Chapter 5 Floyd's Algorithm

"The single biggest problem in communication is the illusion that it has taken place.." - George Bernard Shaw [1]

5.1 Introduction

The purpose of this chapter is to use a relatively easy problem as a means of introducing point-to-point communication among processes. The problem is also an interesting, practical one, known as the all-pairs, shortest-path problem. It has many different applications, but we will use the one described by Quinn [2].

Have you ever seen one of those maps that has a table whose rows and columns are labeled by the major cities on the map, and whose entries are distances between those cities? Because travel between cities is symmetric, the table is usually lower triangular (meaning only the elements below the diagonal are listed.) If not, it should not be hard to picture such a table anyway. In such tables, the distance between the cities is presumably the distance along major roads from one point to another. It could also be the distance along a route that takes the least time, or the distance along a route that uses the least fuel, or the distance along a route that is the least number of miles. (These last two need not be the same route, because if the least mileage route has traffic signals every mile, the fuel consumption will be greater than a longer one with no signals.) Regardless of which notion of distance was used to build the table, what it does is to provide us with the distance between every pair of cities in a given set of cities. Let us assume that it is the shortest distance, by whatever definition of *shortest* the cartographers had in mind.

In this chapter we will develop a parallel algorithm that could be used to create such a table from the appropriate input data. It will be based on a well-known sequential algorithm that solves this problem, Floyd's algorithm. We will also introduce the following MPI functions:

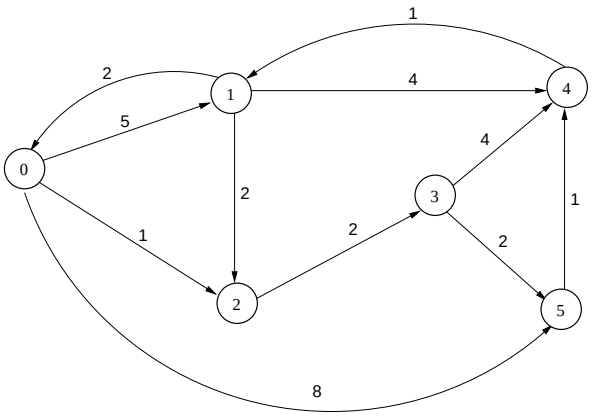
MPI Function	**Purpose**
`MPI_Send`	performs a blocking send to another process
`MPI_Recv`	performs a blocking receive from another process
`MPI_Get_count`	returns count of items received in a receive operation
`MPI_Abort`	terminates the entire MPI execution and cleans up

5.2 The All-Pairs Shortest Path Problem

Abstractly, a collection of cities with inter-city distances can be represented by a graph. For this problem, we will assume that the graph is directed. A ***directed graph*** $G =< V, E >$ is a finite set V of ***vertices*** together with an adjacency relation $E \subseteq V \times V$ on the set. If v and w are vertices such that $(v, w) = e \in E$, we say that there is a ***directed edge*** e from v to w, and we say that e is ***incident*** upon both v and w. We also say that w is ***adjacent*** to v.

The edges in a graph can have associated weights, in which case the graph is called a ***weighted graph***. We can think of the edge weights as distances between the vertices. Weighted graphs are a useful abstraction: edge weights can represent distances, costs or even capacities, as if the edges represented pipes and the weights were their diameters. In our application, edge weights are distances.

Figure 5.1 contains a weighted, directed graph. It shows that the distance from vertex 0 to vertex 1 is 5, for example, and that the direct distance from vertex 3 to vertex 4 is 4. But notice that the distance from vertex 3 to vertex 4 obtained by traveling through vertex 5 is $1 + 2 = 3$, illustrating that the shortest distance between two vertices is not necessarily the one along the least number of edges. The ***all-pairs,***



	0	1	2	3	4	5
0	0	5	1	∞	∞	8
1	2	0	2	∞	4	∞
2	∞	∞	0	2	∞	∞
3	∞	∞	∞	0	4	2
4	∞	1	∞	∞	0	∞
5	∞	∞	∞	∞	1	0

Figure 5.1: A weighted, directed graph and its adjacency matrix representation.

shortest-path problem is, given a graph G, to find the lengths of the shortest paths between every pair of vertices in the graph.

The first step in designing an algorithm to solve this problem is deciding how to represent the graph itself. There are a few different ways of representing a graph, and some are more suitable than others for particular problems. For this particular problem, the best representation is an ***adjacency matrix***. An adjacency matrix for a graph G with n vertices is an $n \times n$ matrix A such that A_{ij} is the distance from vertex i to vertex j, for $0 \leq i, j \leq n-1$. If there is no edge from vertex i to vertex j, a suitable constant is assigned to A_{ij}. It depends upon the application. For the all-pairs, shortest-path problem, conceptually we want to assign ∞ to these entries, and when we do arithmetic, we would use the rule that $\infty + c = \infty$ for all constants c. In Figure 5.1, the adjacency matrix has ∞ in all entries that correspond to non-existent edges. In an actual application, we would have to use a very large number instead.

The adjacency matrix representation uses an amount of storage proportional to $O(n^2)$. Other graph representations use storage that is asymptotically smaller. When the graph is sparse, meaning that the number of edges is small in comparison to n^2, this representation may not be suitable. But for this particular problem, the big advantage is that it is constant time access to each edge weight, and this advantage outweighs the high cost of storage. This is why we say it is the best representation in this case. A convenience of using the matrix is that we can use it to store the final, shortest distances between all pairs of vertices. In other words, when the algorithm terminates, the matrix entry A_{ij} will contain the shortest distance between vertex i and vertex j .

There are several different sequential algorithms to solve this problem; however, we will parallelize Floyd's algorithm. ***Floyd's algorithm*** is also known as the *Floyd-Warshall* algorithm. It was discovered independently by Robert Floyd and Stephen Warshall in 1962[1]. This algorithm has one restriction: it will not work correctly if there are negative cycles in the graph. A ***cycle*** in a graph is a path whose final vertex is the same as its starting vertex. A ***negative cycle*** is a cycle whose total edge weight is negative. We will assume there are no such cycles in our graph. After all, distances between cities cannot be negative. Floyd's algorithm runs in $\Theta(n^3)$ time. A pseudo-code description is in Listing 5.1 below.

Listing 5.1: Floyd's algorithm for all-pairs shortest paths.

```
1 // let A be a n by n adjacency matrix
2 for k = 0 to n-1
3     for i = 0 to n-1
4         for j = 0 to n-1
5             if ( A[i,j] > A[i,k] + A[k,j] )
6                   A[i,j] = A[i,k] + A[k,j];
7             end if
```

[1] Its history is even more complicated than this, as it was also discovered by Bernard Roy earlier, but for finding the transitive closure of a matrix.

```
8           end for
9       end for
10  end for
```

Initially, the length of the shortest path between every pair of vertices `i` and `j` is just the weight of the edge from `i` to `j` stored in `A[i,j]`. In each iteration of the outer loop, every pair of vertices `(i,j)` is checked to see whether there is a shorter path between them that goes through vertex `k` than is currently stored in the matrix. For example, when `k = 0`, the algorithm checks whether the path from `i` to `0` and then `0` to `j` is shorter than the edge `(i,j)`, and if so, `A[i,j]` is replaced by the length of that path. It repeats this for each successive value of k. With the graph in Figure 5.1, until `k = 5`, the entry `A[3,4]` has the value 4, as this is the weight of the edge `(3,4)`. But when `k = 5`, the algorithm will discover that `A[3,5] + A[5,4] < A[3,4]`, so it will replace `A[3,4]` by that sum. In the end, the lengths of the shortest paths have replaced the edge weights in the matrix. Note that this algorithm does not tell us what the shortest paths are, only what their lengths are. The result of applying Floyd's algorithm to the matrix in Figure 5.1 is shown in Figure 5.1. This particular graph has the property that there is a path from every vertex to every other vertex, which is why there are no entries with infinite distance in the resulting matrix. Such graphs are called ***strongly connected*** graphs.

	0	1	2	3	4	5
0	0	5	1	3	6	5
1	2	0	2	4	4	6
2	8	6	0	2	5	4
3	6	4	6	0	3	2
4	3	1	3	5	0	7
5	4	2	4	6	1	0

Table 5.1: The shortest-paths matrix for the graph in Figure 5.1.

5.3 Dynamic Two-Dimensional Arrays

In a problem such as the all-pairs, shortest-path problem, the input consists of the elements of the adjacency matrix for a graph. The data will be read from a file and stored into a two-dimensional array. If our program only worked on graphs of a fixed size, it would not be very useful. For it to be useful, it needs to handle arbitrary adjacency matrices, which implies that it must determine their size and allocate storage for them at runtime. In *C*, there are at least three different ways to do this, but one is much more convenient than the others. In this section, we describe them all and explain the differences. For each description below, the goal is to create a two-dimensional array of type `float` with `M` rows and `N` columns, where `M` and `N` have been obtained at runtime.

5.3.1 Creating a One-dimensional Dynamic Array

One method of creating a two-dimensional array dynamically is to create a one-dimensional array and use arithmetic to map row and column indices into it. The following code demonstrates this idea

```
/* Allocate a linear array of M * N floats */
float * A = malloc(N * M * sizeof(float));
if ( NULL == A ) {
    /* handle error and exit */
}
```

```
/* To access entry A[i][j], use A[i*N + j], as in the following:   */
for ( i = 0; i < M; i++ )
    for ( j = 0; j < N; j++ )
        scanf("%f",  A[i*N + j]);

/* When finished, free as follows: */
free(A);
```

This method is correct, but it has two problems. One is that each access to the array element requires a multiplication and an addition, which is costly in terms of time. A second problem is that it requires the programmer to do the arithmetic correctly every time. What it does guarantee though is that the rows of the two-dimensional array are stored in consecutive memory, meaning that the first element of row $i+1$ always comes immediately after the last element of row i. This is important.

5.3.2 Creating a Two-dimensional Dynamic Array

In *C*, a two-dimensional dynamic array is actually an array of one-dimensional dynamic arrays. There is no way to create a two-dimensional dynamic array directly. The way it is created is as follows.

```
/* Allocate an array of M pointers to float, one pointer per row */
float ** B = malloc (M * sizeof(float*));
if ( NULL == B ) {
     /* handle error and exit */
}
/* Now allocate M arrays of N floats each, assigning the addresses
   to B[i], for i = 0..M-1 */
for ( i = 0; i < M; i++ ) {
    B[i] = malloc (N * sizeof(float));
    if ( NULL == B[i]  )  {
        /* handle error and exit */
    }
}

/* We can access entry B[i][j] directly now, as in   */
for ( i = 0; i < M; i++ )
    for ( j = 0; j < N; j++ )
        scanf("%f",  B[i][j] );

/* When finished, free as follows: */
for ( i = 0; i < M; i++ ) {
    free( B[i] );
free(B);
```

This works correctly as well and does not have the disadvantage of requiring two arithmetic operations for each element access. However, there is no guarantee from the *C* standard that successive rows are adjacent to each other. Each time that `malloc` is called when allocating individual rows, it is free to allocate memory from wherever in the heap it sees fit. Quite often it will even create padding between the end of one row and the start of the next to align it on a multiple word boundary. This is a problem in an MPI program in which we will need to send the entire array in a single message as a contiguous sequence of bytes of memory.

5.3.3 The Best of Both Methods

The problems of both of the preceding methods can be overcome at the expense of using a bit more memory. Suppose we want to allocate memory for a matrix `C` with `nrows` rows and `ncols` columns. We do two things:

1. We allocate a contiguous chunk of memory as in the first method, containing `nrows*ncols*size` bytes, where `size` is the number of bytes in each matrix element. We call this the ***linear storage array*** for the matrix. Let `CStorage` be the pointer to its starting address in memory.
2. Then we allocate an array `C` of `nrows` pointers. These will contain the addresses of the starts of the rows in the linear storage array, `CStorage`. This is the actual 2D matrix. In other words, `C[i]` will be the start of row `i` in the linear storage array, and `C[i][j]` will be the `j`th entry in that row.
3. When we are finished, we have to free both of these arrays.

This method has the advantage that we do not have to do arithmetic to locate array entries, and that it is also contiguous in logical memory. The only extra storage is for the pointer to the linear storage array; it essentially uses the same amount of storage as the second method. The steps in C code are as follows:

```
/* Allocate an array of M*N floats  */
float * CStorage = malloc(N * M * sizeof(float));
if ( NULL == CStorage ) {
     /* handle error and exit */
}
/* Allocate an array of M pointers to float, one pointer per row */
float ** C = malloc (M * sizeof(float*));
if ( NULL == C ) {
     /* handle error and exit */
}

/* Now do not allocate more memory; just point the pointers to the row starts */
for ( i = 0; i < M; i++ ) {
    C[i] = &(CStorage[i * N]);
}

/* We can access entry C[i][j] directly now, as in   */
for ( i = 0; i < M; i++ )
    for ( j = 0; j < N; j++ )
        scanf("%f",  C[i][j] );

/* When finished, there is more to free, as follows: */
free(C);
free(CStorage);
```

A function that implements this method is in Listing 5.2 below. We will use this method in our parallel algorithm (and in all future programs that need to allocate dynamic two-dimensional arrays.) With this last method (and with the first as well), we can work with the entire array all at once if we choose. We can use the `memset` function (declared in `<string.h>`) to initialize these arrays to zeros, for example. We cannot do that with the second method.

Listing 5.2: alloc_matrix()

```
1 #include <stdlib.h>
2 #define SUCCESS          0
3 #define OUT_OF_MEM_ERR   1
4
5 /** alloc_matrix(r,c,e, &Mstorage, &M, &err)
```

```
 *  If @err is SUCCESS, on return it allocated storage for two arrays in
 *  the heap. Mstorage is a linear array large enough to hold the elements of
 *  an r by c 2D matrix whose elements are e bytes long. The other, M, is a 2D
 *  matrix such that M[i][j] is the element in row i and column j.
 */
void alloc_matrix(
        int     nrows,          /* number of rows in matrix                    */
        int     ncols,          /* number of columns in matrix                 */
        size_t  element_size,   /* number of bytes per matrix element          */
        void  **matrix_storage, /* address of linear storage array for matrix */
        void ***matrix,         /* address of start of matrix                  */
        int    *errvalue)       /* return code for error, if any               */
{
    int   i;
    void *ptr_to_row_in_storage; /* pointer to a place in linear storage array
                                    where a row begins */
    void **matrix_row_start;     /* address of a 2D matrix row start pointer
                                    e.g., address of (*matrix)[row] */


    /* Step 1: Allocate an array of nrows * ncols * element_size bytes  */
    *matrix_storage = malloc(nrows * ncols * element_size);
    if ( NULL == *matrix_storage ) {
        /* malloc failed, so set error code and quit */
        *errvalue = OUT_OF_MEM_ERR;
        return;
    }

    /* Step 2: To create the 2D matrix, first allocate an array of nrows void*
       pointers */
    *matrix = malloc (nrows * sizeof(void*));
    if ( NULL == *matrix ) {
        /* malloc failed, so set error code and quit */
        *errvalue = OUT_OF_MEM_ERR;
        return;
    }

    /* Step 3: (The hard part) We need to put the addresses into the pointers of
       the 2D matrix that correspond to the starts of rows in the linear storage
       array. The offset of each row in linear storage is a multiple of (ncols *
       element_size) bytes.  So we initialize ptr_to_row_in_storage to the start
       of the linear storage array and add (ncols * element_size) for each new row
        start. The pointers in the array of pointers to rows are of type void* so
       an increment operation on one of them advances it to the next pointer.
       Therefore, we can initialize matrix_row_start to the start of the array of
       pointers, and auto-increment it to advance it. */

    /* Get address of start of array of pointers to linear storage, which is the
       address of first pointer, (*matrix)[0]    */
    matrix_row_start = (void*) &(*matrix[0]);

    /* Get address of start of linear storage array */
    ptr_to_row_in_storage = (void*) *matrix_storage;

    /* For each matrix pointer, *matrix[i], i = 0... nrows-1,
       set it to the start of the ith row in linear storage */
    for ( i = 0; i < nrows; i++ ) {
        /* matrix_row_start is the address of (*matrix)[i] and
            ptr_to_row_in_storage is the address of the start of the  ith row in
```

```
            linear storage. Therefore, the following assignment changes the
            contents of (*matrix)[i] to store the start of the ith row in linear
            storage.  */
54          *matrix_row_start = (void*) ptr_to_row_in_storage;
55
56          /* advance both pointers */
57          matrix_row_start++;      /* next pointer in 2d array */
58          ptr_to_row_in_storage +=  ncols * element_size; /* next row */
59      }
60      *errvalue = SUCCESS;
61  }
```

5.4 Designing the Parallel Algorithm

5.4.1 Partitioning

We use Foster's design methodology to develop the algorithm. The first step is partitioning. From the pseudo-code description of Floyd's algorithm, we can see that there is little choice as to whether to use domain decomposition or functional decomposition, because the algorithm is essentially performing a single task in a deeply nested set of for-loops. This task executes an if-statement n^3 times. Hence we use domain decomposition because it is very data parallel and has no functional parallelism at all. It should be clear that we can create a task for each of the n^2 matrix elements $A[i, j]$ and let each task be responsible for updating its element.

It would not make sense to create n^3 tasks, because this would be like the boundary value problem from Chapter 3 with the value of k replacing the variable for time in that problem. In that problem we created a two-dimensional grid of tasks during partitioning but after we saw that the tasks in row i could no nothing until the tasks in row $i - 1$ had completed, we agglomerated them into a one-dimensional array of tasks. The same thing would happen in this problem if we created a three-dimensional grid. The tasks with value k could not do anything until those with value $k - 1$ terminated.

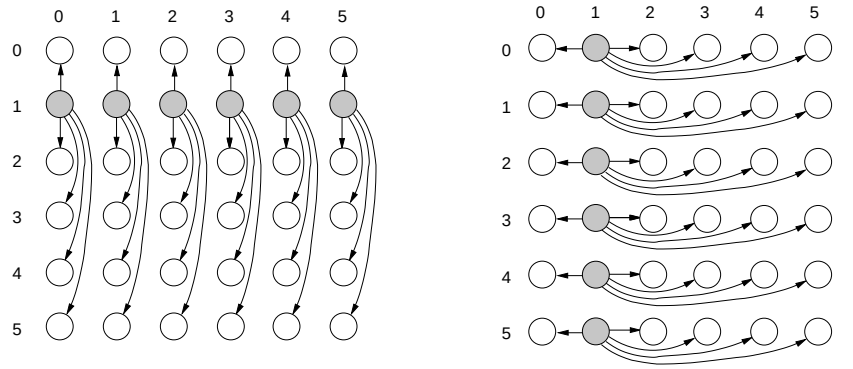
5.4.2 Communication

In the pseudo-code description of Floyd's algorithm, we see that the task responsible for updating $A[i, j]$ needs the values of $A[i, k]$ and $A[k, j]$. Using the graph from Figure 5.1 for example, when $k = 2$,

- the task responsible for $A[0, 1]$ needs access to $A[0, 2]$ and $A[2, 1]$;
- the task responsible for $A[1, 1]$ needs access to $A[1, 2]$ and $A[2, 1]$;
- the task responsible for $A[2, 1]$ needs access to $A[2, 2]$ and $A[2, 1]$;
- the task responsible for $A[3, 1]$ needs access to $A[3, 2]$ and $A[2, 1]$;
- the task responsible for $A[4, 1]$ needs access to $A[4, 2]$ and $A[2, 1]$;
- and the task responsible for $A[5, 1]$ needs access to $A[5, 2]$ and $A[2, 1]$.

Notice that $A[2, 1]$ is required by every task in column 1 of the matrix when $k = 2$. More generally, for any fixed value of k' and j, the value of $A[k', j]$ is required by every task in column j of the matrix when $k = k'$. This is symmetric with respect to rows too. Consider the tasks responsible for the elements $A[0, 0]$, $A[0, 1]$, ..., $A[0, 5]$ when $k = 2$:

- the task responsible for $A[0, 0]$ needs access to $A[0, 2]$ and $A[2, 0]$;
- the task responsible for $A[0, 1]$ needs access to $A[0, 2]$ and $A[2, 1]$;



(a) Each task in row 1 broadcasts to tasks in same column.

(b) Each task in column 1 broadcasts to tasks in same row.

Figure 5.2: Row and column broadcast communication pattern in parallelization of Floyd's algorithm

- the task responsible for $A[0,2]$ needs access to $A[0,2]$ and $A[2,2]$;
- the task responsible for $A[0,3]$ needs access to $A[0,2]$ and $A[2,3]$;
- the task responsible for $A[0,4]$ needs access to $A[0,2]$ and $A[2,4]$;
- and the task responsible for $A[0,5]$ needs access to $A[0,2]$ and $A[2,5]$.

So for any fixed value of k and i, the value of $A[i,k]$ is required by every task in row i of the matrix. These two facts imply that, during iteration k of the outer loop, each element of row k of A must be broadcast to every task in the same column as that element, and every element of column k of A must be broadcast to every task in the same row as that element. A ***broadcast*** is a global communication operation in which a single task sends a message to all processes in its communication group. This is depicted visually in Figure 5.2. The figure shows the two broadcasts that take place during the iteration of the outer loop when $k = 1$: row 1 and column 1 are each broadcast as illustrated. As k is incremented, the shaded (broadcast) row moves downward and the shaded (broadcast) column moves rightward. In other words, the communication pattern changes with every iteration of the loop.

So far, the idea is that each task will execute a loop of the form

```
for k = 0 to n-1
    if ( A[i,j] > A[i,k] + A[k,j] )
         A[i,j] = A[i,k] + A[k,j];
    end if
end for
```

To simplify this a bit, we will replace the if-statement with a call to a *min* function[2], which is computationally equivalent:

```
for k = 0 to n-1
    A[i,j] = min(A[i,j], A[i,k] + A[k,j]);
end for
```

We have not yet put the code in here to broadcast the k^{th} row and column before the updates that each task performs to its matrix element. The first question is whether it is correct for every element of the matrix to be updated simultaneously. In other words, will the final result be the lengths of the shortest paths if we allow simultaneous updates to different matrix elements such as $A[1,3]$ and $A[2,3]$? Isn't it possible that

[2]We will use a macro to implement *min*, since it is much faster in terms of execution time.

the update to $A[1,3]$ changes in such a way that the update to $A[2,3]$ would be incorrect unless the second update waits for the first to finish?

Fortunately, the answer to this question is no. To see this, consider the update to be executed in some iteration k:

$$A[i,j] = \min(\, A[i,j],\ A[i,k] + A[k,j]\,); \tag{5.1}$$

This shows that the update to $A[i,j]$ in iteration k requires the values of $A[i,k]$ and $A[k,j]$. Each of these is being updated simultaneously by two other tasks. Can the change in their values lead to an incorrect shortest path length? Consider first how $A[i,k]$ is updated. The statement that updates it in this iteration is

$$A[i,k] = \min(\, A[i,k],\ A[i,k] + A[k,k]\,);$$

This is where the assumption of no negative cycles in the graph is needed. There are no negative cycles in the graph, which means that there is no path from vertex k back to vertex k whose total weight is negative. Therefore $0 \leq A[k,k]$[3]. This implies that $A[i,k] \leq A[i,k] + A[k,k]$, implying that $A[i,k]$ retains its current value. Now consider how $A[k,j]$ gets updated in this iteration. The statement that updates it in this iteration is

$$A[k,j] = \min(\, A[k,j],\ A[k,k] + A[k,j]\,);$$

Again, because $0 \leq A[k,k]$, $A[k,j]$ also retains its current value. Now returning to Equation 5.1, if neither of $A[i,k]$ or $A[k,j]$ can change its value in this iteration, then the updates can take place simultaneously without altering the final result. This implies that it is correct to perform the broadcast and then let every task simultaneously update its matrix element.

5.4.3 Agglomeration and Mapping

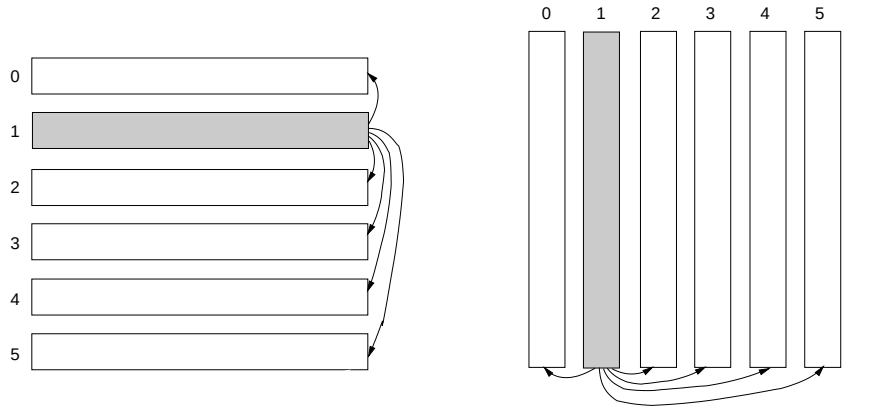
The next step is to determine whether we should agglomerate tasks to reduce communication and how to map them to processors. Once again, we use the decision tree from Chapter 3. Since the number of tasks is static, the communication pattern is structured, and each task has the same amount of computation, we should agglomerate to reduce the communication overhead, and create one task per processor. We will create one task for each physical processor and assign multiple matrix elements to it as needed. Each task will become an MPI process.

Assume we have p processors. We have to agglomerate n^2 primitive tasks into p tasks. There are some obvious ways to do this. We could agglomerate tasks in adjacent rows, or agglomerate tasks in adjacent columns. These are two different decompositions. We could also agglomerate tasks that form sub-matrices of the original matrix. We save that method of agglomeration for a later chapter and consider the first two methods.

5.4.3.1 Choice 1: Row-wise Agglomerating

If we assign adjacent rows of the matrix to each of p tasks, then the broadcast operations that delivered matrix elements to all tasks in the same row are eliminated, because these just become accesses to local memory in the task. The broadcast operations that delivered matrix elements to all tasks in the same columns are not eliminated. If p divides n evenly, then each task gets n/p adjacent rows. If not, then some tasks get $\lfloor n/p \rfloor$ rows and others get $\lceil n/p \rceil$ rows. We will describe the details of this distribution shortly.

[3] In our example graph, there were no edges from a vertex to itself, but in some graphs there may be, in which case the distance from a vertex to itself could be positive.



(a) Broadcast when $k = 1$ in row-wise agglomeration.

(b) Broadcast when $k = 1$ in a column-wise agglomeration.

Figure 5.3: Broadcasts in two different agglomerations.

Either way, in each iteration of the outer loop, exactly one row must be broadcast to all other tasks. (This is illustrated in Figure 5.3a.) Assuming that the binomial tree model of broadcast is used, it takes $\lceil \log p \rceil$ time to broadcast a message to all processes. Each message has a latency of λ. The transfer time is the size of the message divided by the bandwidth of the channels, so it is n/β. The total communication time in each iteration is $\lceil \log p \rceil (\lambda + n/\beta)$.

5.4.3.2 Choice 2: Column-wise Agglomeration

If we assign adjacent columns of the matrix to each of p tasks, then the broadcast operations that delivered matrix elements to all tasks in the same column are eliminated, because these just become accesses to local memory in the task. The broadcast operations that delivered matrix elements to all tasks in the same rows are not eliminated. (This is illustrated in Figure 5.3b.) The analysis is symmetric and the communication time per iteration is the same: $\lceil \log p \rceil (\lambda + n/\beta)$.

5.4.3.3 Chosen Method

Which method do we choose? The choice is pretty much made for us because the *C* language specifies that a two-dimensional matrix should be stored in row-major order, i.e., one row after another in logical memory. If we were to choose a column-wise agglomeration, then every process would have nonconsecutive memory locations for each column, making the memory accesses inefficient and the programming more complex. Furthermore, the program will be getting the input data from a file, and the natural way to store two-dimensional matrix data in a file is by rows. If we chose to agglomerate by columns, reading that data and distributing it to the different tasks would also be harder. Thus, we will use a row-wise agglomeration.

Each process gets a consecutive sequence of rows of the matrix. A rather elegant approach to this is described by Quinn [2], and it is the one we use. For each $k = 0, 1, 2, ..., (p-2)$, process k will be responsible for rows $\lfloor kn/p \rfloor$ through $\lfloor (k+1)n/p \rfloor - 1$. Process $(p-1)$ will be responsible for the remaining rows, which will therefore be $\lceil n/p \rceil$. To make this concrete, suppose that $n = 43$ and $p = 5$. The following table shows how the rows would be distributed.

Process Rank	Starting Row	Last Row	Number of Rows
0	0	$\lfloor 43/5 \rfloor - 1 = 7$	8
1	$\lfloor 43/5 \rfloor = 8$	$\lfloor 86/5 \rfloor - 1 = 16$	9
2	$\lfloor 86/5 \rfloor = 17$	$\lfloor 129/5 \rfloor - 1 = 24$	8
3	$\lfloor 129/5 \rfloor = 25$	$\lfloor 172/5 \rfloor - 1 = 33$	9
4	$\lfloor 172/5 \rfloor = 34$	42	9

Table 5.2: Example of distribution of rows among processes. In this example, there are 43 rows and 5 processes.

Notice that some of the processes will have more rows than others. The difference will always be at most 1 - some will have$\lceil n/p \rceil$ rows, and others $\lfloor n/p \rfloor$ rows[4]. What happens to be true is

Lemma 1. *If each process $k = 0, 1, 2, ..., p-2$, is responsible for rows $\lfloor kn/p \rfloor$ through $\lfloor (k+1)n/p \rfloor - 1$, and process $p-1$ is responsible for the remaining rows, then process $p-1$ has $\lceil n/p \rceil$ rows, which is at least as large as the number of rows assigned to all other processes.*

This fact will be important when we come to matrix input.

Exercise 2. Prove that, if each process $k = 0, 1, 2, ..., p-2$, is responsible for rows $\lfloor kn/p \rfloor$ through $\lfloor (k+1)n/p \rfloor - 1$, and process $p-1$ is responsible for the remaining rows, then process $p-1$ has $\lceil n/p \rceil$ rows.

The program will need a few functions related to the method of agglomeration. Assume that the number of processes is fixed at p and the number of rows is fixed at n, and that $0 < p \leq n$.

1. Given a process rank i, return the first row assigned to it. We will call this $lb_{p,n}(i)$.
2. Given a process rank i, return the last row assigned to it. We will call this $ub_{p,n}(i)$.
3. Given a process rank i, return the total number of rows it "owns". We will call this $s_{p,n}(i)$.
4. Given a row index j, return the rank of the process to which this row is assigned. This will be $t_{p,n}(j)$.

The first three functions are easy to define.

$$lb_{p,n}(i) = \left\lfloor \frac{in}{p} \right\rfloor \tag{5.2}$$

$$ub_{p,n}(i) = \left\lfloor \frac{(i+1)n}{p} \right\rfloor - 1 \tag{5.3}$$

$$s_{p,n}(i) = \left\lfloor \frac{(i+1)n}{p} \right\rfloor - \left\lfloor \frac{in}{p} \right\rfloor \tag{5.4}$$

and lead to straightforward implementations in *C*. The following function, for example, implements $s_{p,n}(i)$:

```
inline int number_of_rows( int id, int ntotal_rows, int p )
    {
        return ( ( ( id + 1) * ntotal_rows ) / p ) -
               ( ( id *      ntotal_rows ) / p );
    }
```

We make it `inline` to reduce overhead. The fourth, $t_{p,n}(j)$, is more subtle.

[4] If $n = qp + r$, $0 < r < p$, then there will be r processes with $\lceil n/p \rceil$ rows and $p - r$ processes with $\lfloor n/p \rfloor$ rows.

Theorem 3. *The function*

$$t_{p,n}(j) = \left\lfloor \frac{(j+1)p-1}{n} \right\rfloor, \ 0 \le j \le n-1 \tag{5.5}$$

is the unique value of i such that $\lfloor in/p \rfloor \le j \le \lfloor (i+1)n/p \rfloor - 1$.

Assuming that Theorem 3 is true, then the following function can implement $t_{p,n}(j)$:

```
inline int owner(int j, int p, int nrows )
    {
        return ( p * (j+1) -1 ) / nrows;
    }
```

Proof. First observe that for any j in the interval $[0, n-1]$, there is exactly one i for which

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

because the distribution of the set of numbers from 0 to $n-1$ is a partition of the set; every number falls into exactly one such interval and no interval is empty. We take this as obvious. This proves that $t_{p,n}(j)$ is in fact a function. Next observe that the function $t_{p,n}(j)$ is a monotonically non-decreasing function of j: for all j, $t_{p,n}(j+1) \ge t_{p,n}(j)$.

Let j be an arbitrary number in the interval $[0, n-1]$ and assume that i_0 is the number such that $j_{LB} = \lfloor i_0 n/p \rfloor$ is the lower bound and $\lfloor j_{UB} = (i_0+1)n/p \rfloor - 1$ is the upper bound. If we can show that $t_{p,n}(j_{LB}) = i_0$ and $t_{p,n}(j_{UB}) = i_0$, then by the monotonicity of $t_{p,n}(j)$, it follows that for any j in between the bounds, $t_{p,n}(j) = i_0$ as well.

The fraction $i_0 n/p$ is either a whole number or not. We can write $i_0 n = qp + r$ for some integer q and some r such that $0 \le r \le p-1$. (r is just the remainder of division by p, so it must be at most $p-1$.) This means that $i_0 n/p = q + r/p$ where $0 \le r/p \le (p-1)/p$. Let $\alpha = r/p$. Then

$$j_{LB} = \left\lfloor \frac{i_0 n}{p} \right\rfloor = \frac{i_0 n}{p} - \alpha, \ 0 \le \alpha \le \frac{p-1}{p}$$

so that

$$\begin{aligned}
j_{LB} &= \frac{i_0 n}{p} - \alpha \\
\Rightarrow j_{LB} + 1 &= \frac{i_0 n}{p} + (1-\alpha) \\
\Rightarrow (j_{LB}+1)p &= i_0 n + (1-\alpha)p \\
\Rightarrow \frac{(j_{LB}+1)p}{n} &= i_0 + \frac{(1-\alpha)p}{n} \\
\Rightarrow \frac{(j_{LB}+1)p-1}{n} &= i_0 + \frac{(1-\alpha)p-1}{n}
\end{aligned}$$

If we can show that $0 \le \frac{(1-\alpha)p-1}{n} < 1$, then it follows that $t_{p,n}(j_{LB}) = i_0$. We have

$$
\begin{aligned}
& 0 \le \alpha \le \frac{p-1}{p} \\
\Rightarrow & \frac{1-p}{p} \le -\alpha \le 0 \\
\Rightarrow & 1 + \frac{1-p}{p} \le 1-\alpha \le 1 \\
\Rightarrow & \frac{1}{p} \le 1-\alpha \le 1 \\
\Rightarrow & 0 \le (1-\alpha)p - 1 \le p-1 \\
\Rightarrow & 0 \le \frac{(1-\alpha)p-1}{n} \le \frac{p-1}{n} < 1
\end{aligned}
$$

So we have proved the first part, that $t_{p,n}(j_{LB}) = i_0$. We now have to prove that $t_{p,n}(j_{UB}) = i_0$ also. By similar reasoning as before, we can write $(i_0+1)n = qp+r$ for some integers q and r such that $0 \le r \le p-1$. This implies that $(i_0+1)n/p = q + r/p$ where $0 \le r/p \le (p-1)/p$. Let $\beta = r/p$. We have

$$
j_{UB} = \left\lfloor \frac{(i_0+1)n}{p} \right\rfloor - 1 = \frac{(i_0+1)n}{p} - 1 - \beta,\ 0 \le \beta \le \frac{p-1}{p}
$$

from which we have

$$
\begin{aligned}
j_{UB} + 1 &= \frac{(i_0+1)n}{p} - \beta \\
\Rightarrow (j_{UB}+1)p &= (i_0+1)n - \beta p \\
\Rightarrow \frac{(j_{UB}+1)p}{n} &= i_0 + 1 - \frac{\beta p}{n} \\
\Rightarrow \frac{(j_{UB}+1)p-1}{n} &= i_0 + 1 - \frac{\beta p + 1}{n}
\end{aligned}
$$

If we can show that $0 \le 1 - \frac{\beta p+1}{n} < 1$, then it follows that $t_{n,p}(j_{UB}) = i_0$. Since $\beta \ge 0$, $\beta p + 1 > 0$ and $(\beta p+1)/n > 0$, implying that $1 - (\beta p+1)/n < 1$. Since $\beta \le (p-1)/p$, it follows that $\beta p \le (p-1)$ and hence $\beta p + 1 \le p$, which implies that $(\beta p+1)/n \le (p/n) \le 1$ because the number of processes is always less than n. Therefore $-(\beta p+1)/n \ge -1$ and $1 - (\beta p+1)/n \ge 0$. Together this implies $0 \le 1 - \frac{\beta p+1}{n} < 1$, so the second part is proved. This completes the proof of the theorem. □

5.4.4 Input and Output of the Matrix

We first address how the program will read the adjacency matrix data and distribute its rows to the appropriate processes. Although we could design the program so that multiple processes will read the input file, in the end, these read operations will probably be serialized by the operating system, because the blocks of the file can only be delivered one at a time by the disk drive in our architectural model (no parallel I/O devices), and we are not yet ready to learn about the complexity of parallel I/O. Therefore, we want to assign the job of reading the matrix to a single process.

Quinn describes an efficient method for doing this[2]. The idea is to let the highest ranking process of the p processes do the job. It will repeatedly read a set of adjacent rows from the file and send them to the process that is responsible for them, starting with the rows for process 0, then the rows for process 1, and so on, through process $p-2$. After it has read all of these rows and delivered them, the rows that remain are its own rows, and it reads these last rows and just keeps them.

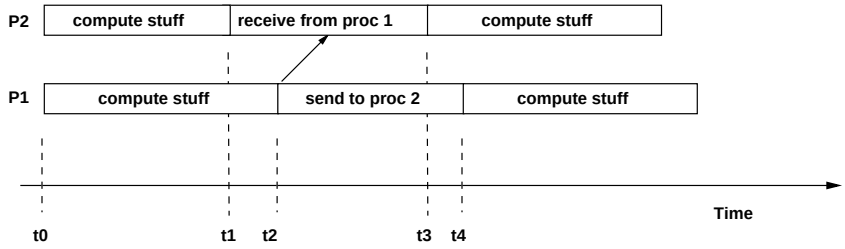
The advantage of this approach is that, because of Lemma 1, the buffer used by process $p-1$ for reading the rows and distributing them to the other processes is large enough to store the rows it has to send to the

other processes, and furthermore, when it is finished, the final set of rows just remains in the buffer. We will discuss how to code this in *C* and MPI shortly. We now turn to how to do output.

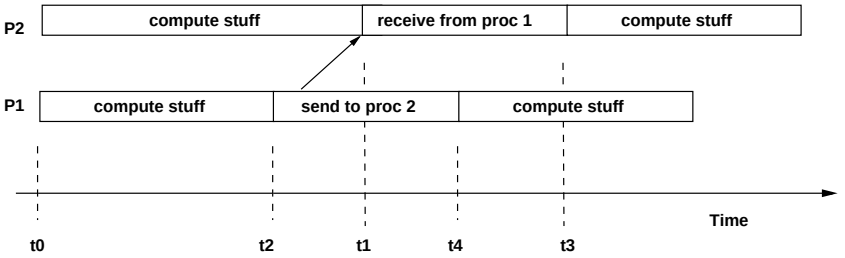
The program will output two matrices for comparison: the original adjacency matrix and the matrix of shortest paths. The output algorithm is symmetric to the input algorithm. Process 0 will be responsible for all output because we want the rows of the matrix to be printed in ascending order. It will print its own rows, and then repeatedly request the next set of rows and print them, from processes 1, 2, ..., $p-1$. It remains in charge of the communication, just as process $p-1$ was in charge during input. In other words, process 0 will request each process to send it data, rather than acting like a passive agent and letting each process send its data to it. This way it gets the data in the right order and at the right pace.

5.5 Point-to-Point Communication in MPI

A function to input or output the matrix in the manner described above will require a different type of communication primitive than what we have discussed so far. A ***point-to-point communication*** is a communication that involves just two processes, a sender and a receiver. It is different from a global or collective communication, which involves the entire group of processes in a communicator. In a point-to-point communication, one process will call a function that sends data, and the process that wants to receive it will call a function that receives it.



(a) Send is called after receive.



(b) Receive is called after send.

Figure 5.4: Two processes in a point-to-point communication. In Figure (a), the process calls *receive* before the sending process has sent anything. In (b), the sending process calls *send* before the other process has called *receive*.

There is no expectation nor any requirement that the two processes have to call the send and receive functions at the exact same time. On the contrary, it is expected that the calls can occur in any order. Figure 5.4 depicts the two most likely scenarios. In Figure 5.4a, process P2 calls *receive* before process P1 calls *send*. This is not much of a problem; when P1 calls *send*, the message is delivered to P1. But in Figure 5.4b, process P1 calls *send* before P2 has called *receive*. This is the purpose of ***system buffering***. The typical implementation of message-passing primitives provides a system buffer; when a process sends a message, it is copied into a buffer, and when the receiver requests it, it is delivered. This is not part of the MPI standard; it is just how most systems will implement it.

There is another problem. Remember that in MPI, every process executes the same program. If one process needs to call a function to send a message but another has to call a function to receive a message, then the

two different processes have to follow different control flow paths in the program. The program must have conditional branches whenever there is point-to-point communication. The basic form of such a program is

```
if ( Sender_Rank == myrank )
    call the send function
else if ( Receiver_Rank == myrank  )
    call the receive function
else ...
```

Only the process with rank `Sender_Rank` will send, and only the process with rank `Receiver_Rank` will receive. We will give a concrete example below.

MPI has a few different functions related to sending and receiving, but we will begin with the two basic ones, `MPI_Send` and `MPI_Recv`.

5.5.1 MPI_Send

The `MPI_Send` function performs what is called a ***standard-mode blocking send*** operation. We will explain what this means shortly. First, its syntax:

```
int MPI_Send(
    void *sendbuf,          /* Initial address of send buffer       */
    int count,              /* Number of elements to send           */
    MPI_Datatype datatype, /* Datatype of each send buffer element */
    int dest,               /* Rank of destination                  */
    int tag,                /* Message tag                          */
    MPI_Comm comm           /* Communicator handle                  */
)
```

Notice that this function's parameters are similar to those of `MPI_Reduce`. The first is the address of the first data item to send, and the second is a count of how many such items should be sent. The third specifies the type of each element. But the similarity ends there. The send operation must specify the rank of the process to which this data is sent in the fourth parameter, `dest`. The fifth parameter is an integer `tag` that can be used by the program as a label for the particular message. The last parameter is the communicator handle of the process group to which the sender and receiver belong.

As stated above, `MPI_Send` is a ***standard-mode blocking send operation***. A send operation is ***blocking*** if the send function call does not return control to the calling process until the message has been removed from the send buffer, so that the sender is free to modify the send buffer. This does not imply that the data has been delivered to the recipient. It just means that the message has been copied out of the send buffer and put someplace safe. It might be in an intermediate buffer, from which it will be delivered eventually to the recipient. MPI does not even require that a matching receive has been executed by the receiver. ***Standard-mode*** basically means that MPI may or may not buffer these messages; it is system dependent. You should never write a program whose correctness depends on whether or not messages are buffered.

5.5.2 MPI_Recv

A process that needs to receive data from a sending process must call `MPI_Recv`. Its syntax is

```
int MPI_Recv(
    void *recvbuf,          /* [OUT]  Initial address of receive buffer   */
    int count,              /* Maximum number of elements to receive      */
    MPI_Datatype datatype, /* Datatype of each receive buffer entry      */
    int source,             /* Rank of the message source                 */
    int tag,                /* Message tag                                */
```

```
    MPI_Comm comm,          /* Communicator handle                      */
    MPI_Status *status      /* [OUT] Status of completed call           */
)
```

Two of these parameters are labeled as `OUT` parameters; their contents are changed by the call. The first parameter, `recvbuf`, is the starting address of the received elements. If the operation was successful, the first received data item starts at that address. The second parameter is an integer that specifies the maximum number of elements that the process is willing to receive. It might be less than what the sender tries to send, or it might be more. If the arriving message is larger than the `count`, it will be truncated and result in an error condition. If it is smaller, it is not an error.

The third parameter, `datatype`, is the type of the elements to be received. The fourth is the rank of the process from which the message should be sent. There is a wild-card for this parameter. If the value `MPI_ANY_SOURCE` is passed to the source parameter, it means that a message will be accepted from any process in the communicator's process group.

The fifth parameter is an integer `tag`. If an actual tag value is specified, it means that the process will only accept a message whose sender put that particular tag value on it. If the receiving process is willing to accept any message, regardless of its tag, it can use the wild card `MPI_ANY_TAG` instead. The communicator handle is the sixth parameter, but not the last.

The last parameter is the address of a status record, of type `MPI_Status`. `MPI_Recv` can terminate either because it successfully received a message, or because of an error condition. The status parameter can be queried to see what happened. In C, `status` is a structure that contains three fields named `MPI_SOURCE`, `MPI_TAG`, and `MPI_ERROR`[5]. Thus,

- `status->MPI_SOURCE`, will contain the rank of the process that sent the received message
- `status->MPI_TAG` will contain the tag of the received message, and
- `status->MPI_ERROR` will contain the error code if it terminated because of an error.

If you really don't care to see the status, you can use the value `MPI_STATUS_IGNORE` as an argument instead of the address of a status structure.

As mentioned above, the actual number of data elements in the message is not necessarily the specified count. The process can call `MPI_Get_count` to get the number of data elements actually received. Its syntax is

```
int MPI_Get_count(
    MPI_Status *status,     /* status returned by MPI_Recv  */
    MPI_Datatype datatype,  /* datatype passed to MPI_Recv  */
    int *count              /* [OUT]  count of elements received */
)
```

The actual count is stored in the third parameter.

`MPI_Recv` is a standard-mode blocking receive operation. The blocking semantics are simpler than those of `MPI_Send`: `MPI_Recv` returns only after the receive buffer contains the newly received message. This is like the default behavior of the input functions from the C/C++ libraries: they block the process until the read operation completes.

There is an asymmetry between sending and receiving. For one, the sender must specify the destination, but not *vice versa*. For another, the receiver does not need to specify the message tag, whereas the sender must tag the message. Third, the notion of blocking is different.

[5] The structure may contain additional fields.

5.5.3 Example

The program in Listing 5.3 demonstrates how two processes can execute the same program, with one sending to the other. It makes process 0 the sender and process 1 the receiver. The receiver uses `MPI_ANY_SOURCE` and `MPI_ANY_TAG` instead of binding a specific process and message tag statically. It inspects the status structure for the rank of the sending process and the tag of the message. Notice also that to send a string, it is sent with `MPI_CHAR` as the type, and the length of the string plus one is the count, so that the terminating null byte is sent. The receiving process specifies a buffer large enough to hold the sender's string. This is an MPI version of a hello-world program. Compile it and then run it using the command

```
$ mpirun -np 2 send_recv_demo
```

Listing 5.3: send_recv_demo.c

```
1 #include <string.h>
2 #include <stdio.h>
3 #include "mpi.h"
4
5 #define  BUFFERMAX  20
6
7 int main( int argc, char **argv )
8 {
9     char message[BUFFERMAX]; /* message to send */
10    int id;                  /* rank of executing process */
11    MPI_Status status;       /* status to pass to MPI_Recv */
12    int        tag = 10;     /* a random tag value */
13
14
15    MPI_Init( &argc, &argv );
16    MPI_Comm_rank( MPI_COMM_WORLD, &id );
17    if (id == 0) { /* process 0 sends */
18        strcpy(message,"Hello, world");
19        MPI_Send(message, strlen(message)+1, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
20    }
21    else if (id == 1) { /* process 1 receives from any process any tag */
22        MPI_Recv(message, BUFFERMAX, MPI_CHAR,
23                 MPI_ANY_SOURCE,
24                 MPI_ANY_TAG,
25                 MPI_COMM_WORLD, &status);
26
27        printf("\"%s,\" from process %d", message, id);
28        printf("  (Sender is process %d; tag = %d)\n",
29                status.MPI_SOURCE, status.MPI_TAG);
30
31    }
32    MPI_Finalize();
33    return 0;
34 }
```

5.5.4 Deadlock

You have to be careful when using `MPI_Send` and `MPI_Recv` because these are blocking operations and if you use them in an incorrect way, your program can ***deadlock***. Informally, ***deadlock*** is a state of a system of parallel processes in which two or more processes in the system are waiting for the other to do something to release it from waiting, but none of these processes are able to release the other because they are all waiting. Deadlock always involves at least two processes and has some form of circular waiting, such as process 1

waiting for process 2, which waits for process 3 which waits for process 1. Misusing `MPI_Send` and `MPI_Recv` can lead to deadlock, as in

```
int r = 1, c = 1;
int m, n = 1;
/* do stuff here */
if ( 0 == id ) {
    MPI_Recv(&r, 1, MPI_INT,  1, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    m = some_function_of(r);
    MPI_Send(&m, 1, MPI_INT,  1, 2, MPI_COMM_WORLD);
}
else if ( 1 == id ) {
    MPI_Recv(&c, 1, MPI_INT,  0, 2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Send(&n, 1, MPI_INT,  0, 1, MPI_COMM_WORLD);
}
/* other code here .... */
```

Each of the two processes calls `MPI_Recv` first. Because `MPI_Recv` blocks the caller, neither will ever reach the `MPI_Send` call to unblock the other. This is a deadlock. Switching the order of the calls that process 1 makes will solve the problem, because process 1 will send to process 0, and then wait for the message from process 0.

Deadlock can also occur if a process waits for a message with the wrong tag value, or if the wrong destination process was named in the call.

5.5.5 Other Pitfalls Related to Point-to-Point Communication

Parallel programming is fraught with peril; you must be very careful in general, but with respect to what we have just covered, here are a few things to keep in mind. If you run the `send_recv_demo` program from Listing 5.3 using the command

```
$ mpirun -np 1 send_recv_demo
```

you will get an error of the following form (this was run on a host named `harpo`):

```
[harpo:16512] *** An error occurred in MPI_Send
[harpo:16512] *** on communicator MPI_COMM_WORLD
[harpo:16512] *** MPI_ERR_RANK: invalid rank
[harpo:16512] *** MPI_ERRORS_ARE_FATAL (your MPI job will now abort)
--------------------------------------------------------------------------
mpirun has exited due to process rank 0 with PID 16512 on
node harpo exiting without calling "finalize". This may
have caused other processes in the application to be
terminated by signals sent by mpirun (as reported here).
```

The problem, of course, is that process 0 tries to send to process 1 but there is no process with that rank. Remember to run your parallel program with at least as many processes as the program hard-codes. If you have conditional instructions that contain hard-coded ranks, these must exist!

If you comment out the code that process 1 is supposed to execute and run the `send_recv_demo`, it will run successfully but with no output. The call to `MPI_Send` will return without error; this is a reminder that a sender does not fail just because there is no receiver. The message is sent but never received and the program does not detect this.

On the other hand, if you comment out the call to `MPI_Send`, process 1 will never return from `MPI_Recv` and thereby hang the program. This is not deadlock; it is a form of ***infinite waiting***, which is waiting for an event that will never happen.

5.6 Matrix Input

We are now in a position to implement a function to read the adjacency matrix from a file and distribute it to the processes in the program. The pseudo-code description follows. We will assume that the matrix input function has the following prototype:

```
void read_and_distribute_matrix (
     char        *filename,       /* [IN]  name of file to read                   */
     void      ***matrix,         /* [OUT] address of matrix to fill with data    */
     void       **matrix_storage, /* [OUT] address of linear storage for  matrix  */
     MPI_Datatype dtype,          /* [IN]  matrix element type                    */
     int         *nrows,          /* [OUT] number of rows in matrix               */
     int         *ncols,          /* [OUT] number of columns in matrix            */
     int         *errval,         /* [OUT] success/error code on return           */
     MPI_Comm     comm)           /* [IN] communicator handle                     */
```

The function `read_and_distribute_matrix` will attempt to open the given filename for reading. If it fails, it will set a nonzero error code. If it is successful, the number of rows and columns in the matrix are read, each process allocates the memory it needs for its part of the matrix, and then the matrix data from the file is read and distributed among the processes in the communicator group using the method described in Section 5.4.4 above. If it fails for any reason, it will set a nonzero error code. If it is successful, the `errval` will be set to 0.

This function will be executed by every process. Therefore, since process $p-1$ alone will be handling the file input and distribution of the rows, there must be conditional statements guarding its portions of the code. The pseudo-code is in Listing 5.4 below, after which the fine points are discussed.

Listing 5.4: Pseudo-code for `read_and_distribute_matrix.c`

```
1  int id           = process rank process;
2  int p            = number of processes;
3  int elementsize = number of bytes in element type;
4  size_t nlocal_rows; /* will be number of rows belonging to the calling process */
5
6  if ( p-1 == id ) {
7      open the binary file named by filename, which contains the matrix.
8      if  failure
9          set an error code and set *nrows and *ncols to zero.
10     otherwise
11         read the first two numbers, assigning to *nrows and *ncols respectively.
12 }
13 Do a broadcast from process p-1 of the number *nrows to all other processes.
14 if ( 0 == *nrows )
15     process must exit -- matrix was not read correctly.
16
17 Do a broadcast from process p-1 of the number *ncols to all other processes.
18 if ( 0 == *ncols )
19     process must exit -- matrix was not read correctly.
20
21 nlocal_rows = number_of_rows(calling process rank).
22
23 Allocate 2D matrix with nlocal_rows rows, ncols columns, with elementsize bytes
       per entry, assigning to *matrix and *matrix_storage. /* Each calling process is
        therefore creating just a slice of the total matrix, the full width, but just
       its rows. */
24
25 if ( p-1 == id ) {
26     For each process i, for i = 0 up to p-2 {
```

```
27          Read a consecutive chunk of bytes from the file that contains the rows for
                process i into the storage allocated by process p-1 for itself.
28            Use MPI_Send to send the chunk to process i.
29      }
30      Read the remainder of the file into  the linear storage allocated for itself (
            process p-1).
31  }
32  otherwise {
33      Call MPI_Recv to receive the data that process p-1 sends to it and store it
            into the local storage allocated above.
```

Notes

- Because this function allocates memory for the matrix and its linear storage array, and the addresses for these are determined by the call to `malloc` in the function itself, the parameters for the matrix and its storage array are the addresses returned by `malloc`. Therefore, `matrix` is a `(void***)` parameter, and `matrix_storage` is a `(void**)` parameter. By making them `void***` and `void**` respectively, we can use this function to create matrices of any underlying element type; i.e., it is generic in a sense.
- The function `number_of_rows`, was defined in Section 5.4.3.3.
- We will also need a function, which we call `get_size`, with a parameter of type `MPI_Datatype`, that returns the number of bytes in an object of that type. For example, `get_size(MPI_INT)` will return the number of bytes in an `int` on the given machine. Note that `MPI_Datatype` is not an integer type and that the constants `MPI_INT`, `MPI_CHAR`, etc, may be pointers to structures. This implies that a `switch` statement cannot be used to implement `get_size`. Each process will execute the statement

  ```
  element_size = get_size (dtype);
  ```

  to get the element size.
- To implement line 23 in the pseudo-code above, each process will call the `alloc_matrix` function shown in Listing 5.2 as follows:

  ```
  alloc_matrix( nlocal_rows, *ncols, element_size,
                matrix_storage,
                matrix,
                errval);
  ```

  The variable `nlocal_rows` is the number of rows for which the process is responsible, `*ncols` is the width of the matrix, `element_size` is the number of bytes in each matrix element. The `matrix_storage` and `matrix` parameters will be filled with the addresses supplied by `alloc_matrix` when it allocates memory for the process. The `errval` parameter will contain an error code if it fails.
- All of these functions will be in a separate source file that we call `utilities.c`, with header `utilities.h`.

5.6.1 MPI_Abort

This function will use `MPI_Abort` if it encounters an unrecoverable error. Its syntax is

```
int MPI_Abort(MPI_Comm comm, int errorcode)
```

`MPI_Abort` is called with the communicator group and an integer error code. It will terminate every task in the communicator group. Although it is intended to pass the error code to the calling environment, it will not in most implementations, so you will not be able to see the error from the shell that calls the MPI program, most likely. We use `MPI_Abort` only when there is a chance that some processes may not receive the same error as others, but no processes should proceed if any processes get the error. For example, if one process is unable to allocate memory but the others did, then the program should be aborted.

5.6.2 Function `read_and_distribute_matrix`

The complete function is in the following listing. You should read the comments to understand the logic not explained in the preceding section.

Listing 5.5: read_and_distribute_matrix()

```
#include <stdio.h>
#include <mpi.h>
#include "utilities.h"

void read_and_distribute_matrix (
          char         *filename,        /* [IN]  name of file to read          */
          void       ***matrix,          /* [OUT] matrix to fill with data      */
          void        **matrix_storage,  /* [OUT] linear storage for the matrix */
          MPI_Datatype dtype,           /* [IN]  matrix element type           */
          int          *nrows,          /* [OUT] number of rows in matrix      */
          int          *ncols,          /* [OUT] number of columns in matrix   */
          int          *errval,         /* [OUT] sucess/error code on return   */
          MPI_Comm      comm)           /* [IN] communicator handle            */
{

    int    id;              /* process rank process */
    int    p;               /* number of processes  in communicator group */
    size_t element_size;    /* number of bytes in matrix element type */
    int    mpi_initialized; /* flag to check if MPI_Init was called already */
    FILE   *file;           /* input file stream pointer */
    int    nlocal_rows;     /* number of rows calling process "owns" */
    MPI_Status   status;    /* result of MPI_Recv */
    const int    MSG_TAG=1;
    /* Make sure we are being called by a program that init-ed MPI */
    MPI_Initialized(&mpi_initialized);
    if ( !mpi_initialized ) {
       *errval = -1;
       return;
    }

    /* Get process rank and the number of processes in group */
    MPI_Comm_size (comm, &p);
    MPI_Comm_rank (comm, &id);

    /* Get the number of bytes in a matrix element */
    element_size = get_size (dtype);
    if ( element_size <= 0 ) {
       *errval = -1;
       return;
    }

    if ( p-1 == id ) {
        /* Process p-1 opens the binary file containing the matrix and  reads the
           first two numbers, which are the number of rows and  columns
           respectively. */
        file = fopen (filename, "r");
        if ( NULL == file ) {
            *nrows = 0;
            *ncols = 0;
        }
        else {
            fread (nrows, sizeof(int), 1, file);
            fread (ncols, sizeof(int), 1, file);
```

```
52          }
53      }
54
55      /* Process p-1  broadcasts the numbers of rows to all other processes. */
56      MPI_Bcast (nrows, 1, MPI_INT, p-1, comm);
57
58      if ( 0 == *nrows  ) {
59         *errval = -1;
60         return;
61      }
62
63      /* Process p-1  broadcasts the numbers of columns to all other processes. */
64      MPI_Bcast (ncols, 1, MPI_INT, p-1, comm);
65
66      /* Each process sets local_rows = the number of rows the process owns. The
           number of rows depends on id, *nrows, and p. It is the difference between
           the high address and the low address. */
67      nlocal_rows = number_of_rows( id, *nrows, p );
68
69      /* Each process creates its linear storage and 2D matrix for accessing the
           elements of its assigned rows. */
70      alloc_matrix( nlocal_rows, *ncols, element_size,
71                    matrix_storage,
72                    matrix,
73                    errval);
74      if ( SUCCESS != *errval ) {
75           MPI_Abort (comm, *errval);
76      }
77
78      if ( p-1 == id  ) {
79           int nrows_to_send;      /* number of rows that p-1 sends to a process */
80           int num_elements;       /* total number of matrix elements to send */
81           size_t nelements_read; /* result of read operation */
82           int i;                  /* loop index */
83
84           /* For each process i, for i = 0 up to p-2, process p-1 reads a
                consecutive chunk of bytes from the file that contains the rows for
                process i and then sends that chunk of bytes to process i. */
85           for ( i = 0; i < p-1; i++) {
86                nrows_to_send  = number_of_rows( i, *nrows, p );
87                num_elements   = nrows_to_send * (*ncols);
88                nelements_read = fread (*matrix_storage, element_size,
89                                        num_elements, file );
90
91                /* Check that the number of items read matches the number requested.
                      If not, abort. */
92                if ( nelements_read != num_elements )
93                     MPI_Abort(comm, FILE_READ_ERROR);
94                MPI_Send (*matrix_storage, num_elements, dtype,
95                          i, MSG_TAG, comm);
96           }
97           /* Process p-1 reads the remainder of the file into its own
98              linear storage. */
99           nelements_read = fread (*matrix_storage, element_size,
100                                   nlocal_rows * (*ncols), file);
101          /* Check that the number of items read matches the number requested */
102          if ( nelements_read != nlocal_rows * (*ncols) )
103               MPI_Abort(comm, FILE_READ_ERROR);
104          /* Process p-1 closes the file */
```

```
105         fclose (file);
106     }
107     else /* what all other processes do */
108         /* store the data sent by p-1 into the linear storage array for *matrix,
               which is *matrix_storage, not **matrix_storage!  The number of values
               expected is the number of rows for this process times matrix width. */
109         MPI_Recv (*matrix_storage, nlocal_rows * (*ncols),
110                   dtype, p-1, MSG_TAG, comm, &status);
111 }
```

5.7 Matrix Output

A program is pretty useless if it cannot output its results. We need a function that can output the matrix that is computed by our parallel version of the all-pairs, shortest-paths algorithm. The output problem is somewhat symmetric to the input problem. The pieces of the matrix are held by different processes, and they need to be output by process 0, as noted in Section 5.4.4. The function must collect the different sets of rows from processes 1, 2, up to $p-1$ onto process 0, which will print them. In the matrix input routine, process $p-1$ did the reading and sent data to the other processes. In this routine, process 0 must receive data sent by other processes and then print it.

There is a fundamental difference between sending and receiving. A process that sends decides when it sends and so controls the pace at which data transfers take place. A process that receives has no control over when data is sent to it, and so it has no control over the data that is sent to it. If all of the other processes sent their data at the same time, the communication channel could be inundated with so much data that it had no buffer space available, causing needless delays and making the processor run slowly. Therefore, it is better if the process that has to collect the data does so proactively, by issuing requests for it. In short, a type of handshake can be used, of the form

```
collector process (rank=0) code:
    MPI_Send (&dummydata, 1, MPI_INT, i, REQUEST_TAG, comm);
    MPI_Recv (&expecteddata, num_items_expected, item_type, i,
              RESPONSE_TAG, comm, &status);
    print the received data;

sending process (rank=i) code:
    MPI_Recv (&dummydata, 1, MPI_INT, 0, REQUEST_TAG, comm, &status);
    MPI_Send (&data_for_printing, num_items_expected, item_type,
              0, RESPONSE_TAG, comm);
```

The collector first sends a message that acts like a synchronizing signal to process i; when process i receives it, it sends the data that process 0 is expecting; process 0 has immediately executed the call to wait for that data, so when it arrives, it can print it. This would be in a loop that process 0 executes.

The function to perform the collection and printing of the matrix will be

```
void collect_and_print_matrix (
          void       **matrix,   /* [IN] matrix to print             */
          MPI_Datatype dtype,    /* [IN] matrix element type         */
          int          nrows,    /* [IN] number of rows in matrix    */
          int          ncols,    /* [IN] number of columns in matrix */
          MPI_Comm     comm);    /* [IN] communicator handle         */
```

Every process will execute it. Each will pass the portion of the matrix that it controls in the first argument, the element type (`dtype`) in the second, the number of rows it has in the third, and the matrix number of columns in the fourth. The communicator is the last argument. It will make use of an auxiliary function that can print a matrix of numbers to an arbitrary file stream. It is in the following listing.

Listing 5.6: print_matrix()

```
void print_matrix (
        void  **matrix,          /* matrix to be printed         */
        int     nrows,           /* number of rows in matrix     */
        int     ncols,           /* number of columns in matrix  */
        MPI_Datatype dtype,      /* MPI type                     */
        FILE  *stream)           /* stream on which to print     */
{
    int i, j;

    for (i = 0; i < nrows; i++) {
        for (j = 0; j < ncols; j++) {
            if (dtype == MPI_DOUBLE)
                fprintf (stream, "%6.3f ", ((double **)matrix)[i][j]);
            else  if (dtype == MPI_FLOAT)
                fprintf (stream,"%6.3f ", ((float **)matrix)[i][j]);
            else if (dtype == MPI_INT)
                fprintf (stream,"%6d ", ((int **)matrix)[i][j]);
        }
        fprintf(stream,"\n");
   }
}
```

The implementation of `collect_and_print_matrix` follows.

Listing 5.7: collect_and_print_matrix()

```
void collect_and_print_matrix (
          void       **matrix,    /* [IN] matrix to print              */
          MPI_Datatype dtype,     /* [IN] matrix element type         */
          int          nrows,     /* [IN] number of rows in matrix    */
          int          ncols,     /* [IN] number of columns in matrix */
          MPI_Comm     comm)      /* [IN] communicator handle         */
{
    int    id;              /* process rank process                          */
    int    p;               /* number of processes  in communicator group    */
    size_t element_size;    /* number of bytes in matrix element type        */
    int    nlocal_rows;     /* number of rows calling process "owns"         */
    MPI_Status   status;    /* result of MPI_Recv                            */
    void **submatrix_buffer;/* matrix to hold submatrices sent by processes */
    void  *buffer_storage;  /* linear storage for submatrix_buffer           */
    int    max_num_rows;    /* largest number of rows of any process         */
    int    prompt;          /* synchronizing variable                        */
    int    errval;          /* to hold error values                          */

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

    nlocal_rows = number_of_rows( id, nrows, p );

    if ( 0 == id ) {
        int i;

        /* Process 0 prints its rows first. */
        print_matrix (matrix, nlocal_rows, ncols, dtype, stdout);
        if (p > 1) {
            /* Get the number of bytes in a matrix element */
            element_size = get_size (dtype);

```

```
33              /* Get the maximum number of rows used by any process, which is the
                    number process p-1 uses. */
34              max_num_rows = number_of_rows( p-1, nrows, p );
35
36              /* Allocate the 2D matrix and backing linear storage to hold
37                 arrays received from remaining processes */
38              alloc_matrix( max_num_rows, ncols, element_size,
39                    &buffer_storage,
40                    &submatrix_buffer,
41                    &errval);
42              if ( SUCCESS != errval ) {
43                   MPI_Abort (comm, errval);
44              }
45
46              /* Request each other process to send its rows. Rather than just
                    printing what it receives, which might flood the processor on which
                     process 0 is running, process 0 prompts each other process to send
                     its data and then waits for it. This is a form of lock-step
                    synchronization */
47
48              for (i = 1; i < p; i++) {
49                   /* Calculate the number of elements to be received from
50                      process i */
51                   int num_rows     = number_of_rows( i, nrows, p );
52                   int num_elements = num_rows * ncols;
53
54                   /* Send a message to process i telling it to send data */
55                   MPI_Send (&prompt, 1, MPI_INT, i, PROMPT_MSG, comm);
56
57                   /* Wait for data to arrive from proess i */
58                   MPI_Recv (buffer_storage, num_elements, dtype,
59                             i, RESPONSE_MSG, comm, &status);
60
61                   /* Print the matrix just received */
62                   print_matrix (submatrix_buffer,num_rows, ncols, dtype, stdout);
63              }
64              /* Free the allocated memory */
65              free (submatrix_buffer);
66              free (buffer_storage);
67         }
68         fprintf(stdout, "\n");
69     }
70     else {
71         /* Wait for prompt message from process 0 */
72         MPI_Recv (&prompt, 1, MPI_INT, 0, PROMPT_MSG, comm, &status);
73
74         /* On receiving it, send the matrix received by the call, which is
75            the set of rows belonging to the process that called this
76            function. */
77         MPI_Send (*matrix, nlocal_rows * ncols, dtype, 0, RESPONSE_MSG, comm);
78    }
79}
```

5.8 The Parallel Version of Floyd's Algorithm

We are now ready to examine the parallel version of Floyd's algorithm, based on the previous discussions. It is essentially the version that Quinn provides in his chapter on the algorithm [2]. The actual computation of

shortest paths is in the `compute_shortest_paths` function, the function that all processes call to compute their portion of the shortest-paths matrix. The pseudo-code description follows.

Listing 5.8: Pseudo-code for compute_shortest_paths()

```
Allocate a linear array named temp_row large enough to store a row of the
    adjacency matrix.
For k = 0 to n-1 do
{
    Determine which process is responsible for row k and call it the root process.
    If the executing process's id == root, (there is exactly one process for which
         this will be true )
        It then determines the index of row k relative to its starting row. (
            Remember that it has just a subset of the rows of the entire matrix.
            For example, if k = 7 and it holds rows 5 through 9, then row k has
            index 2 in this process.
        It copies its row (which is really row k of the whole matrix)  into the
            temp_row.
    All processes participate in the broadcast by the root process of this row.
        Remember that exactly one process will be root. The broadcast will send
        temp_row to all other processes.
    Every process will execute the doubly-nested loop of the sequential algorithm
        for its set of rows, using the temp_row in the update step:
        for (i = 0; i < number_of_rows_of_this_process; i++)
             for (j = 0; j < n; j++)
                  a[i][j] = MIN(a[i][j],a[i][k]+temp_row[j]);
    Because temp_row is actually row k of the matrix, the update in the loop body
        is equivalent to
                  a[i][j] = MIN(a[i][j],a[i][k]+a[k][j]);
}
```

The *C* code for this is in Listing 5.9. Most of the code is self-explanatory. The function that determines the owner of the row to be distributed was defined in Section 5.4.3.3. This implementation includes code to compute the timing, but not quite like the way it was done in the circuit satisfiability program from Chapter 4. In that program, the time that was printed was the time used by process 0. In this program we check with process took the longest time and print that as the running time. That can be accomplished with the `MPI_Reduce` function and the `MPI_MAX` operator for it. We are also curious about the total time spent by all processes when executing this program, so we call `MPI_Reduce` a second time, but with the `MPI_SUM` operator, to add up the time spent by all processes.

Listing 5.9: compute_shortest_paths()

```
void compute_shortest_paths (
          int id,             /* rank of calling process                    */
          int p,              /* number of processes in communicator group  */
          Element_type **a,   /* portion of matrix owned by calling process */
          int n,              /* number of rows and columns (must be square)*/
          MPI_Comm comm)      /* communicator handle                        */

{
    int  i, j, k;             /* for genereal use */
    int  local_index;         /* local index of broadcast row */
    int  root;                /* process controlling row to be bcast */
    Element_type* tmp;        /* holds the broadcast row        */
    int  nlocal_rows;         /* number of rows owned by process */


    /* Allocate a linear array large enough to hold one row of the matrix */
    tmp = (Element_type *) malloc (n * sizeof(Element_type));

```

```
19      for (k = 0; k < n; k++) {
20          /* Determine which process owns the kth row. */
21          root = owner(k,p,n);
22          /* If the executing process is the owner of the row, it has to send
23             a copy to all other processes */
24          if ( root == id ) {
25              /* Compute row k's index in the submatrix owned by process id
26                 It is k - first row number, which is floor(id*n/p)       */
27              local_index = k - (id*n)/p;
28
29              /* Copy the row from the submatrix owned by this process into
30                 the temporary array to be broadcast. We cannot broadcast the
31                 actual row because we will be updating it. */
32              for (j = 0; j < n; j++)
33                  tmp[j] =  a[local_index][j];
34          }
35          /* Broadcast tmp from root to all other processes */
36          MPI_Bcast (tmp, n, MPI_TYPE, root, comm);
37          nlocal_rows = number_of_rows(id,n,p);
38          for (i = 0; i < nlocal_rows; i++)
39              for (j = 0; j < n; j++)
40                  a[i][j] = MIN(a[i][j],a[i][k]+tmp[j]);
41      }
42      free (tmp);
43  }
```

Listing 5.10: mpi_floyd.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <mpi.h>
 4 #include "utilities.h"
 5 #include "matrix_io.h"
 6
 7 typedef int Element_type;
 8 #define MPI_TYPE MPI_INT
 9
10 void compute_shortest_paths (
11           int id,             /* rank of calling process                    */
12           int p,              /* number of processes in communicator group  */
13           Element_type **a,   /* portion of matrix owned by calling process */
14           int n,              /* number of rows and columns (must be square)*/
15           MPI_Comm comm);     /* communicator handle                        */
16
17
18 int main (int argc, char *argv[])
19 {
20      Element_type** adjmatrix;      /* Doubly-subscripted array */
21      Element_type*  matrix_storage; /* Local portion of array elements */
22      int      id;                   /* Process rank */
23      int      nrows;                /* Rows in matrix */
24      int      ncols;                /* Columns in matrix */
25      int      p;                    /* Number of processes */
26      double   time;
27      double   max_time;
28      double   total_time=0;
29      int      error;
30      char     errstring[127];
31
```

```
32      MPI_Init (&argc, &argv);
33      MPI_Comm_rank (MPI_COMM_WORLD, &id);
34      MPI_Comm_size (MPI_COMM_WORLD, &p);
35
36      if ( argc < 2 ) {
37          sprintf(errstring, "Usage: %s filename, where filename contains"
38                             " binary adjacency matrix\n", argv[0]);
39          terminate(id, errstring);
40      }
41
42      /* Read the matrix from the file named on the command line and
43         distribute the rows to each process. Check error on return */
44      read_and_distribute_matrix (argv[1],
45                          (void *) &adjmatrix,
46                          (void *) &matrix_storage,
47                          MPI_TYPE, &nrows, &ncols, &error,
48                          MPI_COMM_WORLD);
49
50      /* Check if successful */
51      if ( SUCCESS != error ) {
52          terminate(id, "Error reading or allocating matrix.\n");
53      }
54
55      /* Check if the matrix is square and exit if not */
56      if (nrows != ncols) {
57          terminate(id, "Error: matrix is not square.\n");
58      }
59
60      /* Gather the submatrices from all processes onto process 0 and print
61         out the entire matrix */
62      collect_and_print_matrix ((void **) adjmatrix, MPI_TYPE, nrows, ncols,
            MPI_COMM_WORLD);
63
64      /* Time how long the longest process takes to do its work.
65         Start them at the gate together using barrier synchronization. */
66      MPI_Barrier ( MPI_COMM_WORLD);
67
68      /* Get the baseline time */
69      time = -MPI_Wtime();
70
71      /* Compute the shortest paths for the part of the matrix owned by the calling
            process */
72      compute_shortest_paths (id, p, (Element_type **) adjmatrix, ncols,
            MPI_COMM_WORLD);
73
74      /* Get the time (each process gets its own local time) */
75      time += MPI_Wtime();
76
77      /* Use a reduction with the MAX operator to put the largest time
78         on process 0 */
79      MPI_Reduce ( &time, &max_time, 1, MPI_DOUBLE, MPI_MAX, 0,
80                   MPI_COMM_WORLD);
81
82      /* Use a reduction using the SUM operator to get the total time
83         for comparing to the total serial time. */
84      MPI_Reduce ( &time, &total_time, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
85
86      if  (0 == id )
87          printf ("Floyd, matrix size %d, %d processes. "
```

```
88                  "Elapsed time %6.4f seconds,"
89                  " Total time %6.4f  seconds\n",
90                   ncols, p, max_time,total_time);
91
92      /* Gather the submatrices from all processes onto process 0 and print
93         out the entire matrix */
94      collect_and_print_matrix ((void **) adjmatrix, MPI_TYPE, nrows, ncols,
            MPI_COMM_WORLD);
95
96      MPI_Finalize();
97      return 0;
98  }
99
100 /* Code for compute_shortest_paths() is here */
```

5.9 Analysis and Benchmarking

Whenever we develop a parallel algorithm, we will analyze its time complexity, and sometimes its space complexity. The final step in this chapter is to see how well this algorithm performs.

Because both the sequential program and the parallel program must read and print the matrix, we do not include the time spent doing input and output of the matrix in our analysis. The sequential program does all of the I/O in a single process. In the parallel version, one process performs the I/O and uses message-passing to distribute or collect from the other processes. The overhead of the message passing does increase the I/O time, but it is small in comparison to the time spent in the `compute_shortest_paths` function. If you look at the code in `read_and_distribute_matrix`, which is symmetric to that of `collect_and_print_matrix`, you see that it sends a message whose average size is $\Theta(\lceil n^2/p \rceil)$ to each of $p-1$ processes, because the matrix has n^2 elements. Therefore, the overhead that is added is proportional to n^2. In general, it is not customary to include the overhead of distributing input data to the separate processors in the performance analysis.

We therefore turn to the time complexity of the `compute_shortest_paths` function. Before the loop begins, there is a call to allocate a linear array of size n. This is a constant time operation that we can ignore (because the implementation of `malloc` does not iteratively allocate small chunks at a time.) What about the complexity of the outermost loop? For each iteration of the outermost loop, there is a call to get the owner of the current row, which is constant time ($\Theta(1)$), a loop that copies the row from the matrix into a buffer, which has time complexity $\Theta(n)$, the broadcast, and the cost of the inner pair of nested loops. Let us look at those nested loops first. In the code

```
for (i = 0; i < nlocal_rows; i++)
    for (j = 0; j < n; j++)
        a[i][j] = MIN(a[i][j],a[i][k]+tmp[j]);
```

the assignment is constant time; the inner loop is $\Theta(n)$, and the outer loop iterates at most $\lceil n/p \rceil$times. Therefore this code has complexity $\Theta(n^2/p)$.

What about the broadcast operation? The process that owns the k^{th} row broadcasts it to all other processes. We know from previous chapters that the time to broadcast a message has both latency and transfer time. In this case we can assume the transfer time dominates the cost. It is proportional to the message length, so its complexity is $\Theta(n)$. The message is broadcast to $p-1$ processes. As we will always assume that it is possible to broadcast using the binomial tree method described in Chapter 3, the time to broadcast a message to all other processes is $\lceil \log(p-1) \rceil$ and the total complexity for the broadcast is $\Theta(n \log p)$.

Therefore each iteration of the outermost loop has time complexity

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

It is executed n times, so the total complexity is

$$\Theta(n^2 \log p + n^3/p) \tag{5.6}$$

Eq. 5.6 is an asymptotic time complexity, but not a predictor of actual running time. We can derive an expression for the running time of the actual program with a more detailed analysis and then see how well that estimate compares to the running time on an actual parallel computer. Quinn [2] chooses a commodity cluster as the hardware. Our analysis will use a multiprocessor with 6 cores instead.

The program has a computational component and a communication component. Let β be the bandwidth of the communication links and let λ be the latency. Assume that the distances stored in the adjacency matrix are four-byte quantities, either integers or floats. Then each broadcast has a time requirement of $\lambda + 4n/\beta$. It takes $\lceil \log p \rceil$ steps to distribute the broadcast row to each process, and there are n iterations of the outer loop and hence n broadcasts, so the total communication time is $n \lceil \log p \rceil (\lambda + 4n/\beta)$.

Let χ be the average time needed to perform a single update to a matrix element (the assignment in the innermost loop.) Then from the above analysis of `compute_shortest_paths`, the time to do one row of updates is $n\chi$ and time to do all of the rows that a process owns is $\lceil n/p \rceil n\chi$ and the total computation time, because this iterates n times, would be

$$n^2 \lceil n/p \rceil \chi \tag{5.7}$$

Adding this to the communication time gives

$$n \lceil \log p \rceil (\lambda + 4n/\beta) + n^2 \lceil n/p \rceil \chi \tag{5.8}$$

This is actually an overestimate of the actual time that the program will take, because it assumes that the communication time does not overlap computation time. Generally, this is not the case. When a broadcast takes place, the process that sends the message usually returns from the call before the message has been delivered, and the transfer time overlaps part of the computation time.

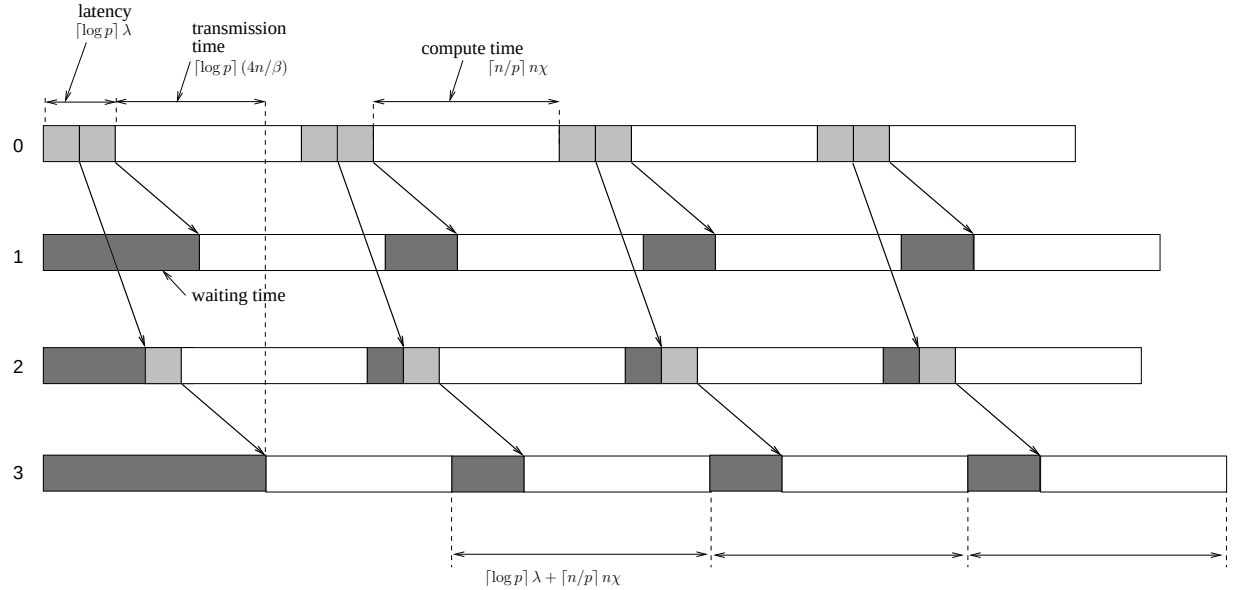


Figure 5.5: Overlap of computation and communication in the first four iterations of the parallel version of Floyd's algorithm, with process 0 responsible for the first 4 rows of the matrix. The figure shows how the transfer of a message overlaps the computation of the sending process.

To make this more concrete, suppose that there are 4 processes and that the problem is sufficiently large that process 0 owns the first 4 rows of the matrix and is therefore the sender for the first 4 iterations of the outer loop in `compute_shortest_paths`. Figure 5.5 depicts this. The horizontal axis is time. Process 0

repeatedly broadcasts a row and then does its own update of all of the rows that it owns. Remember that the way the broadcast takes place, process 0 first sends the message to process 2, and then each of processes 0 and 2 send their copy of the message to the process whose rank is 1 greater, so 0 sends to 1 and 2 sends to 3. The time spent by the process setting up to send the message, its latency, is shown in light gray and is fixed for all processes. If the interconnection network is symmetric the transfer time will be the same in all cases, but it may not be. The figure depicts the transfer time as being approximately the same. The dark rectangles represent time during which the process cannot do anything because it is waiting to receive the next row.

Process 0 begins by sending its row 0 to process 2 and then to process 1, after which it performs its update. The first transfer overlaps the setup time of the second, and the second transfer overlaps the compute time. Process 1 does nothing but compute, but it must wait each time for the data to arrive. Its initial delay is longer, but once it gets the first row, the delay is reduced, because it is computing for the same amount of time as process 0 and it has become synchronized with it. A similar statement is true of process 3 and its delay with respect to process 2.

Because the computation time is longer than the time to transfer the messages, once this initial delay in the first iteration has passed, the pattern of delay is regular, as you can see from the figure, and the amount of time in each iteration is the same.

Let us do a bit of math. A white region of computation in the figure is the time for the process to perform the nested loops after the broadcast, which we determined was $\lceil n/p \rceil \, n\chi$. The time to broadcast a single row to all processes is the time from the moment process 0 has finished setting up (after the second gray rectangle) until process 3 receives the row that was set up, which is $\lceil \log p \rceil \, (4n/\beta)$. If $\lceil \log p \rceil \, (4n/\beta) < \lceil n/p \rceil \, n\chi$, then the transmission time is strictly smaller than the compute time and after the first iteration, it does not contribute to the total time. Therefore a more accurate expression for the running time is

$$n \lceil \log p \rceil \, \lambda + n^2 \lceil n/p \rceil \, \chi + \lceil \log p \rceil \, 4n/\beta \tag{5.9}$$

because the transmission time is added to the total only once.

We ran the program on a six-core multiprocessor using inputs of size $n = 1000, 2000, 3000$. The bandwidth on such as machine is very high, so we would expect that the transmission time would be very negligible. Running the program on three input sizes with six different numbers of processors generated eighteen linear equations of the form $a_{i,1}\lambda + a_{i,2}\chi + a_{i,3}(1/\beta) = t_i, \; 1 \leq i \leq 18$. We used a linear system solver to estimate the values of λ, χ, and β and then plotted the predicted curves using Eq. 5.9 and the actual data in Figure 5.6. The values of the three parameters were estimated to be $\lambda = 4.39$ nsec, $\chi = 25.5$ μsec, and $\beta = 47.6$ KB/sec, which is probably significantly incorrect. The figure shows that Eq. 5.9is a fairly good predictor nonetheless.

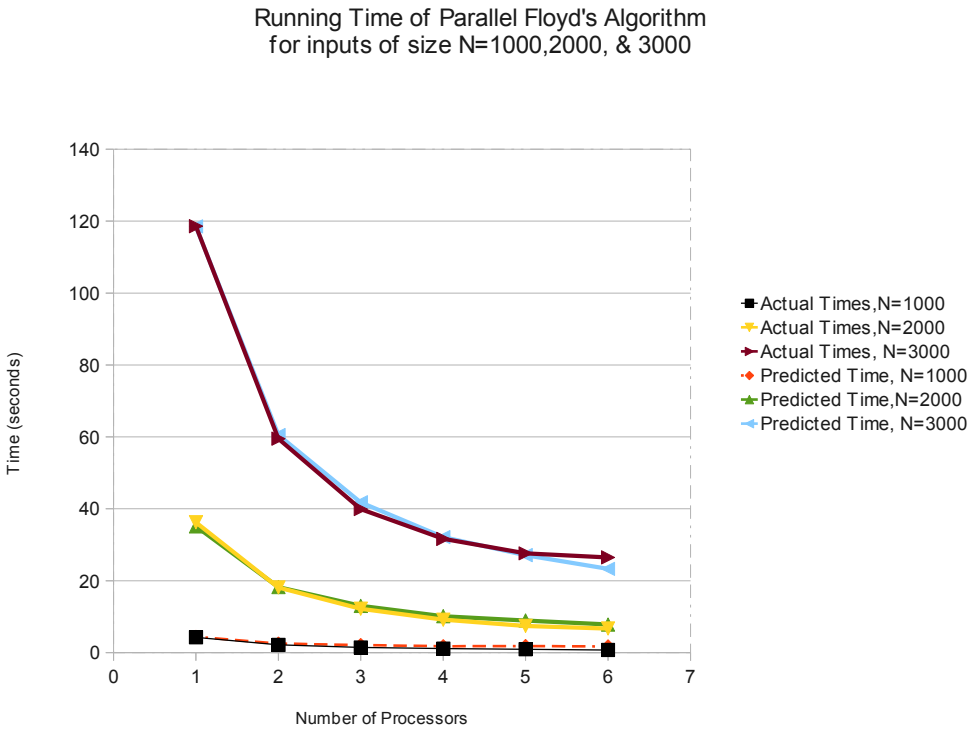


Figure 5.6: Graph of actual and predicted run times of parallel version of Floyd's algorithm on a 6-core multiprocessor.

References

[1] Marlene Caroselli. *Leadership Skills for Managers.* McGraw-Hill, Inc., New York, NY, USA, 2000.

[2] M.J. Quinn. *Parallel Programming in C with MPI and OpenMP.* McGraw-Hill Higher Education. McGraw-Hill Higher Education, 2004.

Subject Index