one widget appears every time unit. Hence the second widget appears at time unit four, the third widget at time unit five, and so on.

Figure 1-4c shows a group of three data-parallel widget-assembly machines. Each machine performs every subassembly task, as does the sequential widget assembler. Throughput is increased by replicating machines. Another three widgets appear every three time units. Note that the time needed to produce four widgets is the same as the time needed to produce five or six widgets.

Figure 1-5 illustrates the speedup achieved by the pipelined and data-parallel widget machines. The x axis represents the number of widgets assembled; the y axis represents the speedup achieved. For any particular number of widgets i, the speedup is computed by dividing the time needed for the sequential machine to assemble i widgets by the time needed for the pipelined or data-parallel machine to assemble i widgets. For example, the sequential machine requires 12 time units to assemble four widgets, while the pipelined machine requires six time units to assemble four widgets. Hence the pipelined machine exhibits a speedup of two for the task of assembling four widgets.

1.3.2 Control Parallelism

Our discussion has focused on data-parallel and pipelined algorithms. Pipelining is actually a special case of a more general class of parallel algorithms, called control-parallel algorithms. In contrast to data parallelism, in which parallelism is achieved by applying a single operation to a data set, **control parallelism** is achieved by applying different operations to different data elements simultaneously. The flow of data among these processes can be arbitrarily complex. If the data-flow graph forms a simple directed path, then we say the algorithm is pipelined.

Most realistic problems can exploit both data parallelism and control parallelism. Realistic problems also have some precedence relations between different tasks. For example, consider the problem of performing an estate's weekly landscape maintenance as quickly as possible (Fig. 1-6). Suppose four chores must be performed: mowing the lawn, edging the lawn, checking the sprinklers, and weeding the flower beds. With the exception of checking the sprinklers,

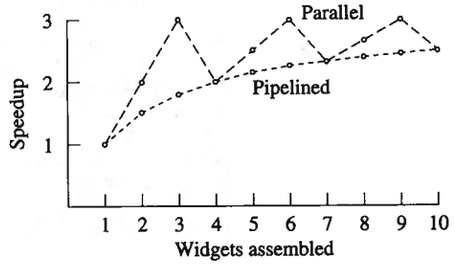


FIGURE 1-5 Speedup achieved by pipelined and parallel widget-assembly machines. Note that speedup is graphed as a function of problem size (number of widgets assembled). This is unusual. Speedup is typically graphed as a function of number of processors used.

FIGURE 1-6 Most realistic problems have data parallelism, control parallelism, and precedence constraints between tasks. In this example, there are four tasks to complete. Three of the tasks—mowing the lawn, edging the lawn, and weeding the garden—may be undertaken simultaneously, an example of control parallelism. Each of these tasks viewed in isolation is data parallel. Finally, all three tasks must be finished before the fourth task—checking the sprinklers—can begin. Since everyone must be finished before the sprinklers are checked, any one of the employees performing the other tasks can also check the sprinklers.

SPEEDY LANDSCAPE, INC.
Work Assignments—Medici Manor

Mow lawn	**Edge lawn**
Allan	Francis
Bernice	Georgia
Charlene	**Weed garden**
Dominic	Hillary
Edward	Irene
Check sprinklers	Jose
Allan	

which is easily performed by a single person, each of the remaining chores can be done more quickly by multiple workers. Increasing the lawn-mowing speed by creating a lawn-mowing team and assigning each team member a portion of the lawn is an example of data parallelism. Since there is no reason why the flower beds cannot be weeded at the same time the lawn is being mowed, we can assign another team to the weeding. Concurrent weeding and lawn mowing is an example of control parallelism. Precedence relations exist between checking the sprinklers and the three other tasks, since all of the other tasks must be completed before the sprinklers are tested.

1.3.3 Scalability

An algorithm is **scalable** if the level of parallelism increases at least linearly with the problem size. An architecture is scalable if it continues to yield the same performance per processor, albeit used on a larger problem size, as the number of processors increases. Algorithmic and architectural scalability are important, because they allow a user to solve larger problems in the same amount of time by buying a parallel computer with more processors.

Data-parallel algorithms are more scalable than control-parallel algorithms, because the level of control parallelism is usually a constant, independent of the problem size, while the level of data parallelism is an increasing function of the problem size. Almost every problem we will study in this book has a data-parallel solution.

1.4 THE SIEVE OF ERATOSTHENES

In this section we will explore methods to parallelize the Sieve of Eratosthenes, the classic prime-finding algorithm. We will design and analyze both control-parallel and data-parallel implementations of this algorithm.

We want to find the number of primes less than or equal to some positive integer n. A prime number has exactly two factors: itself and 1. The Sieve of

Eratosthenes begins with a list of natural numbers 2, 3, 4, ..., n, and removes composite numbers from the list by striking multiples of 2, 3, 5, and successive primes (see Fig. 1-7). The sieve terminates after multiples of the largest prime less than or equal to $\sqrt{n}$ have been struck.

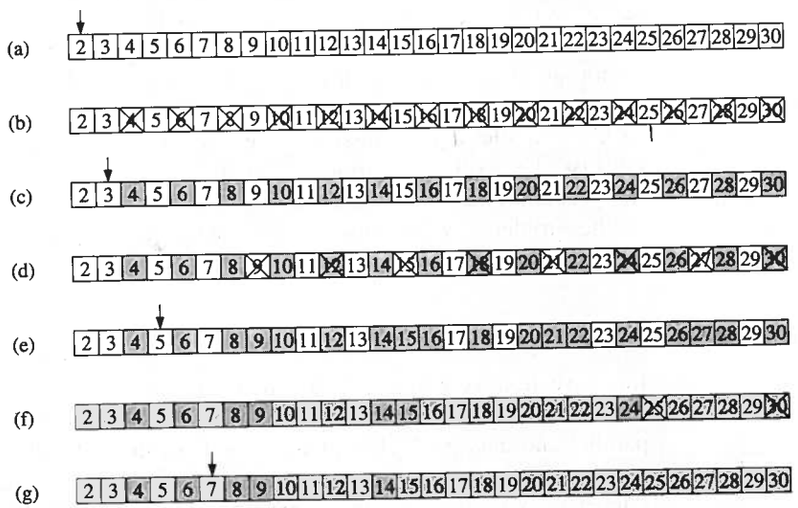
Note: The Sieve of Eratosthenes is impractical for testing the primality of "interesting" numbers—those with hundreds of digits—because the time complexity of the algorithm is $\Omega(n)$, and n increases exponentially with the number of digits. However, more reasonable factoring algorithms make use of sieve techniques in other ways.

A sequential implementation of the Sieve of Eratosthenes manages three key data structures: a boolean array whose elements correspond to the natural numbers being sieved, an integer corresponding to latest prime number found, and an integer used as a loop index incremented as multiples of the current prime are marked as composite numbers (Fig. 1-8a).

1.4.1 Control-Parallel Approach

First let's examine a control-parallel algorithm to find the number of primes less than or equal to some positive integer n. In this algorithm every processor

FIGURE 1-7 Use of the Sieve of Eratosthenes to find all primes less than or equal to 30. (a) Prime is next unmarked natural number—2. (b) Strike all multiples of 2, beginning with 2^2. (c) Prime is next unmarked natural number—3. (d) Strike all multiples of 3, beginning with 3^2. (e) Prime is next unmarked natural number—5. (f) Strike all multiples of 5, beginning with 5^2. (g) Prime is next unmarked natural number—7. Since 7^2 is greater than 30, algorithm terminates. All remaining unmarked natural numbers are also primes.



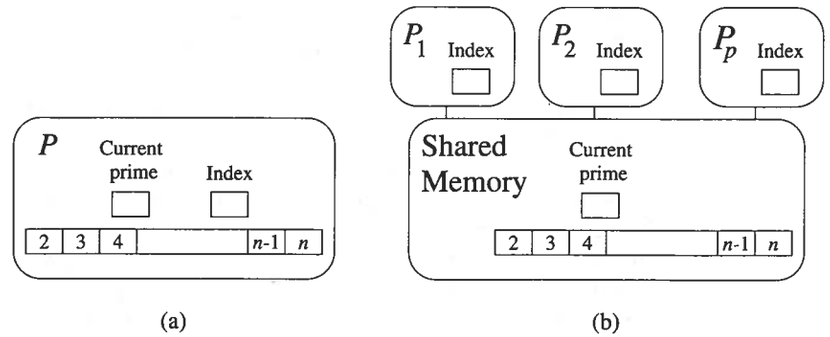


FIGURE 1-8 Shared memory model for parallel Sieve of Eratosthenes algorithm. (a) Sequential algorithm maintains array of natural numbers, variable storing current prime, and variable storing index of loop iterating through array of natural numbers. (b) In parallel model each processor has its own private loop index and shares access to other variables with all processors.

repeatedly goes through the two-step process of finding the next prime number and striking from the list multiples of that prime, beginning with its square. The processors continue until a prime is found whose value is greater than $\sqrt{n}$. Using this approach, processors concurrently mark multiples of different primes. For example, one processor will be responsible for marking multiples of 2 beginning with 4. While this processor marks multiples of 2, another may be marking multiples of 3 beginning with 9.

We will base the control-parallel algorithm on the simple model of parallel computation illustrated in Fig. 1-8b. Every processor shares access to the boolean array representing the natural numbers, as well as the integer containing the value of the latest prime number found. Because processors independently mark multiples of different primes, each processor has its own local loop index.

If a group of asynchronously executing processors share access to the same data structure in an unstructured way, inefficiencies or errors may occur. Here are two problems that can occur in the algorithm we just described. First, two processors may end up using the same prime value to sieve through the array. Normally a processor accesses the value of the current prime and begins searching at the next array location until it finds another unmarked cell, which corresponds to the next prime. Then it updates the value of the integer containing the current prime. If a second processor accesses the value of the current prime before the first processor updates it, then both processors will end up finding the same new prime and performing a sieve based on that value. This does not make the algorithm incorrect, but it wastes time.

Second, a processor may end up sieving multiples of a composite number. For example, assume processor A is responsible for marking multiples of 2, but before it can mark any cells, processor B finds the next prime to be 3, and processor C searches for the next unmarked cell. Because cell 4 has not yet been

marked, processor C returns with the value 4 as the latest "prime" number. As in the previous example, the algorithm is still correct, but a processor sieving multiples of 4 is wasting time.

In later chapters we will discuss ways to design parallel algorithms that avoid such problems.

For now, let's explore the maximum speedup achievable by this parallel algorithm, assuming that none of the time-wasting problems described earlier happen. To make our analysis easier, we'll also ignore the time spent finding the next prime and concentrate on the operation of marking cells.

First let's consider the time taken by the sequential algorithm. Assume it takes 1 unit of time for a processor to mark a cell. Suppose there are k primes less than or equal to $\sqrt{n}$. We denote these primes $\pi_1, \pi_2, \ldots, \pi_k$. (For example, $\pi_1 = 2$, $\pi_2 = 3$, and $\pi_3 = 5$.) The total amount of time a single processor spends striking out composite numbers is

$$\left\lceil \frac{(n+1)-\pi_1^2}{\pi_1} \right\rceil + \left\lceil \frac{(n+1)-\pi_2^2}{\pi_2} \right\rceil + \left\lceil \frac{(n+1)-\pi_3^2}{\pi_3} \right\rceil + \cdots + \left\lceil \frac{(n+1)-\pi_k^2}{\pi_k} \right\rceil$$

$$= \left\lceil \frac{n-3}{2} \right\rceil + \left\lceil \frac{n-8}{3} \right\rceil + \left\lceil \frac{n-24}{5} \right\rceil + \cdots + \left\lceil \frac{(n+1)-\pi_k^2}{\pi_k} \right\rceil$$

There are $\lceil (n-3)/2 \rceil$ multiples of 2 in the range 4 through n, $\lceil (n-8)/3 \rceil$ multiples of 3 in the range 9 through n, and so on. For $n = 1{,}000$ the sum is 1,411.

Now let's think about the time taken by the parallel algorithm. Whenever a processor is unoccupied, it grabs the next prime and marks its multiples. All processors continue in this fashion until the first prime greater than $\sqrt{n}$ is found. For example, Fig. 1-9 illustrates the time required by one, two, and three processors to find all primes less than or equal to 1,000. With two processors the parallel algorithm has speedup 2 (1,411/706). With three processors the parallel algorithm has speedup 2.83 (1,411/499). It is clear that the parallel execution time will not decrease if more than three processors are used, because with three or more processors the time needed for a single processor to sieve all multiples of 2 determines the parallel execution time. Hence an upper bound on the execution time of the parallel algorithm for $n = 1{,}000$ is 2.83.

Increasing n does not significantly raise the upper bound on speedup imposed by a single processor striking all multiples of 2 (see Prob. 1-10).

1.4.2 Data-Parallel Approach

Let's consider another approach to parallelizing the Sieve of Eratosthenes. In our new algorithm, processors will work together to strike multiples of each newly found prime. Every processor will be responsible for a segment of the

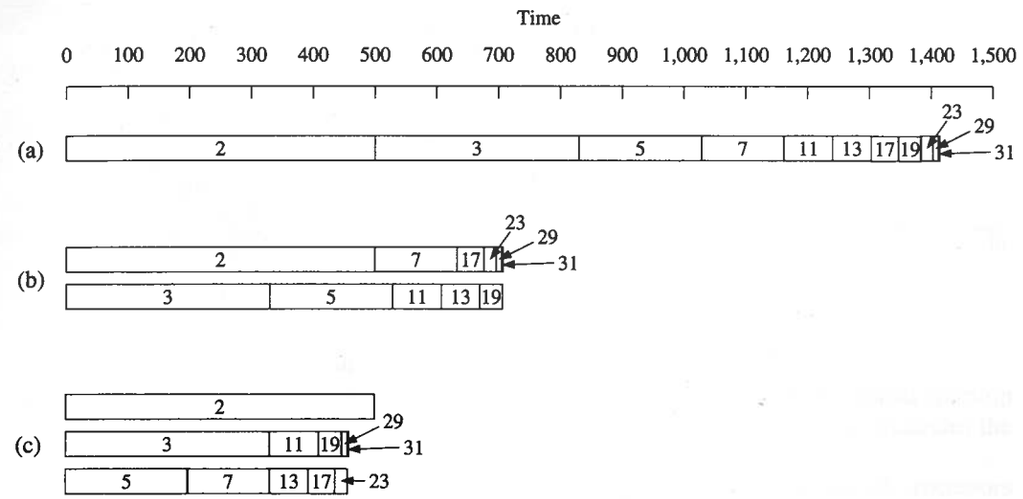


FIGURE 1-9 Study of how adding processors reduces the execution time of the control-parallel Sieve of Eratosthenes algorithm when $n = 1{,}000$. The number in the bar represents the prime whose multiples are being marked. The length of the bar is the time needed to strike these multiples. (a) Single processor strikes out all composite numbers in 1,411 units of time. (b) With two processors execution time drops to 706 time units. (c) With three or more processors execution time is 499 time units, the time needed for a processor to strike all multiples of 2.

array representing the natural numbers. The algorithm is data parallel, because each processor applies the same operation (striking multiples of a particular prime) to its own portion of the data set.

Analyzing the speedup achievable by the data-parallel algorithm on the shared memory model of Fig. 1-8b is straightforward; we have left it as an exercise. Instead, we will consider a different model of parallel computation (Fig. 1-10). In this model there is no shared memory, and processor interaction occurs through message passing.

Assume we are solving the problem on p processors. Every processor is assigned no more than $\lceil n/p \rceil$ natural numbers. We will also assume that p is much less than $\sqrt{n}$. In this case all primes less than $\sqrt{n}$, as well as the first prime greater than $\sqrt{n}$, are in the list of natural numbers controlled by the first processor. Processor 1 will find the next prime and broadcast its value to the other processors. Then all processors strike from their lists of composite numbers all multiples of the newly found prime. This process of prime finding and composite number striking continues until the first processor reaches a prime greater than $\sqrt{n}$, at which point the algorithm terminates.

Let's estimate the execution time of this parallel algorithm. As in the previous analysis, we ignore time spent finding the next prime and focus on the time spent marking composite numbers. However, since this model does not

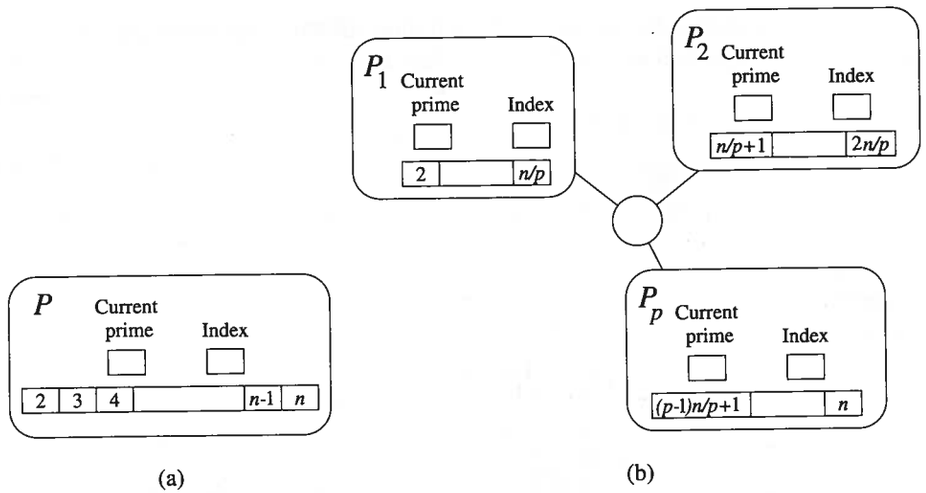


FIGURE 1-10 Private memory model for parallel Sieve of Eratosthenes algorithm. (a) Sequential algorithm maintains array of natural numbers, variable storing current prime, and variable storing index of loop iterating through array of natural numbers. (b) In parallel model each processor has its own copy of the variables containing the current prime and the loop index. Processor 1 finds primes and communicates them to the other processors. Each processor iterates through its own portion of the array of natural numbers, marking multiples of the prime.

have a shared memory, we must also consider the time spent communicating the value of the current prime from processor 1 to all other processors.

Assume it takes χ time units for a processor to mark a multiple of a prime as being a composite number. Suppose there are k primes less than or equal to $\sqrt{n}$. We denote these primes $\pi_1, \pi_2, \ldots, \pi_k$. The total amount of time a processor spends striking out composite numbers is no greater than

$$\left(\left\lceil \frac{\lceil n/p \rceil}{2} \right\rceil + \left\lceil \frac{\lceil n/p \rceil}{3} \right\rceil + \left\lceil \frac{\lceil n/p \rceil}{5} \right\rceil + \cdots + \left\lceil \frac{\lceil n/p \rceil}{\pi_k} \right\rceil\right)\chi$$

Assume every time processor 1 finds a new prime it communicates that value to each of the other $p-1$ processors in turn. If processor 1 spends λ time units each time it passes a number to another processor, its total communication time for all k primes is $k(p-1)\lambda$.

To bring this discussion down to earth, suppose we want to count the number of primes less than 1,000,000. It turns out that there are 168 primes less than 1,000, the square root of 1,000,000. The largest of these is 997. The maximum possible execution time spent striking out primes is

$$\left(\left\lceil \frac{\lceil 1{,}000{,}000/p \rceil}{2} \right\rceil + \left\lceil \frac{\lceil 1{,}000{,}000/p \rceil}{3} \right\rceil + \cdots + \left\lceil \frac{\lceil 1{,}000{,}000/p \rceil}{997} \right\rceil\right)\chi$$

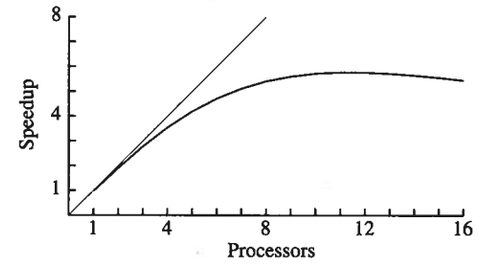


FIGURE 1-11 Estimated speedup of the data-parallel Sieve of Eratosthenes algorithm, assuming that $n = 1{,}000{,}000$ and $\lambda = 100\chi$. Note that speedup is graphed as a function of number of processors used. This is typical.

The total communication time is $168(p-1)\lambda$.

If we know the relation between χ and λ, we can plot an estimated speedup curve for the parallel algorithm. Suppose $\lambda = 100\chi$. Figure 1-11 illustrates the estimated speedup of the data-parallel algorithm.

Notice that speedup is not directly proportional to the number of processors used. In fact, speedup is highest at 11 processors. When more processors are added, speedup declines. Figure 1-12 illustrates the estimated total execution time of the parallel algorithm along with its two components: computation time and communication time. Computation time is inversely proportional to the number of processors used, while communication time increases linearly with the number of processors used. After 11 processors, the increase in communication time is greater than the decrease in computation time, and the total execution time begins to increase.

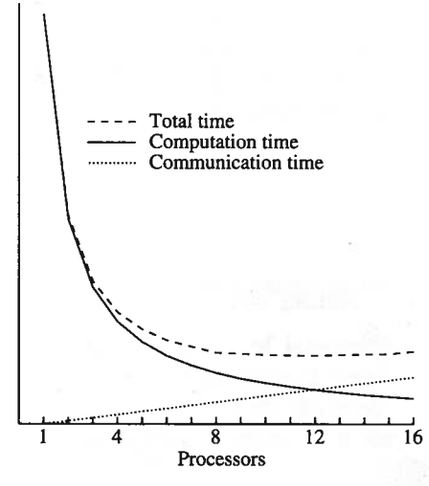


FIGURE 1-12 Total execution time of the data-parallel Sieve of Eratosthenes algorithm is the sum of the time spent computing and the time spent communicating. Computation time is inversely proportional to the number of processors; communication time is directly proportional to the number of processors.

1.4.3 Data-Parallel Approach with I/O

The prime-finding algorithms we have described are unrealistic, because they terminate without storing their results. Let's examine what happens when we execute a data-parallel implementation of the Sieve of Eratosthenes incorporating output on the shared-memory model of parallel computation.

The augmented shared memory model appears in Fig. 1-13. Assume only one processor at a time can access the I/O device. Let $i\beta$ denote the time needed for a processor to transmit i prime numbers to that device.

Let's predict the speedup achieved by the data-parallel algorithm that outputs all primes to the I/O device. Suppose $n = 1{,}000{,}000$. There are 78,498 primes less than 1,000,000. To find the total execution time, we take the total computation time,

$$\left(\left\lceil \frac{\lceil 1{,}000{,}000/p \rceil}{2} \right\rceil + \left\lceil \frac{\lceil 1{,}000{,}000/p \rceil}{3} \right\rceil + \cdots + \left\lceil \frac{\lceil 1{,}000{,}000/p \rceil}{997} \right\rceil\right)\chi$$

and add to it the total I/O time, $78{,}498\beta$.

The solid curve in Fig. 1-14 illustrates the expected speedup of this parallel algorithm for $1, 2, \ldots, 32$ processors, assuming that $\beta = \chi$. Figure 1-15

FIGURE 1-13 A shared-memory parallel model incorporating a sequential I/O device.

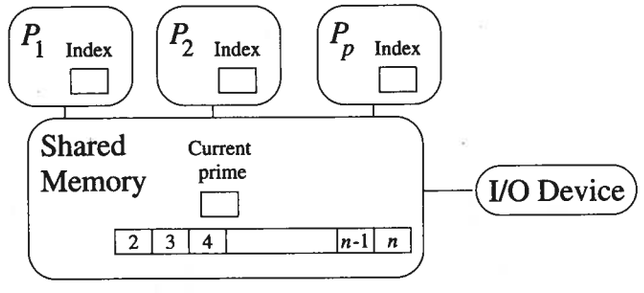


FIGURE 1-14 Expected speedup of data-parallel Sieve of Eratosthenes algorithm that outputs primes to an I/O device. Solid curve is speedup predicted from analysis. Dashed curve is maximum speedup as determined by Amdahl's law. This graph is for the case when $n = 1{,}000{,}000$ and $\beta=\chi$.

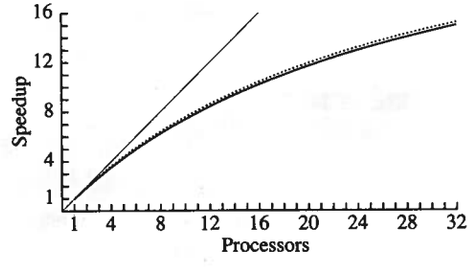
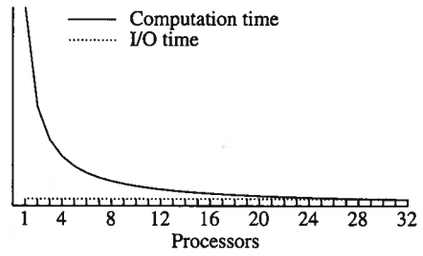


FIGURE 1-15 Total execution time of data-parallel output-producing Sieve of Eratosthenes algorithm as a function of its two components, computation time and I/O time. This graph is for the case when $n = 1{,}000{,}000$ and $\beta=\chi$.



illustrates the two components of the parallel execution time: computation time and output time. Because output to the I/O device must be performed sequentially, it puts a damper on the speedup achievable through parallelization. *Amdahl's law* is a way of expressing maximum speedup as a function of the amount of parallelism and the fraction of the computation that is inherently sequential.

Amdahl's law (Amdahl 1967). Let f be the fraction of operations in a computation that must be performed sequentially, where $0 \leq f \leq 1$. The maximum speedup S achievable by a parallel computer with p processors performing the computation is

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

For example, consider the algorithm we have just explored. When $n = 1{,}000{,}000$, the sequential algorithm marks 2,122,048 cells and outputs 78,498 primes. Assuming these two kinds of operations take the same amount of time, the total sequential time is 2,200,546, and $f = 78{,}498/2{,}200{,}546 = .0357$. An upper bound on the speedup achievable by a parallel computer with p processors is

$$\frac{1}{.0357 + .9643/p}$$

The dotted curve in Fig. 1-14 plots the upper bound on the speedup of the algorithm as predicted by Amdahl's law.

Often, as the size of the problem increases, the fraction f of inherently sequential operations decreases, making the problem more amenable to parallelization. This phenomenon—called the **Amdahl effect**—is true for the application we have been considering. Figure 1-16 plots f as a function of n for the data-parallel sieve algorithm with output, assuming $\beta = \chi$.