receive a message from task A, after which B will send a message to A. Because each task is waiting for the other to send it a message first, both tasks will be blocked forever. Fortunately, deadlocks are not difficult to discover, as the tasks will stop at the point of the deadlock.

2.5. A QUANTITATIVE LOOK AT PARALLEL COMPUTATION

The two main reasons for implementing a parallel program are to obtain better performance and to solve larger problems. Performance can be both modeled and measured, so in this section we will take a another look at parallel computations by giving some simple analytical models that illustrate some of the factors that influence the performance of a parallel program.

Consider a computation consisting of three parts: a setup section, a computation section, and a finalization section. The total running time of this program on one PE is then given as the sum of the times for the three parts.

Equation 2.1

$$T_{total}(1) = T_{setup} + T_{compute} + T_{finalization}$$

What happens when we run this computation on a parallel computer with multiple PEs? Suppose that the setup and finalization sections cannot be carried out concurrently with any other activities, but that the computation section could be divided into tasks that would run independently on as many PEs as are available, with the same total number of computation steps as in the original computation. The time for the full computation on P PEs can therefore be given by Of course, Eq. 2.2 describes a very idealized situation. However, the idea that computations have a serial part (for which additional PEs are useless) and a parallelizable part (for which more PEs decrease the running time) is realistic. Thus, this simple model captures an important relationship.

Equation 2.2

$$T_{total}(P) = T_{setup} + \frac{T_{compute}(1)}{P} + T_{finalization}$$

An important measure of how much additional PEs help is the relative speedup S, which describes how much faster a problem runs in a way that normalizes away the actual running time.

Equation 2.3

$$S(P) = \frac{T_{total}(1)}{T_{total}(P)}$$

A related measure is the efficiency E, which is the speedup normalized by the number of PEs.

Equation 2.4

$$E(P) = \frac{S(P)}{P}$$

Equation 2.5

$$= \frac{T_{total}(1)}{P\,T_{total}(P)}$$

Ideally, we would want the speedup to be equal to P, the number of PEs. This is sometimes called perfect linear speedup. Unfortunately, this is an ideal that can rarely be achieved because times for setup and finalization are not improved by adding more PEs, limiting the speedup. The terms that cannot be run concurrently are called the serial terms. Their running times represent some fraction of the total, called the *serial fraction,* denoted γ.

Equation 2.6

$$\gamma = \frac{T_{setup} + T_{finalization}}{T_{total}(1)}$$

The fraction of time spent in the parallelizable part of the program is then (1 — γ). We can thus rewrite the expression for total computation time with P PEs as

Equation 2.7

$$T_{total}(P) = \gamma\,T_{total}(1) + \frac{(1-\gamma)\,T_{total}(1)}{P}$$

Now, rewriting S in terms of the new expression for $T_{total}(P)$, we obtain the famous Amdahl's law:

Equation 2.8

$$S(P) = \frac{T_{total}(1)}{(\gamma + \frac{1-\gamma}{P})\,T_{total}(1)}$$

Equation 2.9

$$= \frac{1}{\gamma + \frac{1-\gamma}{P}}$$

Thus, in an ideal parallel algorithm with no overhead in the parallel part, the speedup should follow Eq. 2.9. What happens to the speedup if we take our ideal parallel algorithm and use a very large number of processors? Taking the limit as P goes to infinity in our expression for S yields

Equation 2.10

$$S = \frac{1}{\gamma}$$

Eq. 2.10 thus gives an upper bound on the speedup obtainable in an algorithm whose serial part represents γ of the total computation.

These concepts are vital to the parallel algorithm designer. In designing a parallel algorithm, it is important to understand the value of the serial fraction so that realistic expectations can be set for performance. It may not make sense to implement a complex, arbitrarily scalable parallel algorithm if 10% or more of the algorithm is serial—and 10% is fairly common.

Of course, Amdahl's law is based on assumptions that may or may not be true in practice. In real life, a number of factors may make the actual running time longer than this formula implies. For example, creating additional parallel tasks may increase overhead and the chances of contention for shared resources. On the other hand, if the original serial computation is limited by resources other than the availability of CPU cycles, the actual performance could be much better than Amdahl's law would predict. For example, a large parallel machine may allow bigger problems to be held in memory, thus reducing virtual memory paging, or multiple processors each with its own cache may allow much more of the problem to remain in the cache. Amdahl's law also rests on the assumption that for any given input, the parallel and serial implementations perform exactly the same number of computational steps. If the serial algorithm being used in the formula is not the best possible algorithm for the problem, then a clever parallel algorithm that structures the computation differently can reduce the total number of computational steps.

It has also been observed [Gus88] that the exercise underlying Amdahl's law, namely running exactly the same problem with varying numbers of processors, is artificial in some circumstances. If, say, the parallel application were a weather simulation, then when new processors were added, one would most likely increase the problem size by adding more details to the model while keeping the total execution time constant. If this is the case, then Amdahl's law, or fixed-size speedup, gives a pessimistic view of the benefits of additional processors.

To see this, we can reformulate the equation to give the speedup in terms of performance on a P-processor system. Earlier in Eq. 2.2, we obtained the execution time for T processors, $T_{total}(P)$, from the execution time of the serial terms and the execution time of the parallelizable part when executed on one processor. Here, we do the opposite and obtain $T_{total}(1)$ from the serial and parallel terms when executed on P processors.

Equation 2.11

$$T_{total}(1) = T_{setup} + PT_{compute}(P) + T_{finalization}$$

Now, we define the so-called scaled serial fraction, denoted γ_{scaled}, as

Equation 2.12

$$\gamma_{scaled} = \frac{T_{setup} + T_{finalization}}{T_{total}(P)}$$

and then

Equation 2.13

$$T_{total}(1) = \gamma_{scaled} T_{total}(P) + P(1 - \gamma_{scaled}) T_{total}(P)$$

Rewriting the equation for speedup (Eq. 2.3) and simplifying, we obtain the scaled (or fixed-time) speedup.[1]

[1] This equation, sometimes known as Gustafson's law, was attributed in [Gus88] to E. Barsis.

Equation 2.14

$$S(P) = P + (1 - P)\gamma_{scaled}.$$

This gives exactly the same speedup as Amdahl's law, but allows a different question to be asked when the number of processors is increased. Since γ_{scaled} depends on P, the result of taking the limit isn't immediately obvious, but would give the same result as the limit in Amdahl's law. However, suppose we take the limit in P while holding $T_{compute}$ and thus γ_{scaled} constant. The interpretation is that we are increasing the size of the problem so that the total running time remains constant when more processors are added. (This contains the implicit assumption that the execution time of the serial terms does not change as the problem size grows.) In this case, the speedup is linear in P. Thus, while adding more processors to solve a fixed problem may hit the speedup limits of Amdahl's law with a relatively small number of processors, if the problem grows as more processors are added, Amdahl's law will be pessimistic. These two models of speedup, along with a fixed-memory version of speedup, are discussed in [SN90].

2.6. COMMUNICATION

2.6.1. Latency and Bandwidth

A simple but useful model characterizes the total time for message transfer as the sum of a fixed cost