CHAPTER

# Floyd's Algorithm

*Not once or twice in our rough island story*
*The path of duty was the path of glory.*
**Alfred, Lord Tennyson, *Ode on the Death of the Duke of Wellington***

## 6.1 INTRODUCTION

Travel maps often contain tables showing the driving distances between pairs of cities. At the intersection of the row representing city $A$ and the column representing city $B$ is a cell containing the length of the shortest path of roads from $A$ to $B$. In the case of longer trips, this route most likely passes through other cities represented in the table. Floyd's algorithm is a classic method for generating this kind of table.

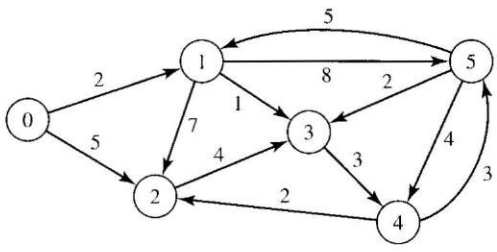
In this chapter we will design, analyze, program, and benchmark a parallel version of Floyd's algorithm. We will begin to develop a suite of functions that can read matrices from files and distribute them among MPI processes, as well as gather matrix elements from MPI processes and print them.

This chapter discusses the following MPI functions:

- `MPI_Send`, which allows a process to send a message to another process
- `MPI_Recv`, which allows a process to receive a message sent by another process

## 6.2 THE ALL-PAIRS SHORTEST-PATH PROBLEM

A **graph** is a set consisting of $V$, a finite set of vertices, and $E$, a finite set of edges between pairs of vertices. Figure 6.1a is a pictorial representation of a graph,




(a)

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0 | 2 | 5 | ∞ | ∞ | ∞ |
| 1 | ∞ | 0 | 7 | 1 | ∞ | 8 |
| 2 | ∞ | ∞ | 0 | 4 | ∞ | ∞ |
| 3 | ∞ | ∞ | ∞ | 0 | 3 | ∞ |
| 4 | ∞ | ∞ | 2 | ∞ | 0 | 3 |
| 5 | ∞ | 5 | ∞ | 2 | 4 | 0 |

(b)

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0 | 2 | 5 | 3 | 6 | 9 |
| 1 | ∞ | 0 | 6 | 1 | 4 | 7 |
| 2 | ∞ | 15 | 0 | 4 | 7 | 10 |
| 3 | ∞ | 11 | 5 | 0 | 3 | 6 |
| 4 | ∞ | 8 | 2 | 5 | 0 | 3 |
| 5 | ∞ | 5 | 6 | 2 | 4 | 0 |

(c)

**Figure 6.1** (a) A weighted, directed graph. (b) Representation of the graph as an adjacency matrix. Element $(i, j)$ represents the length of the edge from $i$ to $j$. Nonexistent edges are considered to have infinite length. (c) Solution to the all-pairs shortest path problem. Element $(i, j)$ represents the length of the shortest path from vertex $i$ to vertex $j$. The infinity symbol represents nonexistent paths.

in which vertices appear as labeled circles and edges appear as lines between pairs of circles. To be more precise, Figure 6.1a is a picture of a weighted, directed graph. It is a **weighted graph** because a numerical value is associated with each edge. Weights on edges can have a variety of meanings. In the case of shortest path problems, edge weights correspond to distances. It is a **directed graph** because every edge has an orientation (represented by an arrowhead).

Given a weighted, directed graph, the **all-pairs shortest-path problem** is to find the length of the shortest path between every pair of vertices. The length of a path is strictly determined by the weights of its edges, not the number of edges traversed. For example, the length of the shortest path between vertex 0 and vertex 5 in Figure 6.1a is 9; it traverses four edges ($0 \rightarrow 1$, $1 \rightarrow 3$, $3 \rightarrow 4$, and $4 \rightarrow 5$).

If we are going to solve this problem on a computer, we must find a convenient way to represent a weighted, directed graph. The **adjacency matrix** is the data structure of choice for this application, because it allows constant-time access to every edge and does not consume more memory than is required for storing the solution. An adjacency matrix is an $n \times n$ matrix representing a graph with $n$ vertices. In the case of a weighted graph, the value of matrix element $(i, j)$ is the weight of the edge from vertex $i$ to vertex $j$. Depending upon the application, the way that nonexistent edges are represented varies. In the case of the single-source shortest-path problem, nonexistent edges are assigned extremely high values (such as the maximum integer representable by the underlying architecture). For convenience, we will use the symbol $\infty$ to represent this extremely high value. Figure 6.1b is an adjacency matrix representation of the same graph shown pictorially in Figure 6.1a.

**Floyd's Algorithm:**

Input: $n$ — number of vertices
$a[0..n-1, 0..n-1]$ — adjacency matrix
Output: Transformed $a$ that contains the shortest path lengths

```
for k ← 0 to n − 1
  for i ← 0 to n − 1
    for j ← 0 to n − 1
      a[i, j] ← min(a[i, j], a[i, k] + a[k, j])
    endfor
  endfor
endfor
```

**Figure 6.2** Floyd's algorithm is an $\Theta(n^3)$ time algorithm that solves the all-pairs shortest-path problem. It transforms an adjacency matrix into a matrix containing the length of the shortest path between every pair of vertices.

When the algorithm terminates, the matrix contains the lengths of the shortest path between every pair of vertices. Figure 6.1c is the solution of the all-pairs shortest-path problem for the graph represented in Figure 6.1a.

More than 40 years ago Floyd invented an $\Theta(n^3)$ time algorithm for solving the all-pairs shortest-path problem. Floyd's algorithm appears in Figure 6.2. For more information on this algorithm, see Cormen et al. [18].

## 6.3 CREATING ARRAYS AT RUN TIME

A program manipulating an array is more useful if the size of the array can be specified at run-time, because it does not have to be recompiled when the size of the array to be manipulated changes. Allocating a one-dimensional array in C is easily done by declaring a scalar pointer and allocating memory from the heap with a malloc statement. For example, here is a way to allocate matrix $A$, a one-dimensional, $n$-element array of integers:

```
int *A;
...
A = (int *) malloc (n * sizeof(int));
```

Allocating a two-dimensional array is more complicated, however, since C treats a two-dimensional array as an array of arrays. We want to ensure that the array elements occupy contiguous memory locations, so that we can send or receive the entire contents of the array in a single message.

Here is one way to allocate a two-dimensional array (see Figure 6.3). First, we allocate the memory where the array values are to be stored. Second, we allocate the array of pointers. Third, we initialize the pointers.

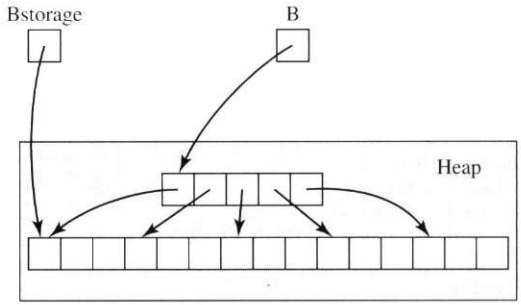



**Figure 6.3** Allocating a 5 × 3 matrix is a three-step process. First, the memory for the 15 matrix values is allocated from the heap. Variable `Bstorage` points to the start of this block of memory. Second, the memory for the five row pointers is allocated from the heap. Variable `B` points to the start of this block of memory. Third, the values of the pointers `B[0]`, `B[1]`, ..., `B[4]` are initialized.

For example, the following C code allocates $B$, a two-dimensional array of integers. The array has `m` rows and `n` columns:

```
int **B, *Bstorage, i;
...
Bstorage = (int *) malloc (m * n * sizeof(int));
B = (int **) malloc (m * sizeof(int *));
for (i = 0; i < m; i++)
   B[i] = &Bstorage[i*n];
```

The elements of $B$ may be initialized in various ways. If they are initialized through a series of assignment statements referencing `B[0][0]`, `B[0][1]`, etc., there is little room for error. However, if the elements of `B` are initialized en masse, for example, through a function call that reads the matrix elements from a file, remember to use `Bstorage`, rather than `B`, as the starting address.

## 6.4 DESIGNING THE PARALLEL ALGORITHM

### 6.4.1 Partitioning

Our first step is to determine whether to choose a domain decomposition or a functional decomposition. In this case, the choice is obvious. Looking at the pseudocode in Figure 6.2, we see that the algorithm executes the same assignment statement $n^3$ times. Unless we subdivide this statement, there is no functional parallelism. In contrast, it's easy to perform a domain decomposition. We can

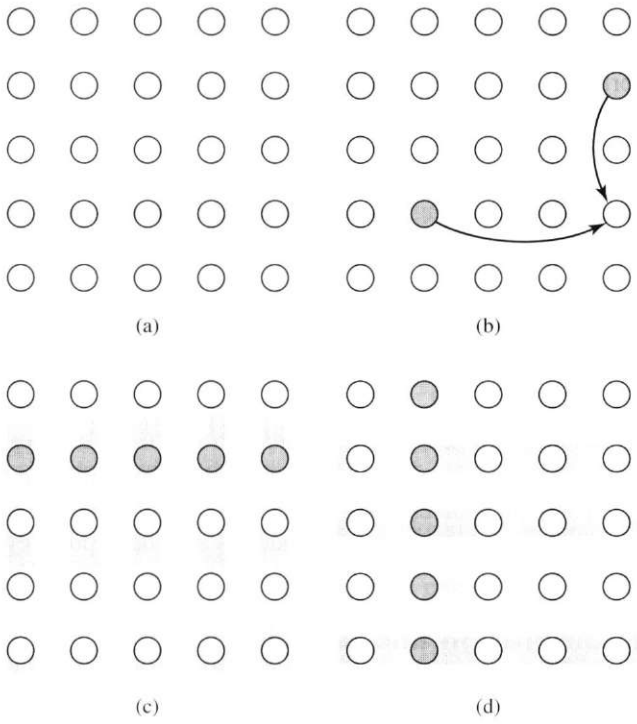



**Figure 6.4** Partitioning and communication in Floyd's algorithm. (a) A primitive task is associated with each element of the distance matrix. (b) Updating $a[3, 4]$ when $k = 1$. The new value of $a[3, 4]$ depends upon its previous value and the values of $a[3, 1]$ and $a[1, 4]$. (c) During iteration $k$ every task in row $k$ must broadcast its value to the other tasks in the same column. In this drawing $k = 1$. (d) During iteration $k$ every task in column $k$ must broadcast its value to the other tasks in the same row. In this drawing $k = 1$.

divide matrix $A$ into its $n^2$ elements and associate a primitive task with each element (Figure 6.4a).

### 6.4.2 Communication

Each update of element $a[i, j]$ requires access to elements $a[i, k]$ and $a[k, j]$. For example, Figure 6.4b illustrates the elements needed to update $a[3, 4]$ when $k = 1$. Notice that for any particular value of $k$, element $a[k, m]$ is needed by every task associated with elements in column $m$. Similarly, for any particular value of $k$, element $a[m, k]$ is needed by every task associated with elements in row $m$. What this means is that during iteration $k$ each element in row $k$ of $a$ gets

broadcast to the tasks in the same column (Figure 6.4c). Likewise, each element in column $k$ of $a$ gets broadcast to the tasks in the same row (Figure 6.4d).

It's important to question whether every element of $a$ can be updated simultaneously. After all, if updating $a[i, j]$ requires the values of $a[k, k]$ and $a[k, j]$, shouldn't we have to compute those values first?

The answer to this question is no. The reason is that the values of $a[i, k]$ and $a[k, j]$ don't change during iteration $k$. That's because during iteration $k$ the update to $a[i, k]$ takes this form:

$$a[i, k] \leftarrow \min(a[i, k], a[i, k] + a[k, k])$$

Since all values are positive, $a[i, k]$ can't decrease. Similarly, the update to $a[k, j]$ takes this form:

$$a[k, j] \leftarrow \min(a[k, j], a[k, k] + a[k, j])$$

The value of $a[k, j]$ can't decrease. Hence there is no dependence between the update of $a[i, j]$ and the updates of $a[i, k]$ and $a[k, j]$. In short, for each iteration $k$ of the outer loop, we can perform the broadcasts and then update every element of $a$ in parallel.

### 6.4.3 Agglomeration and Mapping

We'll use the decision tree of Figure 3.7 to determine our agglomeration and mapping strategy. The number of tasks is static, the communication pattern among tasks is structured, and the computation time per task is constant. Hence we should agglomerate tasks to minimize communication, creating one task per MPI process.

Our goal, then, is to agglomerate $n^2$ primitive tasks into $p$ tasks. How should we collect them? Two natural agglomerations group tasks in the same row or column (Figure 6.5). Let's examine the consequences of both of these agglomerations.

If we agglomerate tasks in the same row, the broadcast that occurs among primitive tasks in the same row (Figure 6.4d) is eliminated, because all of these data values are local to the same task. With this agglomeration, during every iteration of the outer loop one task will broadcast $n$ elements to all the other tasks. Each broadcast requires time $\lceil \log p \rceil (\lambda + n/\beta)$.

If we agglomerate tasks in the same column, then the broadcast that occurs among primitive tasks in the same column (Figure 6.4c) is eliminated. This agglomeration, too, results in a message passing time of $\lceil \log p \rceil (\lambda + n/\beta)$ per iteration.

(The truth is that we haven't considered an even better agglomeration, which groups primitive tasks associated with $(n/\sqrt{p}) \times (n/\sqrt{p})$ blocks of elements of $A$. We'll develop a matrix-vector multiplication program based on this data decomposition in Chapter 8, when we have a lot more MPI functions under our belt.)

To decide between the rowwise and columnwise agglomerations, we need to look outside the computational kernel of the algorithm. The parallel program must input the distance matrix from a file. Assume that the file contains the matrix

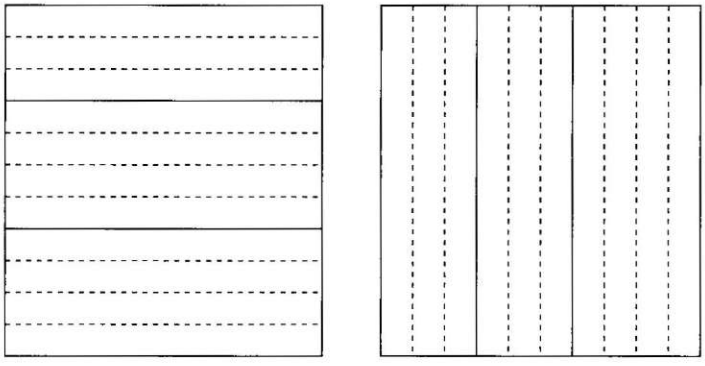



**Figure 6.5** Two data decompositions for matrices. (a) In a rowwise block-striped decomposition, each process is responsible for a contiguous group of rows. Here 11 rows are divided among three processes. (b) In a columnwise block-striped decomposition, each process is responsible for a contiguous group of columns. Here 10 columns are divided among three processes.

in row-major order. (The file begins with the first row, then the second row, etc.) In C, matrices are also stored in primary memory in row-major order. Hence distributing rows among processes is much easier if we choose a rowwise block-striped decomposition. This distribution also makes it much simpler to output the result matrix in row-major order. For this reason we choose the rowwise block-striped decomposition.

### 6.4.4 Matrix Input/Output

We must now decide how we are going to support matrix input/output.

First, let's focus on reading the distance matrix from a file. We could have each process open the file, seek to the proper location in the file, and read its portion of the adjacency matrix. However, we will let one process be responsible for file input. Before the computational loop, this process will read the matrix and distribute it to the other processes. Suppose we have $p$ processes. If process $p-1$ is responsible for reading and distributing the matrix elements, it is easy to implement the program so that no extra space is allocated for file input buffering.

Here is the reason why. If process $i$ is responsible for rows $\lfloor in/p \rfloor$ through $\lfloor (i+1)n/p \rfloor - 1$, then process $p-1$ is responsible for $\lceil n/p \rceil$ rows (see Exercise 6.1). That means no process is responsible for more rows than process $p-1$. Process $p-1$ can use the memory that will eventually store its $\lceil n/p \rceil$ rows to buffer the rows it inputs for the other processes.

Figure 6.6 shows how this method works. The last process opens the file, reads the rows destined for process 0, and sends these rows to process 0. It repeats these steps for the other processes. Finally, it reads the rows it is responsible for.

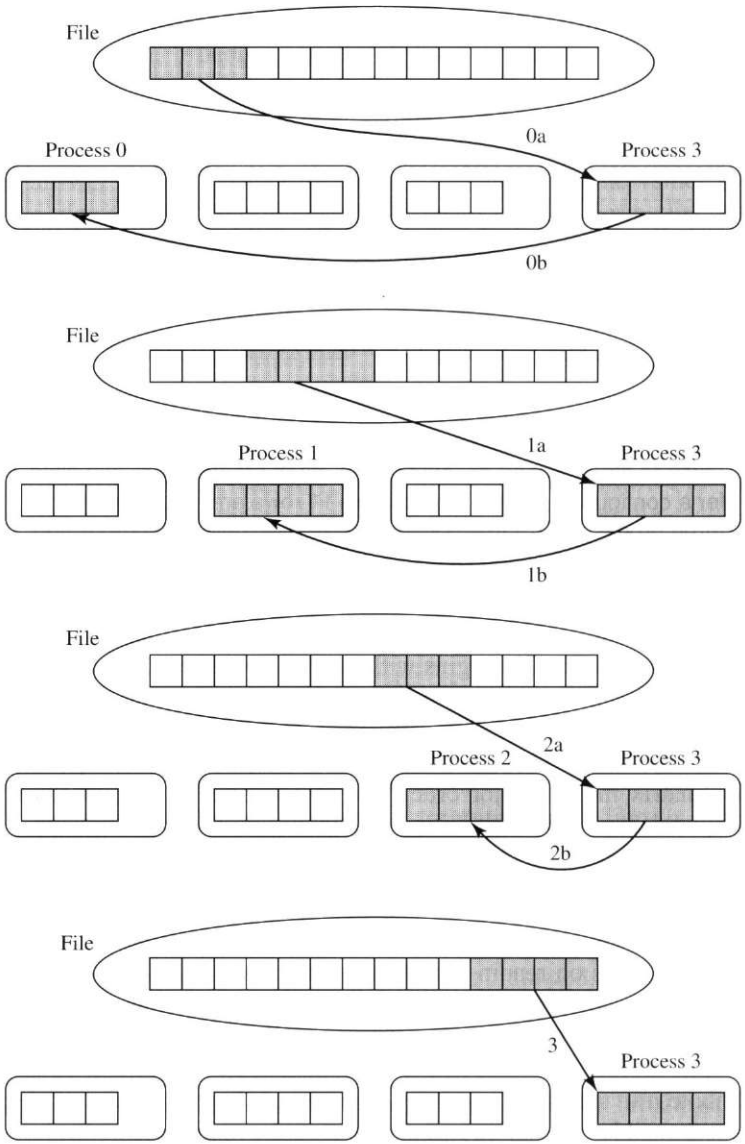



**Figure 6.6** Example of a single process managing file input. Here there are four processes, labeled 0, 1, 2, and 3. Process 3 opens the file for reading. In step 0a it reads process 0's share of the data; in step 0b it passes the data to process 0. In steps 1 and 2 it does the same for processes 1 and 2, respectively. In step 3 it inputs its own data.

The complete function, called `read_row_striped_matrix`, appears in Appendix B. Given the name of the input file, the data type of the matrix elements, and a communicator, it returns (1) a pointer to an array of pointers, allowing the matrix elements to be accessed via double-subscripting, (2) a pointer to the location containing the actual matrix elements, and (3) the dimensions of the matrix.

Our implementation of Floyd's algorithm will print the distance matrix twice: when it contains the original set of distances and after it has been transformed into the shortest-path matrix.

Process 0 does all the printing to standard output, so we can be sure the values appear in the correct order. First it prints its own submatrix, then it calls upon each of the other processes in turn to send their submatrices. Process 0 will receive each submatrix and print it.

Little is required of processes $1, 2, \ldots, p-1$. Each of these processes simply waits for a message from process 0, then sends process 0 its portion of the matrix.

Using this protocol, we ensure that process 0 never receives more than one submatrix at a time. Why don't we just let every process fire its submatrix to process 0? After all, process 0 can distinguish between them by specifying the rank of the sending process in its call to `MPI_Recv`. The reason we don't let processes send data to process 0 until requested is we don't want to overwhelm the processor on which process 0 is executing. There is only a finite amount of bandwidth into any processor. If process 0 needs data from process 1 in order to proceed, we don't want the message from process 1 to be delayed because messages are also being received from many other processes.

The source code for function `print_row_striped_matrix` appears in Appendix B.

## 6.5 POINT-TO-POINT COMMUNICATION

In our function that reads the matrix from a file, process $p-1$ reads a contiguous group of matrix rows, then sends a message containing these rows directly to the process responsible for managing them. In our function that prints the matrix, each process (other than process 0) sends process 0 a message containing its group of matrix rows. Process 0 receives each of these messages and prints the rows to standard output. These are examples of point-to-point communications.

A **point-to-point communication** involves a pair of processes. In contrast, the collective communication operations we have previously explored involve every process in a group.

Figure 6.7 illustrates a point-to-point communication. In this example, process $h$ is not involved in a communication. It continues executing statements manipulating its local variables. Process $i$ performs local computations, then sends a message to process $j$. After the message is sent, it continues on with its computation. Process $j$ performs local computations, then blocks until it receives a message from process $i$.

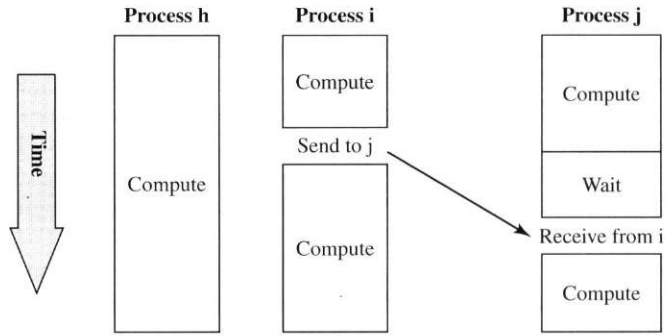



**Figure 6.7** Point-to-point communications involve pairs of processes.

```
...
if (id == i) {
   ...
   /* Send message to j */
   ...
} else if (id == j) {
   ...
   /* Receive message from i */
   ...
}
...
```

**Figure 6.8** MPI functions performing point-to-point communications often occur inside conditionally executed code.

If every MPI process executes the same program, how can one process send a message while a second process receives a message and a third process does neither?

In order for execution of MPI function calls to be limited to a subset of the processes, these calls must be inside conditionally executed code. Figure 6.8 demonstrates one way that process $i$ could send a message to process $j$, while the remaining processes skip the message-passing function calls.

Now let's look at the headers of two MPI functions that we can use to perform a point-to-point communication.

### 6.5.1 Function `MPI_Send`

The sending process calls function `MPI_Send`:

```
int MPI_Send (
      void          *message,
      int            count,
```

```
      MPI_Datatype  datatype,
      int           dest,
      int           tag,
      MPI_Comm      comm
)
```

The first parameter, `message`, is the starting address of the data to be transmitted. The second parameter, `count`, is the number of data items, while the third parameter, `datatype`, is the type of the data items. All of the data items must be of the same type. Parameter 4, `dest`, is the rank of the process to receive the data. The fifth parameter, `tag`, is an integer "label" for the message, allowing messages serving different purposes to be identified. Finally, the sixth parameter, `comm`, indicates the communicator in which this message is being sent.

Function `MPI_Send` blocks until the message buffer is once again available. Typically the run-time system copies the message into a system buffer, enabling `MPI_Send` to return control to the caller. However, it does not have to do this.

### 6.5.2 Function `MPI_Recv`

The receiving process calls function `MPI_Recv`:

```
int MPI_Recv (
      void          *message,
      int           count,
      MPI_Datatype  datatype,
      int           source,
      int           tag,
      MPI_Comm      comm,
      MPI_Status    *status
)
```

The first parameter, `message`, is the starting address where the received data is to be stored. Parameter 2, `count`, is the maximum number of data items the receiving process is willing to receive, while parameter 3, `datatype`, is the type of the data items. The fourth parameter, `source`, is the rank of the process sending the message. The fifth parameter, `tag`, is the desired tag value for the message. Parameter 6, `comm`, identifies the communicator in which this message is being passed.

Note the seventh parameter, `status`, which appears in `MPI_Recv`, but not `MPI_Send`. Before calling `MPI_Recv`, you need to allocate a record of type `MPI_Status`. Parameter `status` is a pointer to this record, which is the only user-accessible MPI data structure.

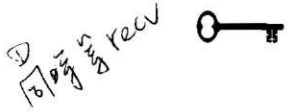
Function `MPI_Recv` blocks until the message has been received (or until an error condition causes the function to return). When function `MPI_Recv`

returns, the status record contains information about the just-completed function. In particular:

- `status->MPI_source` is the rank of the process sending the message.
- `status->MPI_tag` is the message's tag value.
- `status->MPI_ERROR` is the error condition.

Why would you need to query about the rank of the process sending the message or the message's tag value, if these values are specified as arguments to function `MPI_Recv`? The reason is that you have the option of indicating that the receiving process should receive a message from *any* process by making the constant `MPI_ANY_SOURCE` the fourth argument to the function, instead of a process number. Similarly, you can indicate that the receiving process should receive a message with any tag value by making the constant `MPI_ANY_TAG` the fifth argument to the function. In these circumstances, it may be necessary to look at the status record to find out the identity of the sending process and/or the value of the message's tag.

### 6.5.3 Deadlock





"A process is in a deadlock state if it is blocked waiting for a condition that will never become true" [3]. It is not hard to write MPI programs with calls to `MPI_Send` and `MPI_Recv` that cause processes to deadlock.

For example, consider two processes with ranks 0 and 1. Each wants to compute the average of `a` and `b`. Process 0 has an up-to-date value of `a`; process 1 has an up-to-date value of `b`. Process 0 must read `b` from 1; while process 1 must read `a` from 0. Consider this implementation:

```
float      a, b, c;
int        id;                  /* Process rank */
MPI_Status status;
...
if (id == 0) {
   MPI_Recv (&b, 1, MPI_FLOAT, 1, 0, MPI_COMM_WORLD, &status);
   MPI_Send (&a, 1, MPI_FLOAT, 1, 0, MPI_COMM_WORLD);
   c = (a + b) / 2.0;
} else if (id == 1) {
   MPI_Recv (&a, 1, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, &status);
   MPI_Send (&b, 1, MPI_FLOAT, 0, 0, MPI_COMM_WORLD);
   c = (a + b) / 2.0;
}
```

Before calling `MPI_Send`, process 0 blocks inside `MPI_Recv`, waiting for the message from process 1 to arrive. In the same way, process 1 blocks inside `MPI_Recv`, waiting for the message from process 0 to arrive. The processes are deadlocked.

Okay, that error was fairly obvious (though you might be surprised at how often this kind of bug occurs in practice). Let's consider a more subtle error that also leads to deadlock.

We're solving the same problem. Processes 0 and 1 wish to exchange floating-point values. Here is the code:

```
float      a, b, c;
int        id;                  /* Process rank */
MPI_Status status;
...
if (id == 0) {
   MPI_Send (&a, 1, MPI_FLOAT, 1, 1, MPI_COMM_WORLD);
   MPI_Recv (&b, 1, MPI_FLOAT, 1, 1, MPI_COMM_WORLD, &status);
   c = (a + b) / 2.0;
} else if (id == 1) {
   MPI_Send (&b, 1, MPI_FLOAT, 0, 0, MPI_COMM_WORLD);
   MPI_Recv (&a, 1, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, &status);
   c = (a + b) / 2.0;
}
```

Now both processes send the data before trying to receive the data, but they still deadlock. Can you see the mistake? Process 0 sends a message with tag 1 and tries to receive a message with tag 1. Meanwhile, process 1 sends a message with tag 0 and tries to receive a message with tag 0. Both processes will block inside `MPI_Recv`, because neither process will receive a message with the proper tag.

Another common error occurs when the sending process sends the message to the wrong destination process, or when the receiving process attempts to receive the message from the wrong source process.

## 6.6 DOCUMENTING THE PARALLEL PROGRAM

We can now proceed with our parallel implementation of Floyd's algorithm. Our parallel program appears in Figure 6.9.

We use a typedef and a macro to indicate the type of matrix we are manipulating. If we decided to modify our program to find shortest paths in double-precision floating-point, rather than integer, matrices, we would only have to change these two lines as shown here:

```
typedef double dtype;
#define MPI_TYPE MPI_DOUBLE
```

Function `main` is responsible for reading and printing the original distance matrix, calling the shortest path function, and printing transformed distance matrix. Note that it checks to ensure the matrix is square. If the number of rows does not equal the number of columns, the processes collectively call function

`terminate`, which prints the appropriate error message, shuts down MPI, and terminates program execution. The source code for function `terminate` appears in Appendix B.

Now let's look at the function that actually implements Floyd's algorithm. Function `compute_shortest_paths` has four parameters: the process rank, the number of processes, a pointer to the process's portion of the distance matrix, and the size of the matrix.

Recall that during each iteration `k` of the algorithm, row `k` must be made available to every process, in order to perform the computation

```
a[i][j] = MIN(a[i][j],a[i][k]+a[k][j]);
```

```
/*
 *   Floyd's all-pairs shortest-path algorithm
 */

#include <stdio.h>
#include <mpi.h>
#include "MyMPI.h"

typedef int dtype;
#define MPI_TYPE MPI_INT

int main (int argc, char *argv[]) {
   dtype** a;          /* Doubly-subscripted array */
   dtype*  storage;    /* Local portion of array elements */
   int     i, j, k;
   int     id;         /* Process rank */
   int     m;          /* Rows in matrix */
   int     n;          /* Columns in matrix */
   int     p;          /* Number of processes */

   void compute_shortest_paths (int, int, int**, int);

   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &id);
   MPI_Comm_size (MPI_COMM_WORLD, &p);

   read_row_striped_matrix (argv[1], (void *) &a,
      (void *) &storage, MPI_TYPE, &m, &n, MPI_COMM_WORLD);

   if (m != n) terminate (id, "Matrix must be square\n");

   print_row_striped_matrix ((void **) a, MPI_TYPE, m, n,
      MPI_COMM_WORLD);
   compute_shortest_paths (id, p, (dtype **) a, n);
   print_row_striped_matrix ((void **) a, MPI_TYPE, m, n,
      MPI_COMM_WORLD);
   MPI_Finalize();
}
```

**Figure 6.9** MPI program implementing Floyd's algorithm.

```
void compute_shortest_paths (int id, int p, dtype **a, int n)
{
   int  i, j, k;
   int  offset;    /* Local index of broadcast row */
   int  root;      /* Process controlling row to be bcast */
   int* tmp;       /* Holds the broadcast row */

   tmp = (dtype *) malloc (n * sizeof(dtype));
   for (k = 0; k < n; k++) {
      root = BLOCK_OWNER(k,p,n);
      if (root == id) {
         offset = k - BLOCK_LOW(id,p,n);
         for (j = 0; j < n; j++)
            tmp[j] = a[offset][j];
      }
      MPI_Bcast (tmp, n, MPI_TYPE, root, MPI_COMM_WORLD);
      for (i = 0; i < BLOCK_SIZE(id,p,n); i++)
         for (j = 0; j < n; j++)
            a[i][j] = MIN(a[i][j],a[i][k]+tmp[j]);
   }
   free (tmp);
}
```

**Figure 6.9** (contd.) MPI program implementing Floyd's algorithm.

Every process allocates an array of `n` integers, called `tmp`, that will be used to store row `k`.

As in the sequential algorithm, the parallel algorithm has `n` iterations. During each iteration, the processes determine which process controls row `k`. This process is the root of the broadcast tree. After the call to `MPI_Bcast`, each process has a copy of row `k` in its array `tmp`. Hence the assignment shown previously becomes

```
a[i][j] = MIN(a[i][j],a[i][k]+tmp[j]);
```

## 6.7 ANALYSIS AND BENCHMARKING

It's easy to see that the sequential version of Floyd's algorithm has time complexity $\Theta(n^3)$. Let's analyze the complexity of our parallel version of Floyd's algorithm.

The innermost loop, the one that updates a single row of $A$, is identical to the innermost loop in the sequential algorithm and has time complexity $\Theta(n)$. Given a rowwise block-striped decomposition of matrix $A$, each process executes at most $\lceil n/p \rceil$ iterations of the middle loop. Hence the complexity of the inner two loops is $\Theta(n^2/p)$.

Immediately before the middle loop is the broadcast step. Passing a single message of length $n$ from one processor to another has time complexity $\Theta(n)$. Since broadcasting to $p$ processors requires $\lceil \log p \rceil$ message-passing steps, the overall time complexity of broadcasting each iteration is $\Theta(n \log p)$.

For every iteration of the outermost loop the parallel algorithm must compute the new root processor, which takes constant time. The root processor copies the correct row of $A$ to array `tmp`, which takes $\Theta(n)$ time. The outermost loop executes $n$ times.

Hence the overall time complexity of the parallel algorithm is

$$\Theta\big(n(1 + n + n \log p + n^2/p)\big) = \Theta(n^3/p + n^2 \log p)$$

Now let's come up with a prediction for the execution time of our parallel program on a commodity cluster. The parallel program requires $n$ broadcasts. Each broadcast has $\lceil \log p \rceil$ steps. Each step involves passing messages that are $4n$ bytes long. Hence the expected communication time of the parallel program is

$$n\lceil \log p \rceil(\lambda + 4n/\beta)$$

If $\chi$ is the average time needed to update a single cell, then the expected computation time of the parallel program is $n^2\lceil n/p \rceil \chi$.

Adding computation time to broadcast time gives us a simple expression for the expected execution time of the parallel algorithm:

$$n^2\lceil n/p \rceil \chi + n\lceil \log p \rceil(\lambda + 4n/\beta)$$

However, this expression will overestimate the parallel execution time, because it ignores the fact that there can be considerable overlap between computation and communication.

See Figure 6.10, which illustrates the first four iterations of Floyd's algorithm executing on four processes, each on its own processor. Assume $n \geq 16$, so process 0 is the root process for the first four iterations. During each broadcast step, process 0 sends messages to processes 2 and 1. After it has initiated these messages,

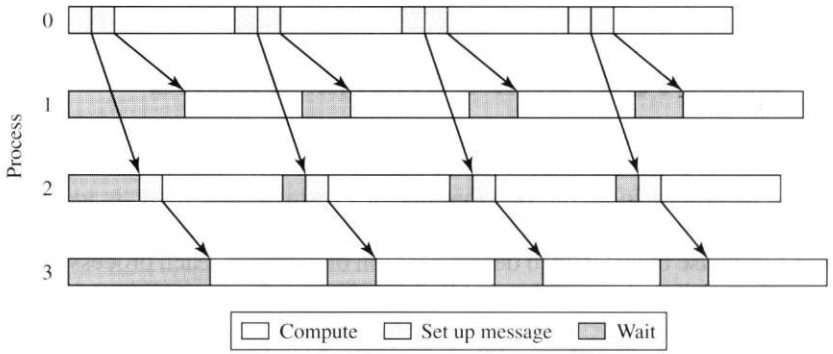



**Figure 6.10** During the execution of the parallel version of Floyd's algorithm, there is significant overlap between message transmission (indicated by arrows) and computation.

it may begin updating its share of the rows of the matrix. Communications and computations overlap.

Examine process 1. It may not begin updating its portion of the matrix until it receives row 0 from process 0. During the first iteration, it must wait for the message to show up. However, this delay offsets its computational time frame from that of process 0. Process 1 completes its iteration 1 computation after process 0. Since process 0 initiates its transmission of the second row of the matrix to process 1 while process 1 is still working with the first row, process 1 will not have as long to wait for the second row.

In the figure, computation time per iteration exceeds the time needed to pass messages. For this reason, after the first iteration each process spends the same amount of time waiting for or setting up messages: $\lceil \log p \rceil \lambda$.

If $\lceil \log p \rceil 4n/\beta < \lceil n/p \rceil n\chi$, the message transmission time after the first iteration is completely overlapped by the computation time and should not be counted toward the total execution time. This is the case on our cluster when $n = 1000$. Hence a better expression for the expected execution time of the parallel program is

$$n^2 \lceil n/p \rceil \chi + n \lceil \log p \rceil \lambda + \lceil \log p \rceil 4n/\beta$$

Figure 6.11 plots the predicted and actual execution times of our parallel program solving a problem of size 1000 on a commodity cluster, in which

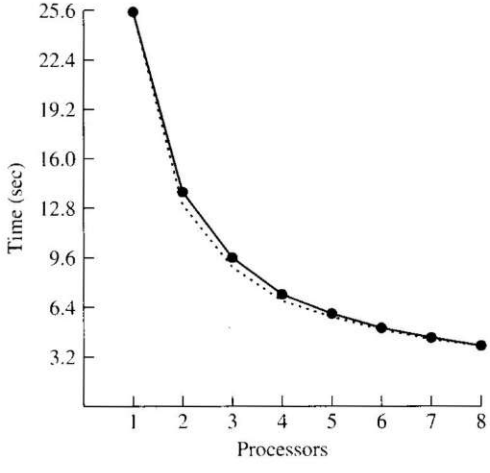



**Figure 6.11** Predicted (dotted line) and actual (solid line) execution times of parallel implementation of Floyd's algorithm on a commodity cluster, solving a problem of size 1,000.

$\chi = 25.5$ nsec, $\lambda = 250$ $\mu$sec, and $\beta = 10^7$. The average error between the predicted and actual execution times on $2, \ldots, 7$ processors is 3.8 percent.

## 6.8 SUMMARY

We have developed a parallel version of Floyd's algorithm in C with MPI. The program achieves good speedup on a commodity cluster for moderately sized matrices. Our implementation uses point-to-point messages among pairs of processors. We have introduced the local communication functions `MPI_Send` and `MPI_Recv` that support point-to-point messages.

We have also begun the development of a library of functions that will eventually support the input, output, and redistribution of matrices and vectors with a variety of data decompositions. The two input/output functions referenced in this chapter are based on a rowwise block-striped decomposition of a matrix. Function `read_row_striped_matrix` reads a matrix from a file and distributes its elements to the processes in a group. Function `print_row_striped_matrix` prints the elements of a matrix distributed among a group of processes.

## 6.9 KEY TERMS

adjacency matrix
all-pairs shortest-path problem
directed graph
graph
point-to-point communication
weighted graph

## 6.10 BIBLIOGRAPHIC NOTES

Floyd's algorithm originally appeared in the *Communications of the ACM* in 1962 [27]. It is a generalization of Warshall's transitive closure algorithm, which appeared in the *Journal of the ACM* just a few months earlier [111].

Foster compares two parallel versions of Floyd's algorithm [31]. The first agglomerates primitive tasks in the same row, resulting in a rowwise block-striped data decomposition. The second agglomerates two-dimensional blocks of primitive tasks. In the next chapter we'll see this introduced as a "block checkerboard" decomposition. Foster shows that the second design is superior.

Grama et al. also describe a parallel implementation of Floyd's algorithm based on a block checkerboard data decomposition [44].

## 6.11 EXERCISES

**6.1** Suppose we have chosen a block agglomeration of $n$ elements (labeled $0, 1, \ldots, n-1$) to $p$ processes (labeled $0, 1, \ldots, p-1$) in which

process $i$ is responsible for elements $\lfloor in/p \rfloor$ through $\lfloor (i+1)n/p \rfloor - 1$. Prove that the last process is responsible for $\lceil n/p \rceil$ elements.

**6.2** Reflect on the example of file input illustrated in Figure 6.6. What is the advantage of having process 3 input and pass along the data, rather than process 0?

**6.3** Outline the changes that would need to be made to the parallel implementation of Floyd's all-pairs shortest-path algorithm if we decided to use a columnwise block-striped data distribution.

**6.4** Outline the changes that would need to be made to the parallel implementation of Floyd's all-pairs shortest-path algorithm if we decided to use a rowwise interleaved striped decomposition (illustrated in Figure 12.3a).

**6.5** Consider another version of Floyd's algorithm based on a third data decomposition of the matrix. Suppose $p$ is a square number and $n$ is a multiple of $\sqrt{p}$. In this data decomposition, each process is responsible for a square submatrix of $A$ of size $(n/\sqrt{p}) \times (n/\sqrt{p})$.

a. Describe the communications necessary for every iteration of the outer loop of the algorithm.

b. Derive an expression for the communication time of the parallel algorithm, as a function of $n$, $p$, $\lambda$, and $\beta$.

c. Compare this communication time with the communication time of the parallel algorithm developed in this chapter.

**6.6** Suppose the cluster used for benchmarking the parallel program developed in this chapter had 16 CPUs. Estimate the execution time that would result from solving a problem of size 1000 on 16 processors.

**6.7** Assuming the same parallel computer used for the benchmarking in this chapter, estimate the execution time that would result from solving problems of size 500 and 2000 on $1, 2, \ldots, 8$ processors.

**6.8** Assume that the time needed to send an $n$-byte message is $\lambda + n/\beta$. Write a program implementing the "ping pong" test to determine $\lambda$ (latency) and $\beta$ (bandwidth) on your parallel computer. Design the program to run on exactly two processes. Process 0 records the time and then sends a message to process 1. After process 1 receives the message, it immediately sends it back to process 0. Process 0 receives the message and records the time. The elapsed time divided by 2 is the average message-passing time. Try sending messages multiple times, and experiment with messages of different lengths, to generate enough data points that you can estimate $\lambda$ and $\beta$.

**6.9** Write your own version of `MPI_Reduce` using functions `MPI_Send` and `MPI_Recv`. You may assume that

*datatype* = `MPI_INT`,
*operator* = `MPI_SUM`, and
*comm* = `MPI_COMM_WORLD`

Compare the performance of your implementation with the "real" `MPI_Reduce` in your system's MPI library.

**6.10** Using functions `MPI_Send` and `MPI_Recv`, implement your own version of function `MPI_Bcast`. Compare the performance of your implementation with the "real" `MPI_Bcast` in your system's MPI library.

**6.11** Recall the parallel Sieve of Eratosthenes program developed in Chapter 5. Lester suggests that the execution time of the program can be improved by replacing the broadcast step with a pipeline of sends and receives [71]. In other words, instead of process 0 broadcasting the next prime to all the other processes, it sends the next prime to process 1, which receives the value and sends it to process 2, and so on. For each prime found, the maximum number of communication steps per process is reduced from $\lceil \log p \rceil$ to 2.

a. Implement a new parallel version of the Sieve of Eratosthenes program, replacing the broadcast step with a receive/send step. (Process 0 does not perform a receive, and the process of highest rank does not perform a send.)

b. Use the analytical model developed in Chapter 5 to predict the execution time improvement this program will achieve.

c. Benchmark the original program and the new program on a parallel computer.

**6.12** Write two parallel programs that input a topographical map and output the locations that are in the shade given certain predetermined positions of the sun.

For both programs the format of the input and output files are the same. The input file is an $n \times n$ matrix representing $n^2$ evenly spaced points on a square piece of land. The values in the matrix are integers ranging from 0 to about 100. Each value represents the altitude of the land at the corresponding point.

The output file is an $n \times n$ matrix of 0s and 1s. The output value is 0 if the land at that point is in the sunlight and 1 if the land at the point is in the shade.

a. Your program should assume that the sun is shining from the west. A location is in the shade if there is a hill to its west high enough to block the sun. Here is how to compute which locations are in the shade: First, the locations in the leftmost column are by definition in the sunlight. For each remaining location $(i, j)$, it is in the shade if there is another location $(i, j - k)$ whose altitude is at least $4k$ greater than the altitude at $(i, j)$.

For example, here is a sample input matrix (the directional letters are to help you orient the map and are not in the file):

```
              N
   0     0    5     3     2     0
   5     7   11    10     8     4
W 15     9   16     7     8     3 E
  22    15    4    11     7     2
  10     2   10     9     7     2
   0     0    5     7     4     0
              S
```

The correct output matrix would be:

```
0    0    0    0    0    0
0    0    0    0    0    1
0    1    0    1    1    1
0    1    1    0    1    1
0    1    0    0    0    1
0    0    0    0    0    1
```

b. Your program should assume that the sun is shining from the northwest. A location is in the shade if there is a hill to its northwest high enough to block the sun. Here is how to compute which locations are in the shade. First, the locations in the top row and the leftmost column are by definition in the sunlight. For each remaining location $(i, j)$, it is in the shade if there is another location $(i - k, j - k)$ whose altitude is at least $4k$ greater than the altitude at $(i, j)$.

For example, given the same input matrix in the previous example, here is the correct output matrix:

```
0    0    0    0    0    0
0    0    0    0    0    0
0    0    0    1    0    1
0    0    1    1    0    1
0    1    1    0    1    1
0    1    1    1    1    1
```

**6.13** In 1970, Princeton mathematician John Conway invented the game of Life. Life is an example of a cellular automaton. It consists of a rectangular grid of cells. Each cell is in one of two states: alive or dead. The game consists of a number of iterations. During each iteration a dead cell with exactly three neighbors becomes a live cell. A live cell with two or three neighbors stays alive. A live cell with less than two neighbors or more than three neighbors becomes a dead cell. All cells

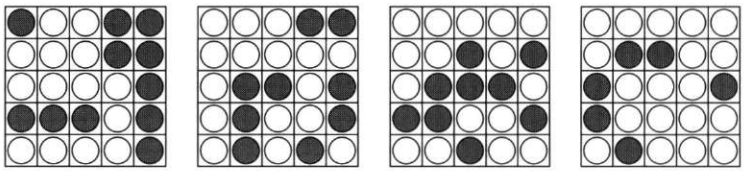


**Figure 6.12** An initial state and three iterations of Conway's game of Life.

are updated simultaneously. Figure 6.12 illustrates three iterations of Life for a small grid of cells.

Write a parallel program that reads from a file an $m \times n$ matrix containing the initial state of the game. It should play the game of Life for $j$ iterations, printing the state of the game once every $k$ iterations, where $j$ and $k$ are command-line arguments.