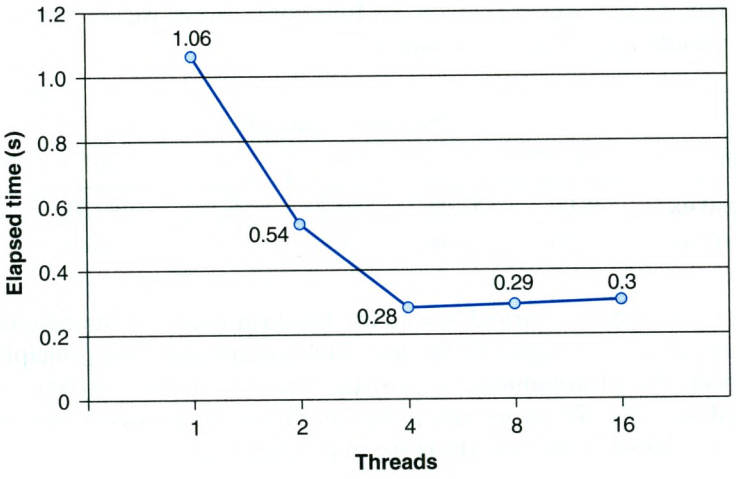

Figure 12.35
Performance of `psum-local` (Figure 12.34). Summing a sequence of 2^{31} elements using four processor cores.

An important lesson to take away from this exercise is that writing parallel programs is tricky. Seemingly small changes to the code have a significant impact on performance.

Characterizing the Performance of Parallel Programs

Figure 12.35 plots the total elapsed running time of the `psum-local` program in Figure 12.34 as a function of the number of threads. In each case, the program runs on a system with four processor cores and sums a sequence of $n = 2^{31}$ elements. We see that running time decreases as we increase the number of threads, up to four threads, at which point it levels off and even starts to increase a little.

In the ideal case, we would expect the running time to decrease linearly with the number of cores. That is, we would expect running time to drop by half each time we double the number of threads. This is indeed the case until we reach the point ($t > 4$) where each of the four cores is busy running at least one thread. Running time actually increases a bit as we increase the number of threads because of the overhead of context switching multiple threads on the same core. For this reason, parallel programs are often written so that each core runs exactly one thread.

Although absolute running time is the ultimate measure of any program's performance, there are some useful relative measures that can provide insight into how well a parallel program is exploiting potential parallelism. The *speedup* of a parallel program is typically defined as

Speedup:

$$S_p = \frac{T_1}{T_p}$$

where p is the number of processor cores and T_k is the running time on k cores. This formulation is sometimes referred to as *strong scaling*. When T_1 is the execution

Threads (t)	1	2	4	8	16
Cores (p)	1	2	4	4	4
Running time (T_p)	1.06	0.54	0.28	0.29	0.30
Speedup (S_p)	1	1.9	3.8	3.7	3.5
Efficiency (E_p)	100%	98%	95%	91%	88%

Figure 12.36 Speedup and parallel efficiency for the execution times in Figure 12.35.

time of a sequential version of the program, then S_p is called the *absolute speedup*. When T_1 is the execution time of the parallel version of the program running on one core, then S_p is called the *relative speedup*. Absolute speedup is a truer measure of the benefits of parallelism than relative speedup. Parallel programs often suffer from synchronization overheads, even when they run on one processor, and these overheads can artificially inflate the relative speedup numbers because they increase the size of the numerator. On the other hand, absolute speedup is more difficult to measure than relative speedup because measuring absolute speedup requires two different versions of the program. For complex parallel codes, creating a separate sequential version might not be feasible, either because the code is too complex or because the source code is not available.

A related measure, known as *efficiency*, is defined as

efficiency:

$$E_p = \frac{S_p}{p} = \frac{T_1}{pT_p}$$

and is typically reported as a percentage in the range (0, 100]. Efficiency is a measure of the overhead due to parallelization. Programs with high efficiency are spending more time doing useful work and less time synchronizing and communicating than programs with low efficiency.

Figure 12.36 shows the different speedup and efficiency measures for our example parallel sum program. Efficiencies over 90 percent such as these are very good, but do not be fooled. We were able to achieve high efficiency because our problem was trivially easy to parallelize. In practice, this is not usually the case. Parallel programming has been an active area of research for decades. With the advent of commodity multi-core machines whose core count is doubling every few years, parallel programming continues to be a deep, difficult, and active area of research.

There is another view of speedup, known as *weak scaling*, which increases the problem size along with the number of processors, such that the amount of work performed on each processor is held constant as the number of processors increases. With this formulation, speedup and efficiency are expressed in terms of the total amount of work accomplished per unit time. For example, if we can double the number of processors and do twice the amount of work per hour, then we are enjoying linear speedup and 100 percent efficiency.

Weak scaling is often a truer measure than strong scaling because it more accurately reflects our desire to use bigger machines to do more work. This is particularly true for scientific codes, where the problem size can be easily increased and where bigger problem sizes translate directly to better predictions of nature. However, there exist applications whose sizes are not so easily increased, and for these applications strong scaling is more appropriate. For example, the amount of work performed by real-time signal-processing applications is often determined by the properties of the physical sensors that are generating the signals. Changing the total amount of work requires using different physical sensors, which might not be feasible or necessary. For these applications, we typically want to use parallelism to accomplish a fixed amount of work as quickly as possible.

Practice Problem 12.11 (solution page 1038)

Fill in the blanks for the parallel program in the following table. Assume strong scaling.

Threads (t)	1	2	4
Cores (p)	1	2	4
Running time (T_p)	12	8	6
Speedup (S_p)	_____	1.5	_____
Efficiency (E_p)	100%	_____	50%

12.7 Other Concurrency Issues

You probably noticed that life got much more complicated once we were asked to synchronize accesses to shared data. So far, we have looked at techniques for mutual exclusion and producer-consumer synchronization, but this is only the tip of the iceberg. Synchronization is a fundamentally difficult problem that raises issues that simply do not arise in ordinary sequential programs. This section is a survey (by no means complete) of some of the issues you need to be aware of when you write concurrent programs. To keep things concrete, we will couch our discussion in terms of threads. Keep in mind, however, that these are typical of the issues that arise when concurrent flows of any kind manipulate shared resources.

12.7.1 Thread Safety

When we program with threads, we must be careful to write functions that have a property called thread safety. A function is said to be *thread-safe* if and only if it will always produce correct results when called repeatedly from multiple concurrent threads. If a function is not thread-safe, then we say it is *thread-unsafe*.

We can identify four (nondisjoint) classes of thread-unsafe functions:

Class 1: *Functions that do not protect shared variables.* We have already encountered this problem with the `thread` function in Figure 12.16, which