CHAPTER 8

Measuring Speedup

in which we define several metrics for a parallel program's performance; we predict what the ideal theoretical metrics should look like; and we measure the actual metrics for our parallel program

CHAPTER 8 Measuring Speedup

8.1 Speedup Metrics

In Chapter 1, we said that one way parallel computing could help with today's massive computational problems is to reduce the time needed to get the results—a **speedup**. The time has come to define speedup precisely and to see how much speedup a real parallel program can achieve.

A program's **problem size**, N, is the number of computations the program performs to solve that problem. The particular problem determines how the problem size is measured. For an image-processing problem, the problem size might be the number of pixels in the image. For an $n \times n$-pixel image, the problem size would be $N = n^2$. For the AES key search problem, the problem size is the number of keys tested: $N = 2^n$, where n is the number of missing key bits. In general, the problem size is defined so that the amount of computation is proportional to N.

A program's **running time**, T, is the amount of time the program takes to compute the answer to a problem. Many factors influence T. The computer's hardware characteristics—such as CPU clock speed, memory speed, caches, and so on—affect T; the faster the computer, the shorter the running time. T depends on the problem size; the larger the problem, the longer the running time. Furthermore, the algorithm used to solve the problem determines how quickly T increases as n increases; an $O(n \log n)$ algorithm's running time will not grow nearly as quickly as an $O(n^2)$ algorithm's. T also depends on how the program is implemented; the same algorithm can sometimes run faster when coded differently. T depends on the number of processors K; adding more processors reduces the running time (one hopes).

When comparing the performance of different versions of a program, such as sequential and parallel versions using the same basic algorithm, we will always run the programs on the same computer. Furthermore, each processor of the parallel computer will have the same hardware characteristics; each processor will have the same CPU clock speed, for example. Thus, there will be no variation in the running times due to different algorithms or different hardware characteristics, and the only factors that influence T are N and K. We use the notation $T(N,K)$ to emphasize that the running time is a function of the problem size and the number of processors. When we need to distinguish the running times of the sequential version and the parallel version of a certain program, we write $T_{\text{seq}}(N,K)$ and $T_{\text{par}}(N,K)$.

A program's **speed**, S, is the rate at which program runs can be done. Speed is the reciprocal of running time:

$$S(N,K) = \frac{1}{T(N,K)} \tag{8.1}$$

T is measured in seconds per program run, S is measured in program runs per second.

A program's **speedup** is the speed of the *parallel* version running on K processors relative to the speed of the *sequential* version running on one processor for a given problem size N:

$$Speedup(N,K) = \frac{S_{\text{par}}(N,K)}{S_{\text{seq}}(N,1)} \tag{8.2}$$

Note that the denominator is $S_{\text{seq}}(N,1)$, not $S_{\text{par}}(N,1)$. Speedup compares the parallel version of a program to the sequential version, not the parallel version to itself. If we were solving the problem on one processor, we would run the sequential version of the program to avoid the parallel version's extra overhead. (Unfortunately, some published papers calculate the speedup of the parallel version with respect to itself rather than the sequential version, which can result in misleading performance numbers.)

Substituting Equation 8.1 into Equation 8.2 yields a formula for calculating speedup directly from running time, which is more convenient:

$$Speedup(N,K) = \frac{\dfrac{1}{T_{\text{par}}(N,K)}}{\dfrac{1}{T_{\text{seq}}(N,K)}} = \frac{T_{\text{seq}}(N,K)}{T_{\text{par}}(N,K)} \tag{8.3}$$

Ideally, a parallel program should run twice as fast on two processors as on one processor, three times as fast on three processors, four times as fast on four processors, and so on. Thus, ideally, *Speedup* should equal K. A plot of the ideal *Speedup* versus K is a straight line with unity slope, so this ideal speedup is also called a **linear speedup**.

As we will see, however, real parallel programs usually fall short of this ideal. **Efficiency** is a metric that captures how close to ideal a program's speedup is:

$$Eff(N,K) = \frac{Speedup(N,K)}{K} \tag{8.4}$$

An ideal parallel program has an efficiency of 1 for all problem sizes N and all numbers of processors K. A real parallel program typically has an efficiency less than 1, so that the speedup is less than K—a **sublinear speedup**. When designing parallel programs, we nonetheless strive to achieve the goal of a linear speedup.

The only way to know how closely we have achieved our goal is to run the program with a range of N and K values, measure T, and calculate the speedup and efficiency. However, it helps if we know what the speedup and efficiency are "supposed" to look like as a function of N and K. If the speedup and efficiency measurements don't look the way they're supposed to, the program's design might have a problem that must be fixed.

8.2 Amdahl's Law

Forty years ago, Gene Amdahl—the chief designer of IBM's System/360 family of mainframe computers, and who later founded his own company, Amdahl Corporation, to make IBM plug-compatible mainframes—published a short paper titled "Validity of the single processor approach to achieving large scale computing capabilities." In this paper, he compared two approaches for solving massive computational problems—the single-processor approach (sequential programs) and the then-newfangled multiple-processor approach (parallel programs). Amdahl's key insight was that a certain portion of any program must be executed sequentially—nowadays, we would say in a single thread. This portion consists of initialization, cleanup, thread synchronization, and similar housekeeping overhead, as well as I/O in some programs. This sequential portion can use only one processor, no matter how many processors are available in the parallel computer. Consequently, there is an upper bound on the speedup a parallel program can achieve.

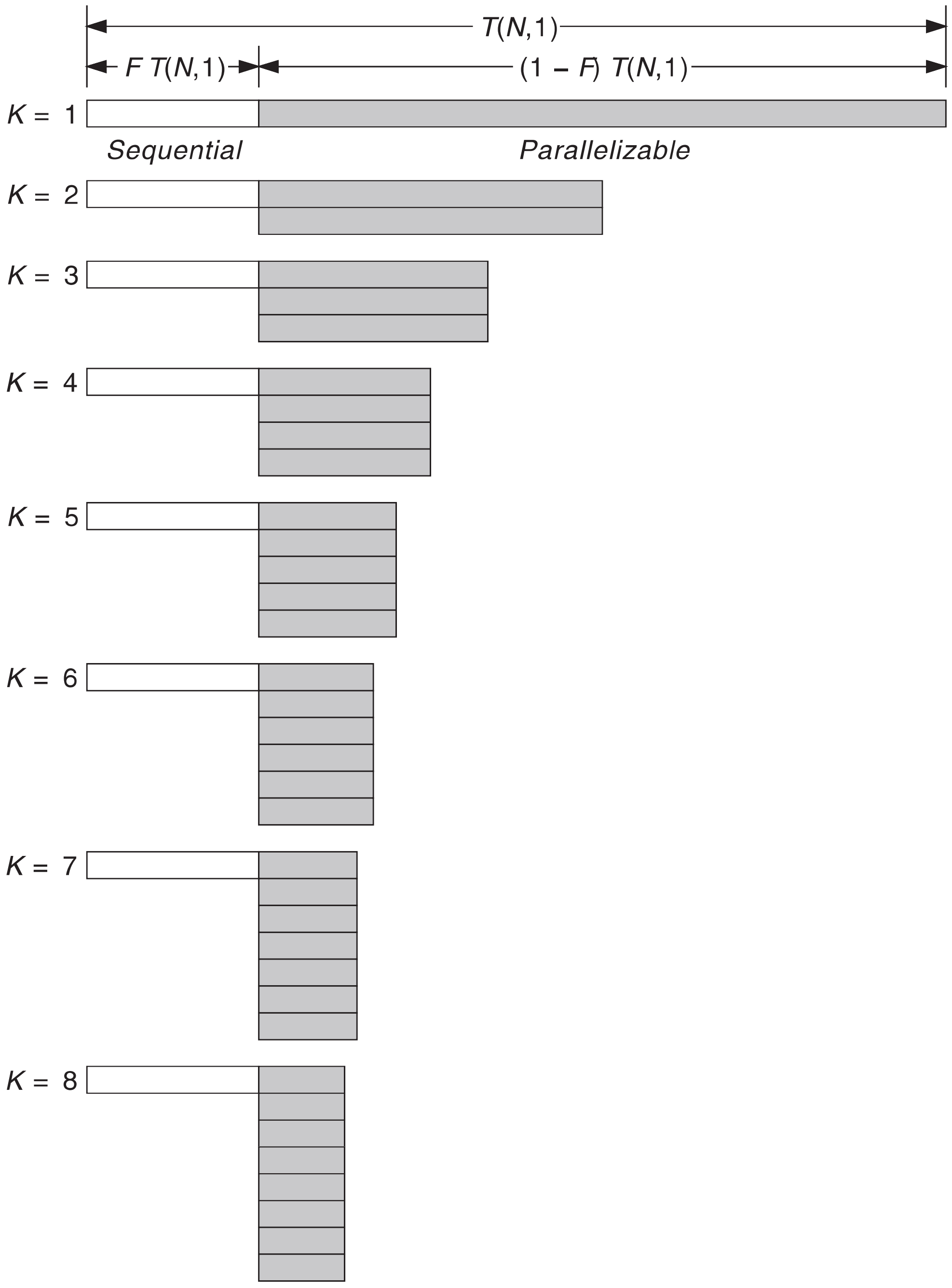

Figure 8.1 A parallel program with running time $T(N,1)$ and sequential fraction F executing with different numbers of processors K

Let the **sequential fraction** F, where $0 \le F \le 1$, be the fraction of a program that must be executed sequentially. If, for a given problem size N, a program's running time on a single processor is $T(N,1)$, then $F \cdot T(N,1)$ of the running time must be executed sequentially and $(1–F) \cdot T(N,1)$ can be executed in parallel. If the latter portion of the running time is split equally among K processors (Figure 8.1), then the parallel program's running time is:

$$T(N,K) = F \cdot T(N,1) + \frac{1}{K}(1-F) \cdot T(N,1) \tag{8.5}$$

Equation 8.5 is known as **Amdahl's Law**, although that moniker was bestowed by others, not by Amdahl.

From Amdahl's Law, we can derive equations for speedup and efficiency as a function of the sequential fraction:

$$\begin{aligned} Speedup(N,K) &= \frac{T(N,1)}{T(N,K)} \\ &= \frac{T(N,1)}{F \cdot T(N,1) + \frac{1}{K}(1-F) \cdot T(N,1)} \\ &= \frac{1}{F + \frac{1-F}{K}} \end{aligned} \tag{8.6}$$

$$Eff(N,K) = \frac{Speedup(N,K)}{K} = \frac{1}{KF + 1 - F} \tag{8.7}$$

Consider what happens to the speedup as the number of processors increases. In the limit as K goes to infinity, the speedup (Equation 8.6) goes to $1/F$. No matter how many processors we add, we will never achieve a speedup greater than the reciprocal of the program's sequential fraction. Furthermore, in the limit as K goes to infinity, the efficiency (Equation 8.7) goes to 0. As we add processors, the efficiency just gets worse and worse. Figures 8.2 and 8.3 plot speedup and efficiency from Amdahl's Law as a function of K for several values of F.

From Amdahl's Law, we can gain three important insights regarding parallel program design. The first insight is that a program's sequential fraction F has to be very small if we want to achieve good speedup and efficiency as we scale up the number of processors in our parallel computer. For example, Figure 8.3 shows that if we want an efficiency of 90 percent or better as we scale up to 100 processors, then we need a sequential fraction of 0.001 or less. In other words, no more than one-tenth of one percent of the running time is allowed to execute on a single processor, and all the rest has to execute in parallel. Sequential overhead in a parallel program severely reduces the program's performance, and when designing parallel programs, we go to great lengths, if necessary, to reduce F.

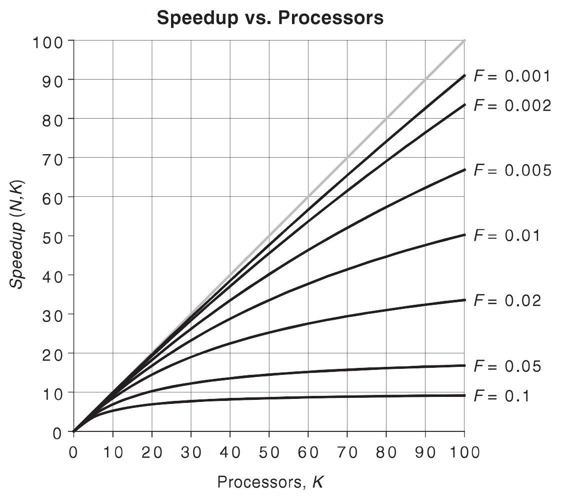

Figure 8.2 Speedup predicted by Amdahl's Law

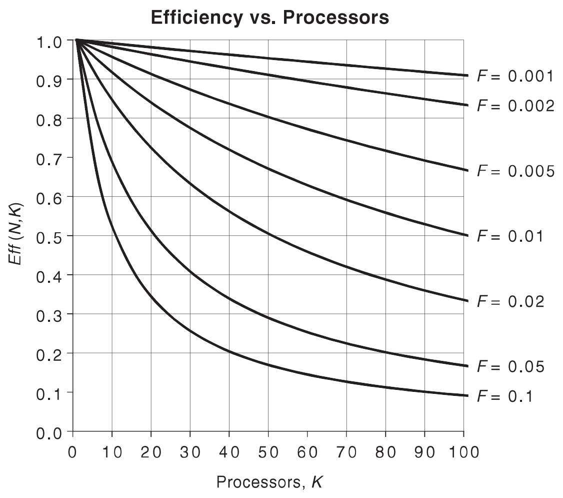

Figure 8.3 Efficiency predicted by Amdahl's Law

However, there is another way to get good performance as the number of processors scales up. Some 20 years after Amdahl's paper John Gustafson published a short paper titled "Reevaluating Amdahl's Law," in which he pointed out that if the problem size is also scaled up as the number of processors is scaled up—what we call a **sizeup**—the program can achieve a near-linear speedup, seemingly contradicting Amdahl's Law. We defer further discussion of Gustafson's observation until Chapter 10. For now, we simply note that Amdahl's Law is not the last word in parallel program performance.

The second insight from Amdahl's Law is the expected shape of the plot of a program's efficiency versus the number of processors. Figure 8.4 plots the efficiencies we expect to see when running a parallel program on an eight-processor SMP parallel computer. The measured efficiencies should start near 1 and decrease in roughly a straight line as K increases, unless F is relatively large. If the measured efficiency plots don't resemble Figure 8.4, we must figure out what's going on and change the program's design to fix the problem.

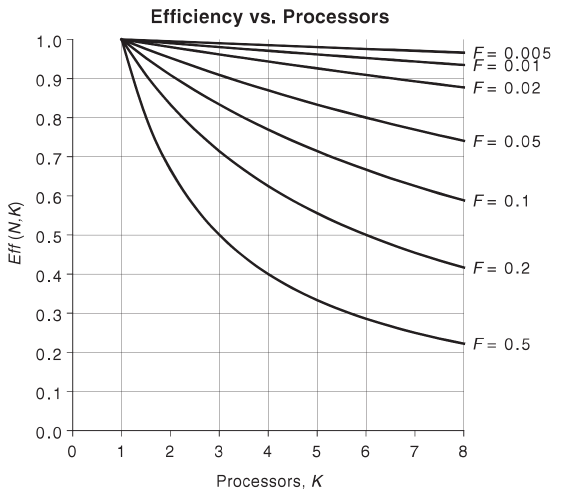

Figure 8.4 Efficiency for an 8-processor SMP parallel computer

Suppose we have measured a program's running time as a function of problem size and number of processors, *T*(*N*,*K*), and have plotted the program's efficiency. The plot looks more or less like Figure 8.4, with efficiency decreasing as *K* increases. But how can we tell whether the efficiency curves are behaving as Amdahl's Law predicts? The third insight from Amdahl's Law yields another metric we can use to analyze a program's performance. Rearranging Equation 8.5 gives a formula for *F* as a function of *T* and *K*:

$$F = \frac{K \cdot T(N,K) - T(N,1)}{K \cdot T(N,1) - T(N,1)} \tag{8.8}$$

Equation 8.8 lets us calculate *F* from the running-time measurements, namely *T*(*N*,1) and *T*(*N*,*K*), for the parallel program. When *F* is determined from the data in this way, it is called the **experimentally determined sequential fraction**, *EDSF*. (Note that we can only calculate *EDSF* for $K \ge 2$.)

In a 1990 paper titled "Measuring parallel processor performance," Alan Karp and Horace Flatt pointed out how the *EDSF* metric can help diagnose problems in a parallel program's performance. If the program is behaving as shown in Figure 8.1—a sequential portion with a fixed running time plus a parallel portion with a running time inversely proportional to *K*—then *F* should be a constant, and a plot of *EDSF* versus *K* should be a horizontal line. If the measured *EDSF* plot does not show up as roughly a horizontal line, it's another indication that something is going on that might require changing the program's design.

Armed with these insights, we are ready to start measuring and analyzing the AES key search program's speedup, efficiency, and *EDSF*.

8.3 Measuring Running Time

The AES key search program measures its own wall-clock running time using the `System.currentTimeMillis()` method. However, two program runs hardly ever yield the same running time measurements, even with identical inputs. There are several reasons for this. The system clock is not perfectly accurate; readings may vary a few tens of milliseconds on typical systems. Chiefly, however, other user programs and background processes running on the same computer take CPU time away from the parallel program, increasing the parallel program's wall-clock running time by an unpredictable amount on each run. We need a way to get a meaningful running time measurement for the program despite all these random fluctuations.

Faced with random errors in an experimental measurement, it's likely that your first thought is to take several readings and use their average. However, this is *not* the proper procedure for measuring a program's running time. To see why, we have to examine the often-unstated assumption behind the procedure of averaging a series of measurements.

An experimental reading of some quantity consists of the true value, known only to Mother Nature, plus a **measurement error** arising from the measuring apparatus. Mathematically, the measurement error is modeled as a random variable with some probability distribution. If we assume that *the measurement error obeys a Gaussian probability distribution with a mean of zero*, then taking the average of a series of measurements gives a close approximation to the true value. Figure 8.5 illustrates why. In this example, the true value of the running time being measured is *T* = 10,000 msec. If the measurement error obeys a Gaussian distribution with a mean of zero and a certain variance σ^2, the measurements obey a Gaussian distribution with a mean of 10,000 and the same variance (the bell-shaped curve). The black dots are several measurements, which are samples of the Gaussian distribution. The **sample mean** $\overline{T}$ is the average of the measurements. Probability theory tells us that with n measurements, $\overline{T}$ obeys a Gaussian distribution with the same mean as T, 10,000, but with a variance of only σ^2/n. Thus, $\overline{T}$ fluctuates around the true value less widely than the raw measurements, making $\overline{T}$ more likely to be closer to, hence a better estimator of, the true value than any of the raw measurements. In Figure 8.5, even though the measured T values fluctuate between about 5,000 and 17,000, the sample mean of 10,095.2 ends up quite close to the true T value.

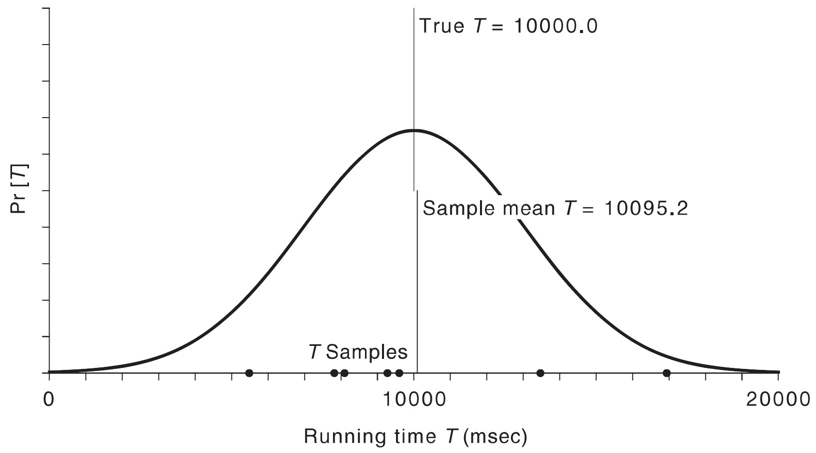

Figure 8.5 Running-time measurement with a Gaussian error distribution

The problem is that when measuring a program's wall-clock running time, *the measurement error distribution is not a zero-mean Gaussian.* A Gaussian distribution is symmetric; positive and negative errors both are equally possible. But the chief source of errors in the program's wall-clock running time does not have a symmetric distribution. When the operating system takes the CPU away from the parallel program to respond to an interrupt or to let another process run, the parallel program's running-time measurement can only increase, never decrease. The measurement-error probability distribution ends up looking more like Figure 8.6, which shows that the sample mean always ends up being larger than the true value. When the measurement error is always positive, the best estimator of the true value is the *minimum* of the measurements, not the average.

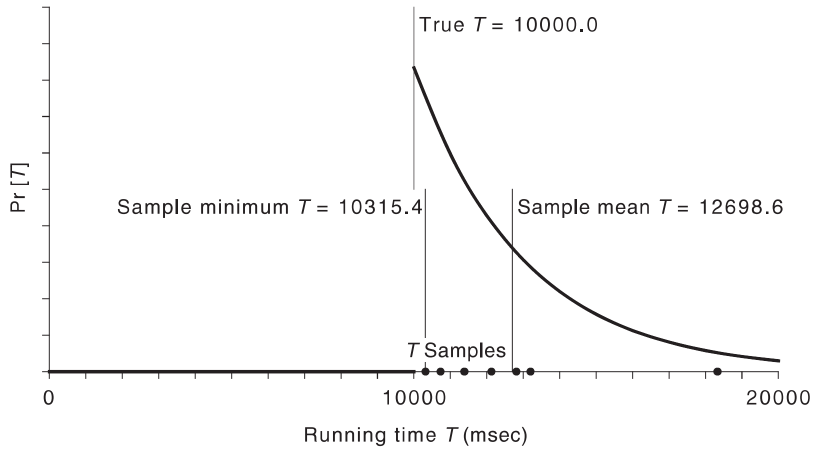

Figure 8.6 Running time measurement with a positive error distribution

Based on these considerations, here's the procedure we use in this book to measure our programs' running times:

1. Ensure that the parallel program is the only user process running. Don't let other users log in while timing measurements are being made. Don't run other programs such as editors, Web browsers, or e-mail clients.
2. To the extent possible, don't have any server or daemon processes running, such as Web servers, e-mail servers, file servers, network time daemons, and so on.
3. Prepare several input data sets, covering a range of problem sizes N. Choose the smallest problem size so that $T_{\text{seq}}(N,1)$ is at least 60 seconds.
4. For each N (each input data set), run the sequential version of the program seven times. (The number of runs, seven, is just an arbitrary choice.) Take the smallest of the running times as the measured $T_{\text{seq}}(N,1)$ value.
5. For each N, and for each K from 1 up to the number of available processors, run the parallel version of the program seven times. Take the smallest of the running times as the measured $T_{\text{par}}(N,K)$ value.

There are two reasons for measuring only problem sizes whose running times are 60 seconds or more. First, if we're willing to wait a certain amount of time for the program to finish before losing patience, and the program takes less time than this "impatience threshold" even on one processor, then there's no point in trying to reduce the running time by going to multiple processors. For this book, we arbitrarily peg the impatience threshold at 60 seconds. Second, when a Java program starts, the JVM needs some time to "warm up"—to load and verify the class files, to detect hot spots, and to run the JIT compiler to compile the hot spots' bytecode to machine instructions. This warm-up overhead contributes to the program's sequential portion. Running the program for a longer time reduces the sequential fraction due to JVM warm-up.

8.4 FindKeySmp Running Time Measurements

Table 8.1 (at the end of the chapter) gives the running-time measurements in milliseconds for the AES key search program, as well as the speedups, efficiencies, and *EDSFs* calculated from the running times. The programs were run on a computer named "parasite," a Sun Microsystems eight-processor SMP parallel computer with four UltraSPARC-IV dual-core CPU chips, a 1.35 GHz CPU clock speed, and 16 GB of main memory. The six input data sets had from $n = 24$ to 29 missing key bits, yielding problem sizes of $N = 16\text{M}$, 32M, 64M, 128M, 256M, and 512M encryption keys tested. ("M" stands for 2^{20}.) In the K column, "seq" denotes the sequential version of the program (FindKeySeq) and 1 through 8 denote the parallel version of the program (FindKeySmp). The T column lists the smallest running time among the seven program runs.

Figure 8.7 plots running time versus number of processors for the AES key search program. Note that this is a **log-log plot**—both axes use a logarithmic scale rather than a linear scale. (See Appendix C for further information about log-log plots.) The logarithmic scale is better suited to plotting data that spans many orders of magnitude, as does the running time data. Furthermore, the log-log plot lets us eyeball whether the parallel program is achieving ideal performance. Ideally, *T* should be proportional to K^{-1}. This would show up on a log-log plot as a straight line with a slope of –1. If the *T* versus *K* curves don't look like that, the program's performance is not ideal. As we can see, the AES key search program's performance is less than ideal, especially for larger numbers of processors.

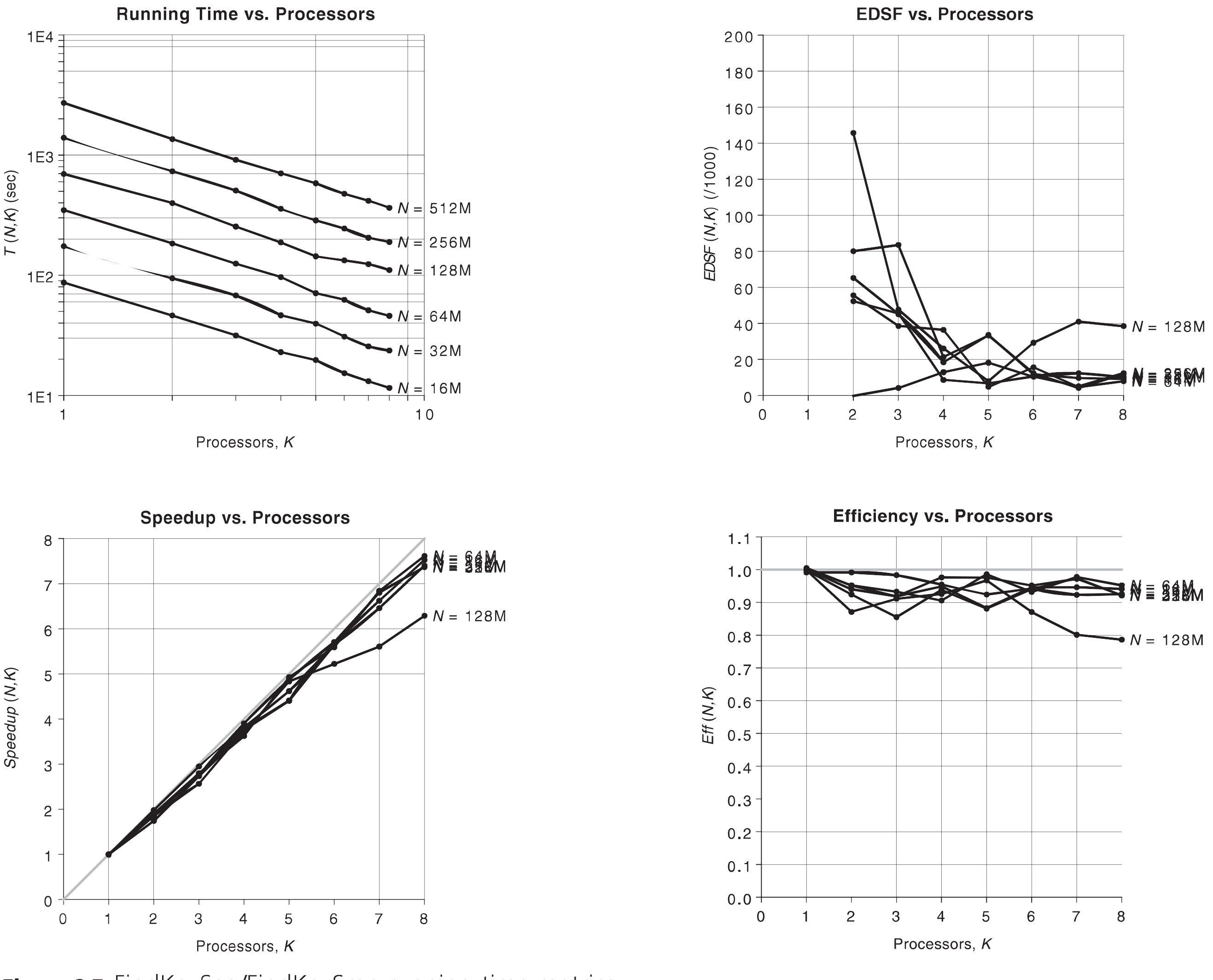

Figure 8.7 FindKeySeq/FindKeySmp running-time metrics

Figure 8.7 also plots speedup, efficiency, and *EDSF* versus number of processors for the AES key search program. These curves reinforce what we saw in the running-time curves. For smaller numbers of processors, we are getting close to linear speedups and efficiencies close to 1, but for larger numbers of processors, the speedup and efficiency curves start jumping around. The efficiency curves in particular do not look anything like what we would expect from Figure 8.4. The *EDSF* curves are all over the map, and are nowhere near constant. Something is going on in the parallel program that must be fixed. This will be the topic of Chapter 9.

By the way, the running-time metric plots in Figure 8.7 were produced by a Java program, class Speedup, in the Parallel Java Library. The program takes the raw running-time measurements as in Table 8.1, calculates and prints the running time, speedup, efficiency, and *EDSF* metrics, and generates the plots. On each plot, the program labels each curve with the corresponding problem size (N = 16M, N = 32M, and so on). If the curves fall on top of each other—as often happens with the speedup and efficiency curves—then the labels fall on top of each other as well. Refer to the Parallel Java documentation for instructions on how to use the Speedup program.

8.5 For Further Information

Amdahl's original paper on parallel program performance:

- G. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the AFIPS Spring Joint Computer Conference*, 1967, pages 483–485.

Gustafson's original paper on parallel program performance:

- J. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, May 1988.

A critique of Amdahl's and Gustafson's work:

- Y. Shi. Reevaluating Amdahl's law and Gustafson's law. October 1996. http://joda.cis.temple.edu/~shi/docs/amdahl/amdahl.html

Karp's and Flatt's original paper on parallel program performance:

- A. Karp and H. Flatt. Measuring parallel processor performance. *Communications of the ACM*, 33(5):539–543, May 1990.

Table 8.1 FindKeySeq/FindKeySmp running-time metrics

N	*K*	*T*	*Spdup*	*Eff*	*EDSF*	*N*	*K*	*T*	*Spdup*	*Eff*	*EDSF*
16M	seq	87026				128M	seq	694908			
16M	1	86914	1.001	1.001		128M	1	696352	0.998	0.998	
16M	2	46291	1.880	0.940	0.065	128M	2	398902	1.742	0.871	0.146
16M	3	31581	2.756	0.919	0.045	128M	3	254240	2.733	0.911	0.048
16M	4	22931	3.795	0.949	0.018	128M	4	187697	3.702	0.926	0.026
16M	5	19715	4.414	0.883	0.034	128M	5	143717	4.835	0.967	0.008
16M	6	15333	5.676	0.946	0.012	128M	6	133041	5.223	0.871	0.029
16M	7	13138	6.624	0.946	0.010	128M	7	123930	5.607	0.801	0.041
16M	8	11561	7.528	0.941	0.009	128M	8	110477	6.290	0.786	0.038
32M	seq	174282				256M	seq	1396225			
32M	1	174621	0.998	0.998		256M	1	1393962	1.002	1.002	
32M	2	94301	1.848	0.924	0.080	256M	2	733449	1.904	0.952	0.052
32M	3	67931	2.566	0.855	0.084	256M	3	506964	2.754	0.918	0.046
32M	4	46449	3.752	0.938	0.021	256M	4	357525	3.905	0.976	0.009
32M	5	39578	4.404	0.881	0.033	256M	5	286268	4.877	0.975	0.007
32M	6	30895	5.641	0.940	0.012	256M	6	244600	5.708	0.951	0.011
32M	7	25567	6.817	0.974	0.004	256M	7	205257	6.802	0.972	0.005
32M	8	23649	7.370	0.921	0.012	256M	8	189344	7.374	0.922	0.012
64M	seq	349907				512M	seq	2693607			
64M	1	348290	1.005	1.005		512M	1	2717345	0.991	0.991	
64M	2	183820	1.904	0.952	0.056	512M	2	1358338	1.983	0.992	0.000
64M	3	125050	2.798	0.933	0.039	512M	3	913330	2.949	0.983	0.004
64M	4	96578	3.623	0.906	0.036	512M	4	705661	3.817	0.954	0.013
64M	5	70986	4.929	0.986	0.005	512M	5	582974	4.620	0.924	0.018
64M	6	62586	5.591	0.932	0.016	512M	6	476279	5.656	0.943	0.010
64M	7	51142	6.842	0.977	0.005	512M	7	416946	6.460	0.923	0.012
64M	8	45945	7.616	0.952	0.008	512M	8	363923	7.402	0.925	0.010