```
/* This program computes pi using the rectangle rule. */

#define INTERVALS 1000000

int main (int argc, char *argv[])
{
   double area;  /* Area under curve */
   double ysum;  /* Sum of rectangle heights */
   double xi;    /* Midpoint of interval */
   int i;

   ysum = 0.0;
   for (i = 0; i < INTERVALS; i++) {
      xi = (1.0/INTERVALS)*(i+0.5);
      ysum += 4.0/(1.0+xi*xi);
   }
   area = ysum * (1.0 / INTERVALS);
   printf ("Area is %13.11f\n", area);
   return 0;
}
```

Figure 4.8 A C program to compute the value of π using the rectangle rule.

```
/* This program uses Simpson's Rule to compute pi. */

#define n 50

double f (int i) {
   double x;
   x = (double) i / (double) n;
   return 4.0 / (1.0 + x * x);
}

int main (int argc, char *argv[]) {
   double area;
   int i;
   area = f(0) - f(n);
   for (i = 1; i <= n/2; i++)
      area += 4.0*f(2*i-1) + 2*f(2*i);
   area /= (3.0 * n);
   printf ("Approximation of pi: %13.11f\n", area);
   return 0;
}
```

Figure 4.9 A C program to compute the value of π using Simpson's Rule.

CHAPTER 5

The Sieve of Eratosthenes

He was not merely a chip of the old block, but the old block itself.
Edmund Burke

5.1 INTRODUCTION

The Sieve of Eratosthenes is a useful vehicle for advancing to the next level of parallel programming with MPI. After an explanation of the sequential algorithm, we will use the domain decomposition methodology to come up with a data-parallel algorithm. During the task agglomeration step we will weigh the pros and cons of several schemes to allocate contiguous blocks of array elements to tasks. The resulting algorithm requires a broadcast step, and we will learn the syntax of an MPI function to perform the broadcast.

After coding and benchmarking an initial parallel program, we will consider three ways to improve its performance, including using redundant computations to reduce process communication time and rearranging the order of computations to increase the cache hit rate. Benchmarking these program improvements highlights the importance of maximizing single-processor performance, even when multiple processors are available.

This chapter introduces the following MPI function:

`MPI_Bcast`, to broadcast a message to all processes in a communicator

5.2 SEQUENTIAL ALGORITHM

Our goal is to develop a parallel version of the prime sieve invented by the Greek mathematician Eratosthenes (276-194 BCE). You can find pseudocode for the Sieve of Eratosthenes in Figure 5.1.

An example of the sieve appears in Figure 5.2. In order to find primes up to 60, integer multiples of the primes 2, 3, 5, and 7 are marked as composite

```
1. Create a list of natural numbers 2, 3, 4, ..., $n$, none of which is marked.
2. Set $k$ to 2, the first unmarked number on the list.
3. Repeat
   (a) Mark all multiples of $k$ between $k^2$ and $n$
   (b) Find the smallest number greater than $k$ that is unmarked. Set $k$ to this new value.
   Until $k^2 > n$
4. The unmarked numbers are primes.
```

Figure 5.1 The Sieve of Eratosthenes finds primes between 2 and *n*.

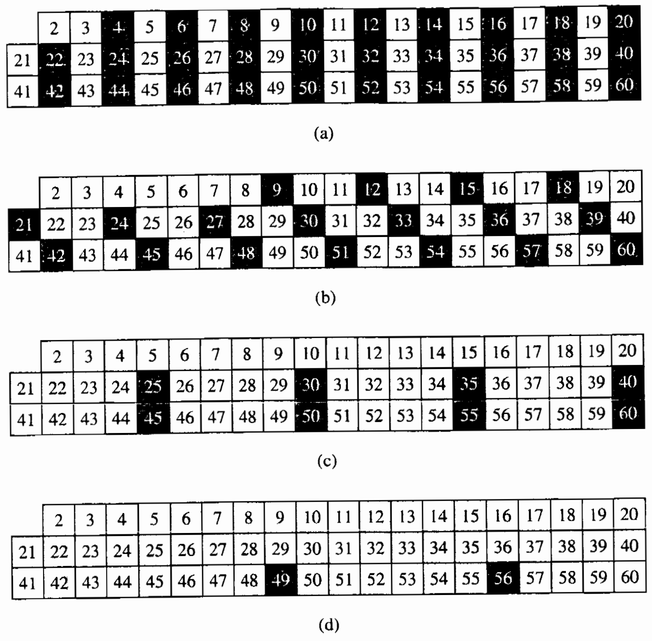

Figure 5.2 The Sieve of Eratosthenes. In this example we are finding all primes less than or equal to 60. (a) Mark all multiples of 2 between 4 and 60, inclusive. (b) The next unmarked value is 3. Mark all multiples of 3 between 9 and 60, inclusive. (c) The next unmarked value is 5. Mark all multiples of 5 between 25 and 60, inclusive. (d) The next unmarked value is 7. Mark all multiples of 7 between 49 and 60. The next unmarked value is 11. Since the square of 11 is 121, and 121 is greater than 60, the algorithm terminates. All remaining unmarked cells represent prime numbers.

numbers. The next prime is 11. The square of its value is 121, which is greater than 60, causing an end to the sieving loop. The unmarked integers that remain are primes.

The Sieve of Eratosthenes is not practical for identifying large prime numbers with hundreds of digits, because the algorithm has complexity $\Theta(n \ln \ln n)$, and n is exponential in the number of digits. However, a modified form of the sieve is still an important tool in number theory research.

When we implement this algorithm in the C programming language, we can use an array of $n - 1$ chars (with indices $0, 1, \ldots, n - 2$) to represent the natural numbers $2, 3, \ldots, n$. The boolean value at index i indicates whether natural number $i + 2$ is marked.

5.3 SOURCES OF PARALLELISM

How should we partition this algorithm? Because the heart of the algorithm is marking elements of the array representing integers, it makes sense to do a domain decomposition, breaking the array into $n - 1$ elements and associating a primitive task with each of these elements.

The key parallel computation is step 3a, where those elements representing multiples of a particular prime k are marked as composite. For the cell representing integer j, this computation is straightforward: if $j \bmod k = 0$, then j is a multiple of k and should be marked.

If a primitive task represents each integer, then two communications are needed to perform step 3b each iteration of the **repeat . . . until** loop. A reduction is needed each iteration in order to determine the new value of k, and then a broadcast is needed to inform all the tasks of the new value of k.

Reflecting on this domain decomposition, the good news is that there is plenty of data parallelism to exploit. The bad news is that there are a lot of reduction and broadcast operations.

The next step in our design is to think about how to agglomerate the primitive tasks into more substantial tasks that still allow us to utilize a reasonable number of processors. In the best case we will end up with a new version of the parallel algorithm that requires less computation *and* less communication than the original parallel algorithm.

5.4 DATA DECOMPOSITION OPTIONS

After we agglomerate the primitive tasks, a single task will be responsible for a group of array elements representing several integers. We often call the final grouping of data elements—the result of partitioning, agglomeration, and mapping—the **data decomposition,** or simply "the decomposition."

5.4.1 Interleaved Data Decomposition

First, let's consider an interleaved decomposition of array elements:

- process 0 is responsible for the natural numbers $2, 2 + p, 2 + 2p, \ldots,$
- process 1 is responsible for the natural numbers $3, 3 + p, 3 + 2p, \ldots,$

and so on.

An advantage of the interleaved decomposition is that given a particular array index i, it is easy to determine which process controls that index (process $i \bmod p$). A disadvantage of an interleaved decomposition *for this problem* is that it can lead to significant load imbalances among the processes. For example, if two processes are marking multiples of 2, process 0 marks $\lceil (n-1)/2 \rceil$ elements while process 1 marks none. A further disadvantage is that the implementation of step 3b (finding the next prime number) still requires some sort of reduction/broadcast.

5.4.2 Block Data Decomposition

An alternative is a **block data decomposition.** That means we divide the array into p contiguous blocks of roughly equal size. If the number of array elements n is a multiple of the number of processes p, the division is straightforward.

If n is not a multiple of p, then it is more complicated. Suppose $n = 1024$ and $p = 10$. In that case $1024/10 = 102.4$. If we give every process 102 elements, there will be four left over. On the other hand, we cannot give every process 103 elements, because the array is not that large. We cannot simply give the first $p - 1$ processes $\lceil n/p \rceil$ combinations and give the last process whatever is left over, because there may not be any elements left (see Exercise 5.2). Allocating no elements to a process is undesirable for two reasons. First, it can complicate the logic of programs in which processes exchange values. Second, it can lead to a less efficient utilization of the communication network.

What we need instead is a block allocation scheme that balances the workload by assigning to each process either $\lceil n/p \rceil$ or $\lfloor n/p \rfloor$ elements. (If n is evenly divisible by p every process will be assigned n/p elements.) Let's consider two different ways of accomplishing this.

The first method begins by computing $r = n \bmod p$. If r is 0, then n is a multiple of p, and every process should get a block of size n/p. If $r > 0$, then the first r processes should get a block of size $\lceil n/p \rceil$ and the remaining $p - r$ processes should get a block of size $\lfloor n/p \rfloor$.

For example, when $n = 1024$ and $p = 10$, the first four processes would get 103 pieces of work, and the last six processes would get 102 pieces of work.

There are two questions we typically need to be able to answer when developing algorithms based on a block allocation of data. What is the range of elements controlled by a particular process? Which process controls a particular element?

Let's answer these questions for our first scheme.

Suppose n is the number of elements and p is the number of processes. The first element controlled by process i is

$$i\lfloor n/p \rfloor + \min(i, r)$$

The last element controlled by process i is the element immediately before the first element controlled by process $i + 1$:

$$(i+1)\lfloor n/p \rfloor + \min(i+1, r) - 1$$

The process controlling a particular array element j is

$$\min(\lfloor j/(\lfloor n/p \rfloor + 1) \rfloor, \lfloor (j - r)/\lfloor n/p \rfloor \rfloor)$$

All of these expressions are somewhat complicated. The expressions for the first and last elements controlled by a particular process are not onerous, because each process could compute these values and store the results at the beginning of the algorithm. However, determining the controlling process from the element index would most likely be done on the fly, so the complexity of this expression is worrisome.

The second block allocation scheme we are considering does not concentrate all of the larger blocks among the smaller-numbered processes. Suppose n is the number of elements and p is the number of processes. The first element controlled by process i is

$$\lfloor in/p \rfloor$$

The last element controlled by process i is the element immediately before the first element controlled by process $i + 1$:

$$\lfloor (i+1)n/p \rfloor - 1$$

The process controlling a particular array element j is

$$\lfloor (p(j+1) - 1)/n \rfloor$$

Figure 5.3 contrasts these two block data decomposition methods.

The second approach is superior because it requires fewer operations to perform the three most common block management computations, especially since integer division in C automatically rounds down the result. It is the block decomposition method we will use for the remainder of the book.

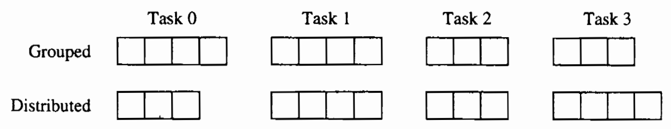

Figure 5.3 An example of two block data decomposition schemes. In this case 14 elements are divided among four tasks. In the first scheme the larger blocks are held by the lowest-numbered tasks; in the second scheme the larger blocks are distributed among the tasks.

5.4.3 Block Decomposition Macros

Let's pause for a moment and define three C macros that can be used in any of our parallel programs where a group of data items is distributed among a set of processors using a block decomposition.

```
#define BLOCK_LOW(id,p,n)  ((id)*(n)/(p))
#define BLOCK_HIGH(id,p,n)  (BLOCK_LOW((id)+1,p,n) - 1)
#define BLOCK_SIZE(id,p,n)  (BLOCK_LOW((id)+1)-BLOCK_LOW(id))
#define BLOCK_OWNER(index,p,n)  (((p)*((index)+1)-1)/(n))
```

Given process rank id, number of processes p, and number of elements n, macro BLOCK_LOW expands to an expression whose value is the first, or lowest, index controlled by the process.

Given the same arguments, macro BLOCK_HIGH expands to an expression whose value is the last, or highest, index controlled by the process.

With the same three arguments, macro BLOCK_SIZE evaluates to the number of elements controlled by process id.

Passed an array index, the number of processes, and the total number of array elements, macro BLOCK_OWNER evaluates to the rank of the process controlling that element of the array.

These four definitions are the start of a set of utility macros and functions we can reference when constructing our parallel programs.

5.4.4 Local Index versus Global Index

When we decompose an array into pieces distributed among a set of tasks, we must remember to distinguish between the local index of an array element and its global index.

For example, consider an array distributed among tasks as shown in Figure 5.4. Eleven array elements are distributed among three tasks. Each task is responsible for either three or four elements; hence the local indices range from 0 to either 2 or 3. However, each local array represents a portion of the larger, global array, whose indices range from 0 to 10.

We must keep this distinction in mind when transforming sequential programs into parallel programs. Sequential codes always use the global indices to reference array elements. We must substitute the local indices when we write our parallel codes.

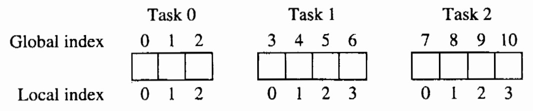

Figure 5.4 When an array is distributed among tasks, you must distinguish between an array element's local index and its global index. Here an 11-element array is distributed blockwise among three tasks.

5.4.5 Ramifications of Block Decomposition

How does our block decomposition affect the implementation of the parallel algorithm?

First, note that the largest prime used to sieve integers up to n is $\sqrt{n}$. If the first process is responsible for integers through $\sqrt{n}$, then finding the next value of k requires no communications at all—it saves a reduction step. Is this assumption reasonable? The first process has about n/p elements. If $n/p > \sqrt{n}$, then it will control all primes through $\sqrt{n}$. Since n is expected to be in the millions, this is a reasonable assumption.

A second advantage of a block decomposition is that it can speed the marking of cells representing multiples of k. Rather than check each array element to see if it represents an integer that is a multiple of k—requiring n/p modulo operations for each prime—the algorithm can find the first multiple of k and then mark that cell (call it j) as well as cells $j+k$, $j+2k$, etc., through the end of the block, for a total of about $(n/p)/k$ assignment statements. In other words, it can use a loop similar to the one used in a sequential implementation of the algorithm. This is much faster.

We have seen, then, how in this case a block decomposition results in fewer computational steps *and* fewer communications steps.

5.5 DEVELOPING THE PARALLEL ALGORITHM

Now that we have determined the data decomposition, we return to the sequential algorithm shown in Figure 5.1 and see how each step translates into equivalent steps in the parallel algorithm.

Step 1 is simple to translate. Instead of a single process creating an entire list of natural numbers, each process in the parallel program will create its portion of the list, containing either $\lceil n/p \rceil$ or $\lfloor n/p \rfloor$ boolean values.

Every process is going to need to know the value of k in order to mark the multiples of k in its region. For that reason, every process in the parallel program executes step 2. This is an example of a parallel program replicating work. Fortunately, in this case the amount of replicated work is trivial.

Step 3a is also easy to translate. Each process is responsible for marking all the multiples of k in its block between k^2 and n. We may need to do a little bit of algebra to determine the location of the first multiple of k in the block, but after that, all we need to do is mark every kth element in the block.

As we have already determined, process 0 is exclusively responsible for determining the next value of k if $p < \sqrt{n}$, which is true for all values of n for which we would reasonably want to execute the parallel algorithm. If process 0 is responsible for finding the next prime in step 3b, which determines the new value of k, then all of the other processes must receive the new value of k so that they may compute the value of the termination expression in the **repeat . . . until** loop and possibly use it in the next iteration of the loop.

In other words, we want to copy the up-to-date value of k on process 0 to the local instances of k located on the other processes. This is an example of broadcasting, a global communication function.

5.5.1 Function `MPI_Bcast`

Let's look at the header of function `MPI_Bcast`, which enables a process to broadcast one or more data items of the same type to all other processes in a communicator:

```
int MPI_Bcast (
   void         *buffer,   /* Addr of 1st broadcast element */
   int           count,    /* # elements to broadcast */
   MPI_Datatype  datatype, /* Type of elements to broadcast */
   int           root,     /* ID of process doing broadcast */
   MPI_Comm      comm)     /* Communicator */
```

The second parameter, `count`, indicates how many elements are being broadcast. Every process calling this function needs to specify the same value for `count`. The first parameter, `buffer`, is the address of the first data item to be broadcast. The function assumes all of the data items are in contiguous memory locations. The third parameter, `datatype`, is an MPI constant indicating the type of the data items to be broadcast. Parameter four, `root`, is the rank of the process broadcasting the data item(s). Finally, the fifth parameter, `comm`, indicates the communicator, the group of processes participating in this collective communication function.

In the case of our parallel sieve algorithm, process 0 needs to broadcast a single integer, k, to all other processes. Hence the call takes this form:

```
MPI_Bcast (&k, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

After this function has executed, every process has an up-to-date value of k and is able to evaluate the termination condition in the **repeat** ... **until** loop.

At the conclusion of the **repeat** ... **until** loop, all the primes between 2 and n have been discovered. They correspond to the unmarked elements of the boolean array. A more meaningful program would then make use of the primes. Since we are more interested in learning about parallel programming than number theory, let's take the easy way out and simply count the number of primes in the range 2 through n.

It is straightforward for each process to count the number of primes (number of array elements equal to 0) in its local array. At that point we need to perform a sum-reduction to accumulate these subtotals into a grand total. As we saw in the previous chapter, this is implemented using the MPI function `MPI_Reduce`.

The task/channel graph for our parallel algorithm appears in Figure 5.5.

5.6 ANALYSIS OF PARALLEL SIEVE ALGORITHM

Now that we have designed a parallel algorithm, let's derive an expression that approximates its execution time.

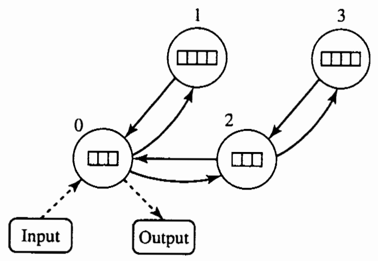

Figure 5.5 Task/channel graph for the parallel Sieve of Eratosthenes algorithm with four tasks. The dotted arrows represent channels used for I/O. The curved arrows represent channels used for the broadcast step. The straight, solid arrows represent channels used for the reduction step (as previously illustrated in Figure 3.12).

Let χ represent the time needed to mark a particular cell as being the multiple of a prime. This time includes not only the time needed to assign 1 to an element of the array, but also time needed for incrementing the loop index and testing for termination. The sequential algorithm has time complexity $\Theta(n \ln \ln n)$. We can determine χ experimentally by running a sequential version of the algorithm. In other words, the expected execution time of the serial algorithm is roughly $\chi n \ln \ln n$.

Since only a single data value is broadcast each iteration, the cost of each broadcast is closely approximated by $\lambda \lceil \log p \rceil$, where λ is message latency.

How many times will this loop iterate? The number of primes between 2 and n is about $n/\ln n$ [11]. Hence a good approximation to the number of loop iterations is $\sqrt{n}/\ln \sqrt{n}$.

Therefore, the expected execution time of the parallel algorithm is approximately

$$\chi(n \ln \ln n)/p + (\sqrt{n}/\ln \sqrt{n})\lambda \lceil \log p \rceil$$

5.7 DOCUMENTING THE PARALLEL PROGRAM

The complete text of the parallel Sieve of Eratosthenes program appears in Figure 5.6. In this section we thoroughly document the program.

We begin with the standard include files. Header file `MyMPI.h` contains macros and function prototypes for the utilities we are developing. From now on, we'll include this header file in our programs. We also define a macro that computes the minimum of two values.

```
/*
 *   Sieve of Eratosthenes
 */

#include <mpi.h>
#include <math.h>
#include <stdio.h>
#include "MyMPI.h"
#define MIN(a,b)  ((a)<(b)?(a):(b))

int main (int argc, char *argv[])
{
   int    count;         /* Local prime count */
   double elapsed_time;  /* Parallel execution time */
   int    first;         /* Index of first multiple */
   int    global_count;  /* Global prime count */
   int    high_value;    /* Highest value on this proc */
   int    i;
   int    id;            /* Process ID number */
   int    index;         /* Index of current prime */
   int    low_value;     /* Lowest value on this proc */
   char  *marked;        /* Portion of 2,...,'n' */
   int    n;             /* Sieving from 2, ..., 'n' */
   int    p;             /* Number of processes */
   int    proc0_size;    /* Size of proc 0's subarray */
   int    prime;         /* Current prime */
   int    size;          /* Elements in 'marked' */

   MPI_Init (&argc, &argv);

   /* Start the timer */

   MPI_Barrier(MPI_COMM_WORLD);
   elapsed_time = -MPI_Wtime();

   MPI_Comm_rank (MPI_COMM_WORLD, &id);
   MPI_Comm_size (MPI_COMM_WORLD, &p);

   if (argc != 2) {
      if (!id) printf ("Command line: %s <m>\n", argv[0]);
      MPI_Finalize();
      exit (1);
   }

   n = atoi(argv[1]);

   /* Figure out this process's share of the array, as
      well as the integers represented by the first and
      last array elements */

   low_value = 2 + BLOCK_LOW(id,p,n-1);
   high_value = 2 + BLOCK_HIGH(id,p,n-1);
   size = BLOCK_SIZE(id,p,n-1);

   /* Bail out if all the primes used for sieving are
      not all held by process 0 */
```

Figure 5.6 MPI program for Sieve of Eratosthenes.

```
   proc0_size = (n-1)/p;

   if ((2 + proc0_size) < (int) sqrt((double) n)) {
      if (!id) printf ("Too many processes\n");
      MPI_Finalize();
      exit (1);
   }

   /* Allocate this process's share of the array. */

   marked = (char *) malloc (size);

   if (marked == NULL) {
      printf ("Cannot allocate enough memory\n");
      MPI_Finalize();
      exit (1);
   }

   for (i = 0; i < size; i++) marked[i] = 0;
   if (!id) index = 0;
   prime = 2;
   do {
      if (prime * prime > low_value)
         first = prime * prime - low_value;
      else {
         if (!(low_value % prime)) first = 0;
         else first = prime - (low_value % prime);
      }
      for (i = first; i < size; i += prime) marked[i] = 1;
      if (!id) {
         while (marked[++index]);
         prime = index + 2;
      }
      MPI_Bcast (&prime,  1, MPI_INT, 0, MPI_COMM_WORLD);
   } while (prime * prime <= n);
   count = 0;
   for (i = 0; i < size; i++)
      if (!marked[i]) count++;
   MPI_Reduce (&count, &global_count, 1, MPI_INT, MPI_SUM,
      0, MPI_COMM_WORLD);

   /* Stop the timer */

   elapsed_time += MPI_Wtime();


   /* Print the results */

   if (!id) {
      printf ("%d primes are less than or equal to %d\n",
         global_count, n);
      printf ("Total elapsed time: %10.6f\n", elapsed_time);
   }
   MPI_Finalize ();
   return 0;
}
```

Figure 5.6 (contd.) MPI program for Sieve of Eratosthenes.

```
#define MIN(a,b)   ((a)<(b)?(a):(b))
```

The user is supposed to specify the upper range of the sieve as a command-line argument. If this value is missing, we terminate execution. In this case it is vital that each process calls `MPI_Finalize()` before it exits. If the command-line argument exists, we convert the string into an integer.

```
if (argc != 2) {
   if (!id) printf ("Command line: %s <m>\n", argv[0]);
   MPI_Finalize();
   exit (1);
}

n = atoi(argv[1]);
```

The program will find all primes from 2 through n, meaning we are checking the primality of a total of $n - 1$ integers. As we discussed earlier, we will give each process a contiguous block of the array that stores the marks. We determine the low and high values for which this process is responsible, as well as the total number of values it is sieving, using the macros we have developed.

```
low_value = 2 + BLOCK_LOW(id,p,n-1);
high_value = 2 + BLOCK_HIGH(id,p,n-1);
size = BLOCK_SIZE(id,p,n-1);
```

Our algorithm works only if the square of the largest value in process 0's array is greater than the upper limit of the sieve. We add code that checks to ensure that this condition is true. If not, the program terminates.

```
proc0_size = (n-1)/p;

if ((2 + proc0_size) < (int) sqrt((double) n)) {
   if (!id) printf ("Too many processes\n");
   MPI_Finalize();
   exit (1);
}
```

Now we can allocate the process's share of the array. Because a single byte is the smallest unit of memory that can be indexed in C, we declare the array to be of type `char`. If the memory allocation fails, the program terminates.

```
marked = (char *) malloc (size);

if (marked == NULL) {
   printf ("Cannot allocate enough memory\n");
   MPI_Finalize();
   exit (1);
}
```

The elements of the list are unmarked.

```
for (i = 0; i < size; i++) marked[i] = 0;
```

Whew! We have completed Step 1 of the algorithm. Fortunately, the remaining steps can be implemented with much less coding. Step 2, for example, requires only two lines. We will begin by sieving multiples of 2. Integer `prime` is the value of the current prime being sieved. Integer `index` is its index in the array of process 0. We conditionalize the initialization of `index` to process 0 to emphasize that only process 0 uses this variable.

```
if (!id) index = 0;
prime = 2;
```

Now we are at the heart of the program, corresponding to Step 3 in the original algorithm. We implement **repeat . . . until** in C as a `do...while` loop.

Each process is responsible for marking in its portion of the list all multiples of `prime` between `prime` squared and `n`. To do this, we need to determine the index corresponding to the first integer needing marking. If `prime` squared is greater than the smallest value stored in the array, then we take the difference between the two values to determine the index of the first element that needs to be marked. Otherwise, we find the remainder when we divide `low_value` by `prime`. If the remainder is 0, `low_value` is a multiple of `prime`, and that is where we should begin marking. Otherwise, we must index into the array to be at the first element that is a multiple of `prime`.

```
if (prime * prime > low_value)
   first = prime * prime - low_value;
else {
   if (!(low_value % prime)) first = 0;
   else first = prime - (low_value % prime);
}
```

The following `for` loop actually does the sieving. Each process marks the multiples of the current prime number from the first index through the end of the array.

```
for (i = first; i < size; i += prime) marked[i] = 1;
```

Process 0 finds the next prime by locating the next unmarked location in the array.

```
if (!id) {
   while (marked[++index]);
   prime = index + 2;
}
```

Process 0 broadcasts the value of the next prime to the other processes.

```
MPI_Bcast (&prime,  1, MPI_INT, 0, MPI_COMM_WORLD);
```

The processes continue to sieve as long as the square of the current prime is less than or equal to the upper limit.

```
} while (prime * prime <= n);
```

Each process counts the number of primes in its portion of the list.

```
count = 0;
for (i = 0; i < size; i++)
   if (!marked[i]) count++;
```

The processes compute the grand total, with the result being stored in variable `global_count` on process 0.

```
MPI_Reduce (&count, &global_count, 1, MPI_INT, MPI_SUM,
   0, MPI_COMM_WORLD);
```

We stop the timer. At this point `elapsed_time` contains the number of seconds it took to execute the algorithm, excluding initial MPI startup time.

```
elapsed_time += MPI_Wtime();
```

Process 0 prints the answer and the elapsed time.

```
if (!id) {
   printf ("%d primes are less than or equal to %d\n",
      global_count, n);
   printf ("Total elapsed time: %10.6f\n", elapsed_time);
}
```

All that remains is a call to `MPI_Finalize` to shut down MPI.

5.8 BENCHMARKING

Let's see how well our model compares with the actual performance of the parallel program finding all primes up to 100 million.

We will execute our parallel program on a commodity cluster of 450 MHz Pentium II CPUs. Each CPU has a fast Ethernet connection to a Hewlett-Packard Procurve 4108GL switch.

First, we determine the value of χ by running a sequential implementation of the program on a single processor of the cluster. The sequential program executes in 24.900 seconds. Hence

$$\chi = \frac{24.900 \text{ sec}}{100{,}000{,}000 \ln \ln 100{,}000{,}000} = 85.47 \text{ nanoseconds}$$

We also need to determine λ. By performing a series of broadcasts on $2, \ldots, 8$ processors, we determine $\lambda = 250\ \mu$sec.

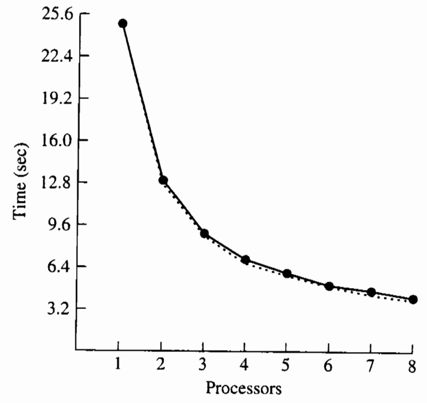

Figure 5.7 Comparison of the predicted (dotted line) and actual (solid line) execution times of the parallel Sieve of Eratosthenes program.

Plugging these values into our formula for the expected execution time of the parallel algorithm, we find

$$\chi(n \ln\ln n)/p + (\sqrt{n}/\ln\sqrt{n})\lambda \log p = 24.900/p + 0.2714\lceil \log p \rceil \text{ sec}$$

We benchmark our parallel program by executing it 40 times—five times for each number of processors between 1 and 8. For each number of processors we compute the mean execution time. Figure 5.7 compares our experimental results with the execution times predicted by our model. The average error of the predictions for $2, \ldots, 8$ processors is about 4 percent.

5.9 IMPROVEMENTS

While the parallel sieve algorithm we have developed does exhibit good performance, there are a few modifications to the program that can improve performance significantly. In this section we present three modifications to the parallel sieve algorithm. Each change builds on the previous ones.

5.9.1 Delete Even Integers

Since 2 is the only even prime, there is little sense in setting aside half of the boolean values in the array for even integers. Changing the sieve algorithm so that only odd integers are represented halves the amount of storage required and doubles the speed at which multiples of a particular prime are marked. With this change the estimated execution time of the sequential algorithm becomes

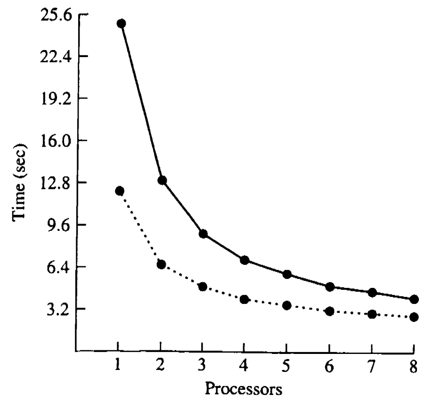

Figure 5.8 Execution time of the original (solid line) and improved (dotted line) parallel programs performing the Sieve of Eratosthenes.

approximately

$$\chi(n \ln \ln n)/2$$

and the estimated execution time of the parallel algorithm becomes approximately

$$\chi(n \ln \ln n)/(2p) + (\sqrt{n}/\ln \sqrt{n})\lambda \log p$$

Figure 5.8 plots the results of benchmarking the original parallel sieve algorithm and the improved algorithm sieving 100 million integers on $1, 2, \ldots, 8$ processors. As expected, the time required for the improved algorithm is about half the time required for the original algorithm, at least when the number of processors is small.

In fact, while our improved sieve runs twice as fast as our original program on one processor, it executes only slightly faster on eight processors. The computation time of the improved program is significantly lower than that of the original program, but the communication requirements are identical. As the number of processors increases, the relative importance of the communication component to overall execution time grows, shrinking the difference between the two programs.

5.9.2 Eliminate Broadcast

Consider step 3b of the original algorithm, in which the new sieve value k is identified. We made this step parallel by letting one process identify the new value of k and then broadcast it to the other processes. During the course of the program's execution this broadcast step is repeated about $\sqrt{n}/\ln \sqrt{n}$ times.

Why not let every task identify the new value of k? In our original data decomposition scheme this is impossible, because only task 0 controls the array elements associated with the integers $2, 3, \ldots, \sqrt{n}$. What if we chose to replicate these values?

Suppose in addition to each task's set of about n/p integers, each task also has a separate array containing integers $3, 5, 7, \ldots, \lfloor\sqrt{n}\rfloor$. Before finding the primes from 3 through n, each task will use the sequential algorithm to find the primes from 3 through $\lfloor\sqrt{n}\rfloor$. Once this has been done, each task now has its own private copy of an array containing all the primes between 3 and $\lfloor\sqrt{n}\rfloor$. Now the tasks can sieve their portions of the larger array without any broadcast steps.

Eliminating the broadcast step improves the speed of the parallel algorithm if

$$\begin{aligned} & (\sqrt{n}/\ln \sqrt{n})\lambda\lceil\log p\rceil > \chi\sqrt{n}\ln\ln\sqrt{n} \\ \Rightarrow\ & (\lambda\lceil\log p\rceil)/\ln\sqrt{n} > \chi \ln\ln\sqrt{n} \\ \Rightarrow\ & \lambda > \chi \ln\ln\sqrt{n}\ln\sqrt{n}/\lceil\log p\rceil \end{aligned}$$

The expected time complexity of the parallel algorithm is now approximately

$$\chi\left((n \ln\ln n)/(2p) + \sqrt{n}\ln\ln\sqrt{n}\right) + \lambda\lceil\log p\rceil$$

(The final term represents the time needed to do the sum-reduction.)

5.9.3 Reorganize Loops

For much of the execution of the parallel sieve algorithm, each process is marking widely dispersed elements of a very large array, leading to a poor cache hit rate. Think of the heart of the algorithm developed in the previous subsection as two large loops. The outer loop iterates over prime sieve values between 3 and $\lfloor\sqrt{n}\rfloor$, while the inner loop iterates over the process's share of the integers between 3 and n. If we exchange the inner and outer loops, we can improve the cache hit rate. We can fill the cache with a section of the larger subarray, then strike all the multiples of all the primes less than $\lfloor\sqrt{n}\rfloor$ on that section before bringing in the next section of the subarray. (See Figure 5.9.)

5.9.4 Benchmarking

Figure 5.10 plots the execution times of the original parallel Sieve of Eratosthenes program and all three improved versions, when finding primes less than 100 million on $1, 2, \ldots, 8$ processors. The underlying hardware is a commodity cluster consisting of 450 MHz Pentium II CPUs connected by fast Ethernet to a Hewlett-Packard Procurve 4108GL switch.

The execution time of the original sequential program is the same as Sieve 1 on one processor. On eight processors, our parallel sieve program that incorporates every described optimization executes about 72.8 times faster than the original sequential program. The larger share of the increase (a factor of 9.8) results from

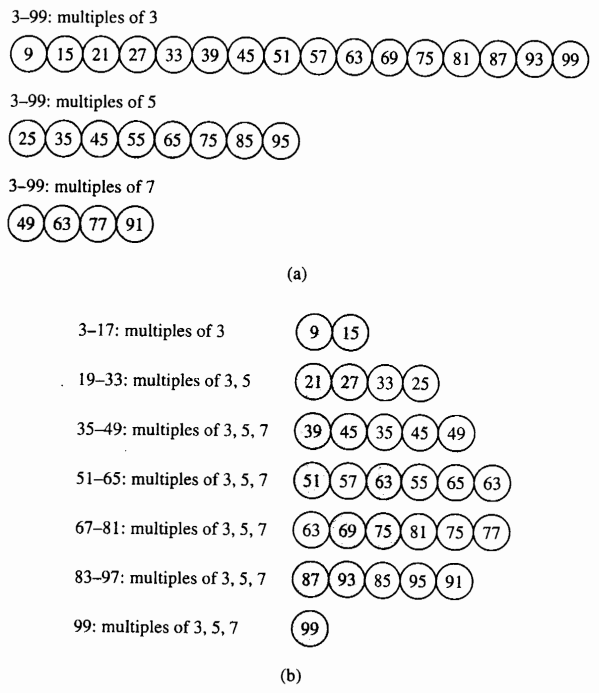

Figure 5.9 Changing the order in which composite integers are marked can dramatically improve the cache hit rate. In this example we are finding primes between 3 and 99. Suppose the cache has four lines, and each line can hold four bytes. One line contains bytes representing integers 3, 5, 7, and 9; the next line holds bytes representing 11, 13, 15, and 17; etc. (a) Sieving all multiples of one prime before considering next prime. Shaded circles represent cache misses. By the time the algorithm returns to the bytes representing smaller integers, they are no longer in the cache. (b) Sieving multiples of all primes for 8 bytes in two cache lines before considering the next group of 8 bytes. Fewer shaded circles indicates the cache hit rate has improved.

eliminating the storage and manipulation of even integers and inverting the two principal loops to improve the cache hit rate. The smaller share of the increase (a factor of about 7.4) results from redundantly computing primes up to $\sqrt{n}$ to eliminate broadcasts and using eight processors instead of one. Our greater gains, then, were the result of improvements to the sequential algorithm before parallelism was applied.

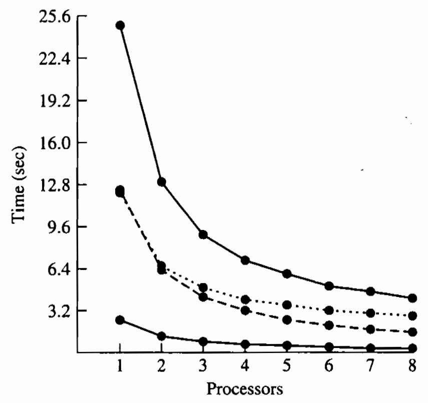

Figure 5.10 Execution time of four parallel implementations of the Sieve of Eratosthenes on a commodity cluster of 450 MHz Pentium II processors connected by fast Ethernet. The upper solid line is the original program. The dotted line is the execution time of the program that does not store or strike even integers. The dashed line plots the execution time of the program that eliminates broadcasts. The lower solid line shows the execution time of the program that incorporates the additional optimization of interchanging loops in order to improve the cache hit rate.

5.10 SUMMARY

We began with a sequential algorithm for the Sieve of Eratosthenes and used the domain decomposition methodology to identify parallelism. For this algorithm, a blockwise distribution of array values to processes is superior to an interleaved distribution. The data-parallel algorithm we designed requires that task 0 broadcasts the current prime to the other tasks. The resulting parallel program uses the function `MPI_Bcast` to perform this broadcast operation. The program achieves good performance on a commodity cluster finding primes up to 100 million.

We then examined three improvements to the original parallel version. The first improvement eliminated all manipulation of even integers, roughly cutting in half both storage requirements and overall execution time. The second improvement eliminates the need for a broadcast step by making redundant the portion of the computation that determines the next prime. The cost of this improvement is a requirement that each process store all odd integers between 3 and $\sqrt{n}$.

The third enhancement improved the cache hit rate by striking all composite values for a single cache-full of integers before moving on to the next segment.

Note that our fourth program executes faster on one processor than our original program does on eight processors. Comparing both programs on eight processors, the fourth program executes more than 11 times faster than our original program. It is important to maximize single processor performance even when multiple processors are available.

5.11 KEY TERMS

block decomposition	data decomposition

5.12 BIBLIOGRAPHIC NOTES

Luo [76] presents a version of the Sieve of Eratosthenes in which neither multiples of 2 nor multiples of 3 appear in the array of integers to be marked.

5.13 EXERCISES

5.1 Consider a simple block allocation of n data items to p processes in which the first $p-1$ processes get $\lceil n/p \rceil$ items each and the last process gets what is left over.

a. Find values for n and p where the last process does not get any elements.

b. Find values for n and p where $\lfloor p/2 \rfloor$ processes do not get any values. Assume $p > 1$.

5.2 This chapter presents two block data decomposition strategies that assign n elements to p processes such that each process is assigned either $\lfloor n/p \rfloor$ or $\lceil n/p \rceil$ elements. For each pair of values n and p, use a table or an illustration to show how these two schemes would assign array elements to processes:

a. $n = 15$ and $p = 4$

b. $n = 15$ and $p = 6$

c. $n = 16$ and $p = 5$

d. $n = 18$ and $p = 4$

e. $n = 20$ and $p = 6$

f. $n = 23$ and $p = 7$

5.3 Use the analytical model developed in Section 5.6 to predict the execution time of the original parallel sieve program on $1, 2, \ldots, 16$ processors. Assume $n = 10^8$, $\lambda = 250\ \mu$sec, and $\chi = 0.0855\ \mu$sec.

5.4 Use the analytical model developed in Section 5.9.1 to predict the execution time of the second version of the parallel sieve program (the one that does not store or mark even integers). Compare the execution

time predicted by the model to the actual execution time reported in column 2 of Table 5.1. What is the average error of the predictions for $2, \ldots, 8$ processors?

Table 5.1 Mean execution times (in seconds) of four parallel implementations of the Sieve of Eratosthenes on a commodity cluster of 450 MHz Pentium II processors connected by fast Ethernet. Sieve 1 is the original program. Sieve 2 does not store or strike even integers. Sieve 3 incorporates the additional optimization of eliminating broadcasts by computing primes between 2 and $\sqrt{n}$ on each processor. Sieve 4 incorporates the additional optimization of interchanging loops in order to improve the cache hit rate.

Processors	Sieve 1	Sieve 2	Sieve 3	Sieve 4
1	24.900	12.237	12.466	2.543
2	12.721	6.609	6.378	1.330
3	8.843	5.019	4.272	0.901
4	6.768	4.072	3.201	0.679
5	5.794	3.652	2.559	0.543
6	4.964	3.270	2.127	0.456
7	4.371	3.059	1.820	0.391
8	3.927	2.856	1.585	0.342

5.5 Use the analytical model developed in Section 5.9.2 as a starting point to predict the execution time of the third version of the parallel sieve program. Assume $n = 10^8$, $\lambda = 250\ \mu$sec, and $\chi = 0.0855\ \mu$sec. Compare the execution time predicted by your model to the actual execution time reported in column *Sieve 3* of Table 5.1. What is the average error of the predictions for $2, \ldots, 8$ processors?

5.6 Modify the parallel Sieve of Eratosthenes program presented in the text to incorporate the first improvement described in Section 5.9: it should not set aside memory for even integers. Benchmark your program, comparing its performance with that of the original parallel sieve program.

5.7 Modify the parallel Sieve of Eratosthenes program presented in the book to incorporate the first two improvements described in Section 5.9. Your program should not set aside memory for even integers, and each process should use the sequential Sieve of Eratosthenes algorithm on a separate array to find all primes between 3 and $\lfloor \sqrt{n} \rfloor$. With this information, the call to `MPI_Bcast` can be eliminated. Benchmark your program, comparing its performance with that of the original parallel sieve program.

5.8 Modify the parallel Sieve of Eratosthenes program presented in the text to incorporate all three improvements described in Section 5.9. Benchmark your program, comparing its performance with that of the original parallel sieve program.

5.9 All the parallel sieve algorithms developed in this chapter are the result of a domain decomposition of the original algorithm. Write a parallel