reach a limit. How quickly it reaches that limit depends on how fast it grows. As long as there is memory available, the efficiency can be maintained, but once the limit is reached, efficiency will drop.

[2]From a practical point of view, the data must reside in internal memory. In theory it can reside on any storage device locally accessible to the processor. In either case, the data is in a storage device locally accessible to the processor.

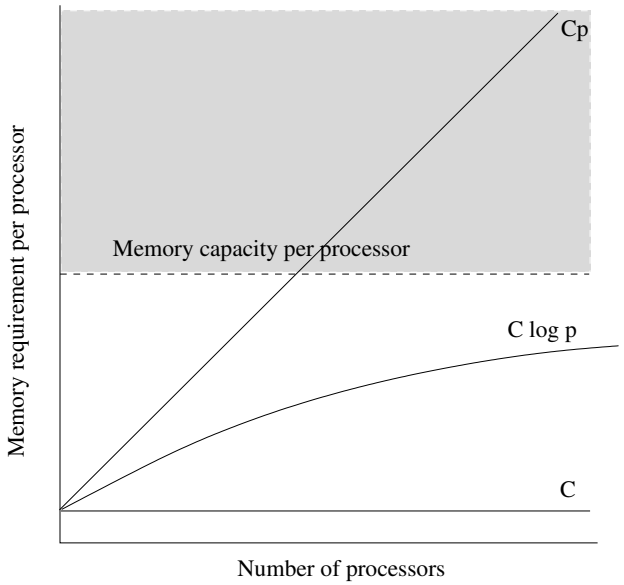

Figure 6.5: Graphs of some scalability functions. If the scalability function grows faster than a constant, then eventually it will reach the memory capacity of each processor. Slower-growing functions may not reach this limit in a practical setting, but faster growing functions will make scaling the parallel system infeasible.

6.6.2 Examples

6.6.2.1 Parallel Reduction

The first example is from Chapter 3, where we introduced the parallel reduction algorithm. A sequential algorithm to perform reduction of n data items has running time $T(n,1) = \Theta(n)$. The parallel reduction required $\Theta(\log p)$ communication steps, which is pure overhead not present in the sequential algorithm. Each of p processes participates and uses $\log p$ steps, so the parallel overhead function $T_0(n,p)$ is $\Theta(p \log p)$. Order notation hides constants but we can assume all constants are factors of C, so the isoefficiency relation is

$$n \geq Cp \log p$$

This means that the input size has to grow at the rate of $p \log p$ for efficiency to be maintained. The question is, will memory allow it? The memory function requirement for sequential reduction is $M(n) = n$, because we need to store n values to perform the reduction, so the scalability function is

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

This shows that the amount of memory per processor must grow proportional to $\log p$ if we are to maintain efficiency. Because memory does not grow at this rate, there will be a limit to how well this will scale.

Does this result make intuitive sense? Consider how the parallel reduction algorithm worked. Given n values to be reduced, each processor performed a sequential reduction on n/p values roughly, and then participated in a parallel reduction taking about $\log p$ steps. That means it needed to store n/p values. Let us see how well it scales. Suppose we multiply the problem size and the number of processors by the same constant k. The sequential algorithm now takes kn steps roughly. In the parallel algorithm, each processor still has the same portion of data, n/p values (because $kn/kp = n/p$), so the per-processor memory requirement does not change, but the parallel overhead increases, since it becomes $kp \cdot \log(kp)$ steps. The efficiency decreases because the parallel overhead increased much more than the sequential execution time ($kp \cdot \log(kp)$ versus kp.) So problem size must grow more than the number of processors for efficiency to be maintained. It must grow proportionately to $p \cdot \log p$, as the isoefficiency relation shows, but the scalability function shows us that this will not be possible without limit.

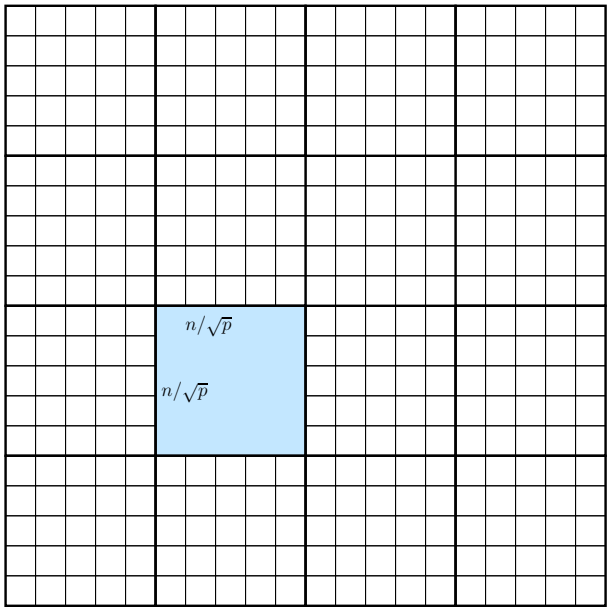

Figure 6.6: Partitioning and agglomeration of an $n \times n$ matrix among p processors using checkerboarding. Each processor has a $n/\sqrt{p} \times n/\sqrt{p}$ submatrix and sends to each neighbor the cells in its block that bound that neighbor's block.

6.6.2.2 Floyd's Algorithm

We return to Floyd's algorithm from Chapter 6. The conservative estimate of communication overhead given there, the one that did not assume overlap of message transmission and computation was $n \lceil \log p \rceil (\lambda + 4n/\beta) = \Theta(n^2 log\, p)$. Each process participates in this step, so the total communication overhead, $T_0(n,p) = \Theta(n^2 p \log p)$. The sequential algorithm uses $\Theta(n^3)$ time. Therefore, the isoefficiency relation is

$$n^3 \geq C \cdot n^2 p \cdot \log p \quad \Rightarrow n \geq C \cdot p \cdot \log p$$

This shows, as in the preceding example, that the problem size must increase as $p \cdot \log p$ if we are to maintain efficiency. Let us see whether this is feasible in terms of memory. The memory function for Floyd's algorithm is $M(n) = n^2$ because the adjacency matrix uses $\Theta(n^2)$ storage. Therefore,

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

This shows that the amount of memory per processor would have to increase as $p \log^2 p$ to maintain efficiency, which is not feasible. In other words, this parallel system is not very scalable.

6.6.2.3 Finite Difference Method

There are many parallel matrix processing algorithms that partition and agglomerate by a method known as ***checkerboarding***, which basically means that the $n \times n$ matrix is divided up into square submatrices of size $n/\sqrt{p} \times n/\sqrt{p}$ and each processor is responsible for one of these submatrices (see Figure 6.6.) The communication pattern depends upon the particular problem, but it often requires that each processor sends data to its four adjacent neighbors (north, east, south, and west.) One problem that uses checkerboarding is the finite difference method for solving a differential equation. It is not important to understand the exact nature of the problem right now; we will study this problem in a later chapter. What does matter is understanding that in this algorithm, in each iteration, each processor sends its boundary values to its four neighbors. The boundary values are simply the values in each matrix cell on one of its boundaries. There are $n/\sqrt{p}$ such cells in each of its north, east, south, and west boundaries. Since each communication to each

neighbor takes time proportional to the amount of data, each individual process takes time proportional to $n/\sqrt{p}$ in a single transfer of data. All p processors must do this in a given iteration of the algorithm, so the parallel overhead is $\Theta(p \cdot n/\sqrt{p}) = \Theta(n\sqrt{p})$. In the sequential algorithm, each iteration takes $\Theta(n^2)$ steps, as just a single process must update each matrix cell one after the other. The isoefficiency relation is therefore

$$n^2 \geq Cn\sqrt{p} \quad \Rightarrow n \geq C\sqrt{p}$$

The memory function for this problem is $M(n) = n^2$ because we took n above to mean the number of rows or columns of the matrix. (Had we taken n to be the total number of matrix entries, then we would have to substitute $\sqrt{n}$ in all formulas above where we used n.) The scalability function is therefore

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

which is a constant. This means that the parallel system is perfectly scalable, because the memory requirements per processor do not need to increase to maintain the same level of efficiency.

References

[1] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[2] Ananth Y. Grama, A. Gupta, and V. Kumar. Isoefficiency: measuring the scalability of parallel algorithms and architectures. *Parallel Distributed Technology: Systems Applications, IEEE*, 1(3):12–21, Aug 1993.

[3] John L. Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31:532–533, 1988.

[4] Alan H. Karp and Horace P. Flatt. Measuring parallel processor performance. *Communications of the ACM*, 33(5):539–543, 1990.

[5] M.J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Higher Education. McGraw-Hill Higher Education, 2004.

Subject Index