CHAPTER 7

Communicators and Topologies

THE USE OF COMMUNICATORS AND TOPOLOGIES makes MPI different from most other message-passing systems. Recollect that, loosely speaking, a communicator is a collection of processes that can send messages to each other. A topology is a structure imposed on the processes in a communicator that allows the processes to be addressed in different ways. In order to illustrate these ideas, we will develop code to implement Fox's algorithm for multiplying two square matrices.

7.1 Matrix Multiplication

Recall that if $A = (a_{ij})$ and $B = (b_{ij})$ are square matrices of order n, then $C = (c_{ij}) = AB$ is also a square matrix of order n, and c_{ij} is obtained by taking the dot product of the ith row of A with the jth column of B. That is,

$$c_{ij} = a_{i0}b_{0j} + a_{i1}b_{1j} + \cdots + a_{i,n-1}b_{n-1,j}.$$

See Figure 7.1. Here's a simple algorithm for matrix multiplication:

```
for each row of C
    for each column of C {
        C[row][column] = 0.0;
        for each element of this row of A
            Add A[row][element]*B[element][column]
                to C[row][column]
    }
```

This can be readily implemented in C as follows:

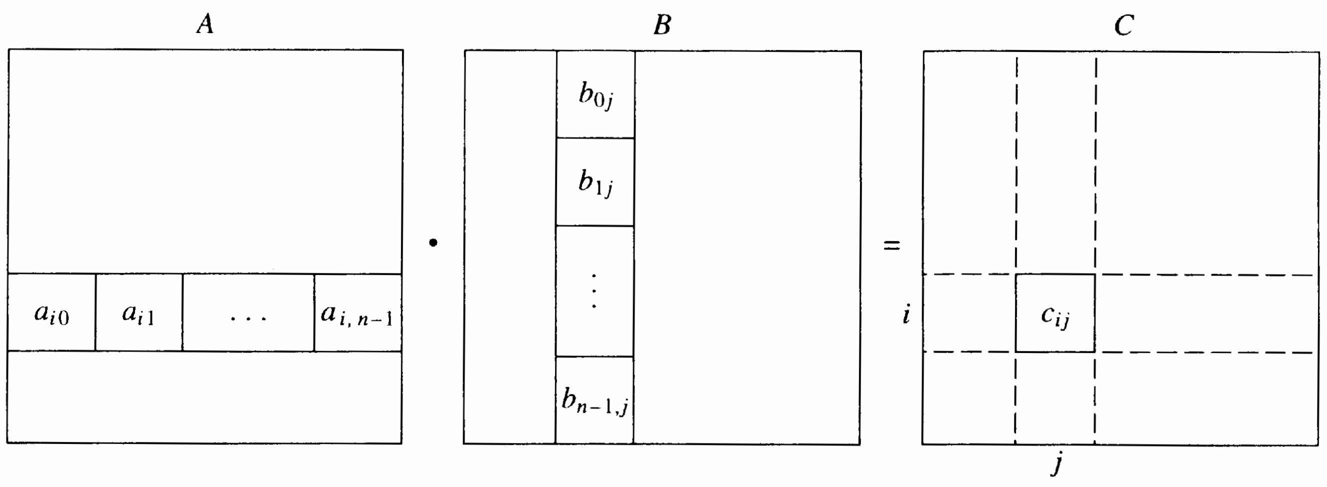


Figure 7.1 Matrix multiplication

```
/* MATRIX_T is a two-dimensional array of floats */
void Serial_matrix_mult(
        MATRIX_T   A   /* in  */,
        MATRIX_T   B   /* in  */,
        MATRIX_T   C   /* out */,
        int        n   /* in  */) {
    int i, j, k;

    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++) {
            C[i][j] = 0.0;
            for (k = 0; k < n; k++)
                C[i][j] = C[i][j] + A[i][k]*B[k][j];
        }
}  /* Serial_matrix_mult */
```

Observe that a straightforward parallel implementation will be quite costly. For example, suppose (for the sake of simplicity) that the number of processes is the same as the order of the matrices; i.e., $p = n$, and suppose that we have distributed the matrices by rows. So process 0 is assigned row 0 of A, B, and C; process 1 is assigned row 1 of A, B, and C; etc. Then in order to form the dot product of the ith row of A with the jth column of B, we will need to gather the jth column of B onto process i. But we will need to form the dot product of the jth column with *every* row of A. So we'll have to carry out an allgather rather than a gather, and we'll have to do this for *every* column of B. If we assume that there is insufficient storage for each process to store all of B, our parallel matrix-matrix multiply might be something like this:

```
for each column of B {
    Allgather(column);
    Compute dot product of my row of A with
        column;
}
```

Process 0		Process 1	
a_{00}	a_{01}	a_{02}	a_{03}
a_{10}	a_{11}	a_{12}	a_{13}
Process 2		Process 3	
a_{20}	a_{21}	a_{22}	a_{23}
a_{30}	a_{31}	a_{32}	a_{33}

Figure 7.2 Checkerboard mapping of a 4×4 matrix to four processes

Even with an efficient implementation of allgather, this will involve quite a lot of (expensive) communication. Similar reasoning shows that an algorithm that distributes the matrices by columns will also involve large amounts of communication. In view of these considerations, most parallel matrix multiplication functions use a **checkerboard** distribution of the matrices. This means that the processes are viewed as a grid, and, rather than assigning entire rows or entire columns to each process, we assign small submatrices. For example, if we have four processes, we might assign the elements of a 4×4 matrix as shown in Figure 7.2. In the next section we'll take a look at one algorithm that uses checkerboard mappings of the matrices.

7.2 Fox's Algorithm

In order to simplify the discussion, let's assume (for the time being) that the matrices have order n, and the number of processes, p, equals n^2. Then a checkerboard mapping assigns a_{ij}, b_{ij}, and c_{ij} to process $i * n + j$, or, loosely, process (i, j). Fox's algorithm for matrix multiplication proceeds in n stages: one stage for each term $a_{ik}b_{kj}$ in the dot product

$$c_{ij} = a_{i0}b_{0j} + a_{i1}b_{1j} + \cdots + a_{i,n-1}b_{n-1,j}.$$

During the initial stage, each process multiplies the diagonal entry of A in its process row by its element of B:

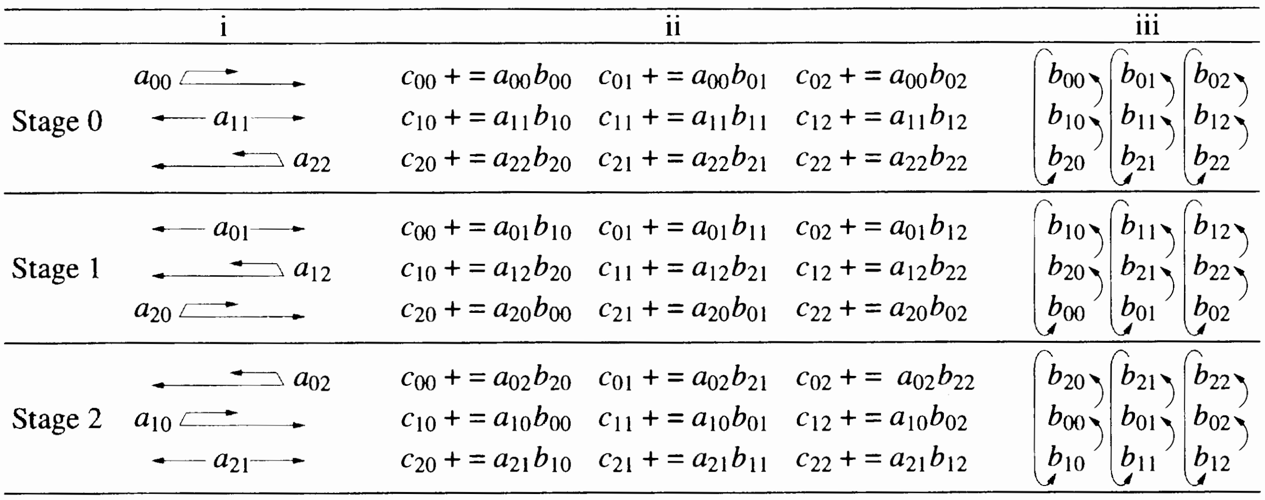
Stage 0 on process (i, j): $c_{ij} = a_{ii}b_{ij}$.

During the next stage, each process multiplies the element immediately to the right of the diagonal of A (in its process row) by the element of B directly beneath its own element of B:

Stage 1 on process (i, j): $c_{ij} = c_{ij} + a_{i,i+1}b_{i+1,j}$.

In general, during the kth stage, each process multiplies the element k columns to the right of the diagonal of A by the element k rows below its own element of B:

Stage k on process (i, j): $c_{ij} = c_{ij} + a_{i,i+k}b_{i+k,j}$.



	i	ii			iii		
	a_{00}	$c_{00} += a_{00}b_{00}$	$c_{01} += a_{00}b_{01}$	$c_{02} += a_{00}b_{02}$	b_{00}	b_{01}	b_{02}
Stage 0	a_{11}	$c_{10} += a_{11}b_{10}$	$c_{11} += a_{11}b_{11}$	$c_{12} += a_{11}b_{12}$	b_{10}	b_{11}	b_{12}
	a_{22}	$c_{20} += a_{22}b_{20}$	$c_{21} += a_{22}b_{21}$	$c_{22} += a_{22}b_{22}$	b_{20}	b_{21}	b_{22}
	a_{01}	$c_{00} += a_{01}b_{10}$	$c_{01} += a_{01}b_{11}$	$c_{02} += a_{01}b_{12}$	b_{10}	b_{11}	b_{12}
Stage 1	a_{12}	$c_{10} += a_{12}b_{20}$	$c_{11} += a_{12}b_{21}$	$c_{12} += a_{12}b_{22}$	b_{20}	b_{21}	b_{22}
	a_{20}	$c_{20} += a_{20}b_{00}$	$c_{21} += a_{20}b_{01}$	$c_{22} += a_{20}b_{02}$	b_{00}	b_{01}	b_{02}
	a_{02}	$c_{00} += a_{02}b_{20}$	$c_{01} += a_{02}b_{21}$	$c_{02} += a_{02}b_{22}$	b_{20}	b_{21}	b_{22}
Stage 2	a_{10}	$c_{10} += a_{10}b_{00}$	$c_{11} += a_{10}b_{01}$	$c_{12} += a_{10}b_{02}$	b_{00}	b_{01}	b_{02}
	a_{21}	$c_{20} += a_{21}b_{10}$	$c_{21} += a_{21}b_{11}$	$c_{22} += a_{21}b_{12}$	b_{10}	b_{11}	b_{12}

Figure 7.3 Fox's algorithm

Of course, we can't just add k to a row or column subscript and expect to always get a valid row or column number. For example, if $i = j = n - 1$, then any positive value added to i or j will result in an out-of-range subscript. One possible solution is to use subscripts modulo n. That is, rather than use $i + k$ for a row or column subscript, use $i + k \bmod n$. Then, we will get a valid pair of subscripts:

$$\text{Stage } k \text{ on process } (i,j)\text{: } \bar{k} = (i + k) \bmod n;\ c_{ij} = c_{ij} + a_{i,\bar{k}}b_{\bar{k},j}.$$

Also observe that we'll compute c_{ij} as follows:

$$c_{ij} = a_{ii}b_{ij} + a_{i,i+1}b_{i+1,j} + \cdots + a_{i,n-1}b_{n-1,j} + a_{i0}b_{0j} + \cdots \\ + a_{i,i-1}b_{i-1,j}.$$

In other words, if we compute the subscripts modulo n, the algorithm is correct.

Perhaps we should say that the *incomplete* algorithm is correct. We still haven't said how we arrange that each process gets the appropriate values $a_{i,\bar{k}}$ and $b_{\bar{k},i}$. Since the algorithm computes the correct element-wise products, we know that process (i,j) will get the correct element of A from its *process* row and the correct element of B from its *process* column. Also observe that during the initial stage each process in the ith row uses a_{ii}. In general, during the kth stage, each process in the ith row uses $a_{i\bar{k}}$, where $\bar{k} = i + k \bmod n$. Thus, we need to broadcast $a_{i\bar{k}}$ across the ith row before each multiplication. Finally observe that during the initial stage, each process uses its own element, b_{ij}, of B. During subsequent stages, process (i,j) will use $b_{i\bar{k}}$. Thus, *after* each multiplication is completed, the elements of B should be "shifted" up one row, and elements in the top row should be sent to the bottom row. Figure 7.3 illustrates the stages in Fox's algorithm for multiplying two 3×3 matrices distributed across nine processes.

It's not obvious that Fox's algorithm is superior to the basic parallel matrix multiplication we discussed in the preceding section. So we'll return to this problem when we discuss parallel program performance in Chapter 11. It is obvious, however, that we're unlikely to have access to n^2 processors even for relatively small (e.g., 100×100) matrices. So how can we modify our algorithm so that it uses fewer than n^2 processes?

A natural solution would seem to be to store submatrices rather than matrix elements on each process, and try carrying out the algorithm we have just outlined using submatrices. It turns out that this idea works, provided the submatrices can be multiplied together as needed. One way we can insure that this is the case is to use a square grid of processes, where the number of process rows or process columns, $\sqrt{p}$, evenly divides n. With this assumption, each process is assigned a square $n/\sqrt{p} \times n/\sqrt{p}$ submatrix of each of the three matrices. Specifically, let $\bar{n} = n/\sqrt{p}$ and define A_{ij} to be the $\bar{n} \times \bar{n}$ submatrix of A whose first entry is $a_{i*\bar{n},j*\bar{n}}$. For example, if $n = p = 4$, then $\bar{n} = 4/\sqrt{4} = 2$, and

$$A_{00} = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix}, \quad A_{01} = \begin{pmatrix} a_{02} & a_{03} \\ a_{12} & a_{13} \end{pmatrix},$$
$$A_{10} = \begin{pmatrix} a_{20} & a_{21} \\ a_{30} & a_{31} \end{pmatrix}, \quad A_{11} = \begin{pmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{pmatrix}.$$

If we make similar definitions of B_{ij} and C_{ij}, assign A_{ij}, B_{ij}, and C_{ij} to process (i, j), and we define $q = \sqrt{p}$, then our algorithm will compute

$$\begin{aligned} C_{ij} = {} & A_{ii}B_{ij} + A_{i,i+1}B_{i+1,j} + \cdots + A_{i,q-1}B_{q-1,j} + A_{i0}B_{0j} + \cdots \\ & + A_{i,i-1}B_{i-1,j}. \end{aligned}$$

If we multiply out each submatrix product, we can verify that this does in fact compute the correct values for each c_{ij}.

To summarize then, if we denote the submatrices A_{ij}, B_{ij}, and C_{ij} by A[i,j], B[i,j], and C[i,j], respectively, we can outline Fox's algorithm as follows:

```
    /* my process row = i, my process column = j */
    q = sqrt(p);
    dest = ((i-1) mod q, j);
    source = ((i+1) mod q, j);
    for (stage = 0; stage < q; stage++) {
        k_bar = (i + stage) mod q;
(a)     Broadcast A[i,k_bar] across process row i;
(b)     C[i,j] = C[i,j] + A[i,k_bar]*B[k_bar,j];
(c)     Send B[k_bar,j] to dest;  Receive
        B[(k_bar+1) mod q ,j] from source;
    }
```

7.3 Communicators

If we start working on coding Fox's algorithm, it becomes apparent that implementation will be greatly facilitated if we can treat certain subsets of processes as a "communication universe"—at least on a temporary basis. For example, in statement (a),

```
(a)  Broadcast A[i,k_bar] across process row i
```

it would be useful to treat each row of processes as a communication universe, while in statement (c),

```
(c)  Send B[k_bar,j] to dest;  Receive
     B[(k_bar+1) mod q ,j] from source;
```

it would be useful to treat each column of processes as a communication universe.

The mechanism that MPI provides for treating a subset of processes as a communication universe is the *communicator*. Up to now, we've been loosely defining a communicator as a collection of processes that can send messages to each other. However, now that we want to construct our own communicators, we will need a more careful discussion.

In MPI, there are two types of communicators: **intra-communicators** and **inter-communicators**. Intra-communicators are essentially a collection of processes that can send messages to each other *and* engage in collective communication operations. Inter-communicators, as the name implies, are used for sending messages between processes belonging to *disjoint* intra-communicators. We'll focus on intra-communicators now and briefly touch on inter-communicators in programming assignment 2.

A minimal (intra-)communicator is composed of

- a group
- a context

A **group** is an ordered collection of processes. If a group consists of p processes, each process in the group is assigned a unique **rank**, which is just a nonnegative integer in the range $0, 1, \ldots, p-1$. A **context** is a system-defined object that uniquely identifies a communicator. Two distinct communicators will have different contexts, even if they have identical underlying groups. A context can be thought of as a system-defined tag that is associated with a group in a communicator. Contexts are used to insure that messages are received correctly. Recall that no message can be received by any process unless the communicator used by the sending process is identical to the communicator used by the receiving process: this is true for both point-to-point (e.g., `MPI_Send/Recv`) and collective communications. Since distinct communicators use distinct contexts, the system can check whether two communicators are identical by simply checking whether the contexts are identical.

In order to understand contexts better, let's speculate for a moment about how a system developer might implement communicators. She might define a group to be an array, `group`, and the rank of process i in the group would correspond to rank `group[i]` in `MPI_COMM_WORLD`.

For example, suppose we've coded Fox's algorithm, and we're running it with nine processes; i.e., `MPI_COMM_WORLD` consists of nine processes. As we've already observed, it is convenient for us to view our nine processes as a 3×3 grid. So we might create a communicator for each row of the grid. In particular, the group for the "second row communicator" might be composed of processes 3, 4, and 5 from `MPI_COMM_WORLD`. Thus

```
group[0] = 3;
group[1] = 4;
group[2] = 5;
```

and process 0 in the second row communicator would be the same as process 3 in `MPI_COMM_WORLD`, process 1 the same as process 4, and process 2 the same as process 5.

She might also define a context to be an int. Each process could keep a list of available contexts. When a new communicator is created, the processes participating in the creation could "negotiate" the choice of a context that is available to each process. Then, in communication functions, rather than sending the entire communicator every time a message is sent, just the context can be sent.

Keep in mind that these constructions of communicators, groups, and contexts are purely hypothetical. The implementation of each object is system dependent, and it's entirely possible that your system uses something very different.

This pairing of a group with a context is the most basic form of a communicator. Other data can be associated with a communicator. In particular, a structure or topology can be imposed on the processes in a communicator, allowing a more natural addressing scheme. We'll discuss topologies in section 7.6.

7.4 Working with Groups, Contexts, and Communicators

To illustrate the basics of working with communicators, let's create a communicator whose underlying group consists of the processes in the first row of our virtual grid. Suppose that `MPI_COMM_WORLD` consists of p processes, where $q^2 = p$. Let's also suppose that our first row of processes consists of the processes with ranks $0, 1, \ldots, q-1$. (Here, the ranks are in `MPI_COMM_WORLD`.) In order to create the group of our new communicator, we can execute the following code:

```
    MPI_Group  group_world;
    MPI_Group  first_row_group;
    MPI_Comm   first_row_comm;
    int*       process_ranks;

    /* Make a list of the processes in the new
     * communicator */
    process_ranks = (int*) malloc(q*sizeof(int));
    for (proc = 0; proc < q; proc++)
        process_ranks[proc] = proc;

    /* Get the group underlying MPI_COMM_WORLD */
    MPI_Comm_group(MPI_COMM_WORLD, &group_world);

    /* Create the new group */
    MPI_Group_incl(group_world, q, process_ranks,
        &first_row_group);

    /* Create the new communicator */
    MPI_Comm_create(MPI_COMM_WORLD, first_row_group,
        &first_row_comm);
```

This code proceeds in a fairly straightforward fashion to build the new communicator. First it creates a list of the processes to be assigned to the new communicator. Then it creates a group consisting of these processes. This requires two commands: first get the group associated with MPI_COMM_WORLD, since this is the group from which the processes in the new group will be taken; then create the group with MPI_Group_incl. Finally, the actual communicator is created with a call to MPI_Comm_create. The call to MPI_Comm_create associates a context with the new group. The result is the communicator first_row_comm. Now the processes in first_row_comm can perform collective communication operations. For example, process 0 (in first_row_group) can broadcast A_{00} to the other processes in first_row_group.

```
    int my_rank_in_first_row;
    float* A_00;

    /* my_rank is process rank in group_world */
    if (my_rank < q) {
        MPI_Comm_rank(first_row_comm,
            &my_rank_in_first_row);
        /* Allocate space for A_00 */
        A_00 = (float*) malloc (n_bar*n_bar*sizeof(float));
        if (my_rank_in_first_row == 0) {
            /* Initialize A_00 */
              .
              .
              .
        }
        MPI_Bcast(A_00, n_bar*n_bar, MPI_FLOAT, 0,
            first_row_comm);
    }
```

Groups and communicators are **opaque objects**. From a practical standpoint, this means that the details of their internal representation depend on the particular implementation, and, as a consequence, they cannot be directly accessed by the user. Rather, the user accesses a **handle** that references the opaque object, and the opaque objects are manipulated by special MPI functions, for example, MPI_Comm_create, MPI_Group_incl, and MPI_Comm_group.

Contexts are not explicitly used in any MPI functions. Rather they are implicitly associated with groups when communicators are created.

The syntax of the commands we used to create first_row_comm is fairly self-explanatory. The first command

```
int MPI_Comm_group(
        MPI_Comm     comm    /* in  */,
        MPI_Group*   group   /* out */)
```

simply returns the group underlying the communicator comm.

The second command

```
int MPI_Group_incl(
        MPI_Group     old_group             /* in  */,
        int           new_group_size        /* in  */,
        int           ranks_in_old_group[]  /* in  */,
        MPI_Group*    new_group             /* out */)
```

creates a new group from a list of processes in the existing group, old_group. The number of processes in the new group is new_group_size, and the processes to be included are listed in ranks_in_old_group. Process 0 in new_group has rank ranks_in_old_group[0] in old_group, process 1 in new_group has rank ranks_in_old_group[1] in old_group, etc.

The final command

```
int MPI_Comm_create(
        MPI_Comm    old_comm     /* in  */,
        MPI_Group   new_group    /* in  */,
        MPI_Comm*   new_comm     /* out */)
```

associates a context with the group new_group and creates the communicator new_comm. All of the processes in new_group belong to the group underlying old_comm.

There is an extremely important distinction between the first two functions and the third. MPI_Comm_group and MPI_Group_incl are both **local** operations. That is, there is no communication among processes involved in their execution. However, MPI_Comm_create is a collective operation. *All* the processes in old_comm—including those not joining new_comm—must call MPI_Comm_create with the same arguments. The main reason for this was noted in the preceding section: it provides a means for the processes to choose a single context for the new communicator. Note that since MPI_Comm_create is collective, it will behave, in terms of the data transmitted, as if it synchronizes. In particular, if several communicators are being created, they must be created in the same order on all the processes.

7.5 MPI_Comm_split

In our matrix multiplication program we need to create multiple communicators—one for each row of processes and one for each column. This would be an extremely tedious process if the number of processes, p, were large, and we had to create each communicator using the three functions discussed in the previous section. Fortunately, MPI provides a function, MPI_Comm_split, that can create several communicators simultaneously. As an example of its use, we'll create one communicator for each row of processes.

```
MPI_Comm  my_row_comm;
int       my_row;

/* my_rank is rank in MPI_COMM_WORLD.
 * q*q = p */
my_row = my_rank/q;
MPI_Comm_split(MPI_COMM_WORLD, my_row, my_rank,
    &my_row_comm);
```

The single call to MPI_Comm_split creates q new communicators, all of them having the same name, my_row_comm. For example, if $p = 9$, the group underlying my_row_comm will consist of the processes $\{0,1,2\}$ on processes 0, 1, and 2. On processes 3, 4, and 5, the group underlying my_row_comm will consist of the processes $\{3,4,5\}$, and on processes 6, 7, and 8, it will consist of processes $\{6,7,8\}$.

The syntax of MPI_Comm_split is

```
int MPI_Comm_split(
        MPI_Comm    old_comm   /* in  */,
        int         split_key  /* in  */,
        int         rank_key   /* in  */,
        MPI_Comm*   new_comm   /* out */)
```

It creates a new communicator for each value of split_key. Processes with the same value of split_key form a new group. The rank in the new group is determined by the value of rank_key. If process A and process B call MPI_Comm_split with the same value of split_key, and the rank_key argument passed by process A is less than that passed by process B, then the rank of A in the group underlying new_comm will be less than the rank of process B. If they call the function with the same value of rank_key, the system will arbitrarily assign one of the processes a lower rank.

MPI_Comm_split is a collective call, and it must be called by all the processes in old_comm. The function can be used even if the user doesn't wish to assign every process to a new communicator. This can be accomplished by passing the predefined constant MPI_UNDEFINED as the split_key argument. Processes doing this will have the predefined value MPI_COMM_NULL returned in new_comm.

7.6 Topologies

Recollect that it is possible to associate additional information—information beyond the group and context—with a communicator. Such information is said to be **cached** with the communicator. One of the most important possibilities for cached information, or **attributes**, is a topology. In MPI, a **topology** is just a mechanism for associating different addressing schemes with the processes belonging to a group. Note that MPI topologies are *virtual* topologies—there may be no simple relation between the process structure implicit in a virtual topology and the actual underlying physical structure of the parallel system.

There are essentially two types of virtual topologies that can be created in MPI—a Cartesian or grid topology and a graph topology. Conceptually, Cartesian topologies form a special case of graph topologies. However, because of the importance of grids in applications, there is a separate collection of functions in MPI whose purpose is the manipulation of virtual grids.

In Fox's algorithm we wish to identify the processes in MPI_COMM_WORLD with the coordinates of a square grid, and each row and each column of the grid needs to form its own communicator. Let's look at one method for building this structure.

We begin by associating a square grid structure with MPI_COMM_WORLD. In order to do this, we need to specify the following information:

1. The number of dimensions in the grid. We have two.
2. The size of each dimension. In our case, this is just the number of rows and the number of columns. We have q rows and q columns.
3. Periodicity of each dimension. In our case, this information specifies whether the first entry in each row or column is "adjacent" to the last entry in that row or column, respectively. Since we want a "circular" shift of the submatrices in each column, we want the second dimension to be periodic. It's unimportant whether the first dimension is periodic.
4. Finally, MPI gives the user the option of allowing the system to optimize the mapping of the grid of processes to the underlying physical processors

by possibly reordering the processes in the group underlying the communicator. Since we don't need to preserve the ordering of the processes in MPI_COMM_WORLD, we should allow the system to reorder.

Having made all these decisions, we simply execute the following code:

```
MPI_Comm  grid_comm;
int       dim_sizes[2];
int       wrap_around[2];
int       reorder = 1;

dim_sizes[0] = dim_sizes[1] = q;
wrap_around[0] = wrap_around[1] = 1;
MPI_Cart_create(MPI_COMM_WORLD, 2, dim_sizes,
    wrap_around, reorder, &grid_comm);
```

After executing this code, the communicator grid_comm will contain all the processes in MPI_COMM_WORLD (possibly reordered), and a two-dimensional Cartesian coordinate system will be associated with it. In order for a process to determine its coordinates, it simply calls the function MPI_Cart_coords:

```
int  coordinates[2];
int  my_grid_rank;

MPI_Comm_rank(grid_comm, &my_grid_rank);
MPI_Cart_coords(grid_comm, my_grid_rank, 2,
    coordinates);
```

Notice that we needed to call MPI_Comm_rank in order to get the process rank in grid_comm. This was necessary because in our call to MPI_Cart_create we set the reorder flag to 1, and hence the original process ranking in MPI_COMM_WORLD may have changed.

The "inverse" to MPI_Cart_coords is MPI_Cart_rank.

```
MPI_Cart_rank(grid_comm, coordinates,
    &grid_rank);
```

Given the coordinates of a process, MPI_Cart_rank returns the rank of the process in its third parameter grid_rank.

The syntax of MPI_Cart_create is

```
int MPI_Cart_create(
        MPI_Comm    old_comm         /* in  */,
        int         number_of_dims   /* in  */,
        int         dim_sizes[]      /* in  */,
        int         wrap_around[]    /* in  */,
        int         reorder          /* in  */,
        MPI_Comm*   cart_comm        /* out */)
```

MPI_Cart_create creates a new communicator, cart_comm, by caching a Cartesian topology with old_comm. Information on the structure of the Cartesian topology is contained in the parameters number_of_dims, dim_sizes, and wrap_around. The first of these, number_of_dims, contains the number of dimensions in the Cartesian coordinate system. The next two, dim_sizes and wrap_around, are arrays with order equal to number_of_dims. The array dim_sizes specifies the order of each dimension, and wrap_around specifies whether each dimension is circular, wrap_around[i] = 1, or linear, wrap_around[i] = 0.

The processes in cart_comm are ranked in *row-major* order. That is, the first row consists of processes 0, 1, ... , dim_sizes[0] − 1; the second row consists of processes dim_sizes[0], dim_sizes[0] + 1, ..., 2*dim_sizes[0] − 1; etc. Thus it may be advantageous to change the relative ranking of the processes in old_comm. For example, suppose the physical topology is a 3×3 grid, and the processes (numbers) in old_comm are assigned to the processors (grid squares) as follows:

3	4	5
0	1	2
6	7	8

Clearly, the performance of Fox's algorithm would be improved if we renumbered the processes. However, since the user doesn't know what the exact mapping of processes to processors is, we must let the system do it by setting the reorder parameter to 1.

Since MPI_Cart_create constructs a new communicator, it is a collective operation.

The syntax of the address information functions is

```
int MPI_Cart_rank(
        MPI_Comm    comm           /* in  */,
        int         coordinates[]  /* in  */,
        int*        rank           /* out */);

int MPI_Cart_coords(
        MPI_Comm   comm             /* in  */,
        int        rank             /* in  */,
        int        number_of_dims   /* in  */,
        int        coordinates[]    /* out */)
```

MPI_Cart_rank returns the rank in the Cartesian communicator comm of the process with Cartesian coordinates coordinates. So coordinates is an array with order equal to the number of dimensions in the Cartesian topology associated with comm. MPI_Cart_coords is the inverse to MPI_Cart_rank: it returns the coordinates of the process with rank rank in the Cartesian communicator comm. Note that both of these functions are local.

7.7 MPI_Cart_sub

We can also partition a grid into grids of lower dimension. For example, we can create a communicator for each row of the grid as follows:

```
int         free_coords[2];
MPI_Comm    row_comm;

free_coords[0] = 0;
free_coords[1] = 1;
MPI_Cart_sub(grid_comm, free_coords, &row_comm);
```

The call to MPI_Cart_sub creates q new communicators. The free_coords parameter is an array of boolean. It specifies whether each dimension "belongs" to the new communicator. Since we're creating communicators for the rows of the grid, each new communicator consists of the processes obtained by fixing the row coordinate and letting the column coordinate vary; i.e., the row coordinate is fixed and the column coordinate is free. Hence we assigned free_coords[0] the value 0—the first coordinate isn't free—and we assigned free_coords[1] the value 1—the second coordinate is free or varies. On each process, the new communicator is returned in row_comm. In order to create the communicators for the columns, we simply reverse the assignments to the entries in free_coords.

```
MPI_Comm col_comm;

free_coords[0] = 1;
free_coords[1] = 0;
MPI_Cart_sub(grid_comm, free_coords, &col_comm);
```

Note the similarity of MPI_Cart_sub to MPI_Comm_split. They perform similar functions—they both partition a communicator into a collection of new communicators. However, MPI_Cart_sub can only be used with a communicator that has an associated Cartesian topology, and the new communicators can only be created by fixing one or more dimensions of the old communicators and letting the other dimensions vary. Also note that MPI_Cart_sub is, like MPI_Comm_split, a collective operation.

The syntax of MPI_Cart_sub is

```
int MPI_Cart_sub(
        MPI_Comm    cart_comm       /* in  */,
        int         free_coords[]   /* in  */,
        MPI_Comm*   new_comm        /* out */)
```

It partitions the processes in cart_comm into a collection of disjoint communicators whose union is cart_comm. Both cart_comm and each new_comm have associated Cartesian topologies. If cart_comm has dimensions

$d_0 \times d_1 \times \cdots \times d_{n-1}$, then the dimension of free_coords is n. If free_coords[i] is 0 (or false), then the ith coordinate is fixed for the construction of the new communicators. If free_coords[j] is 1 (or true), then the jth coordinate is free or allowed to vary. Thus, if free_coords[i] is 0, for $i = i_0, i_1, \ldots, i_{k-1}$, then the call to MPI_Cart_sub will create $d_{i_0} d_{i_1} \cdots d_{i_{k-1}}$ new communicators. Each new communicator will be obtained by letting the remaining dimensions (i.e., those for which free_coords is 1) vary over their ranges.

7.8 Implementation of Fox's Algorithm

To complete our discussion, let's write the code to implement Fox's algorithm. First, we'll write a function that creates the various communicators and associated information. Since this requires a large number of variables, and we'll be using this information in other functions, we'll put it into a struct to facilitate passing it.

```
typedef struct {
    int        p;          /* Total number of processes    */
    MPI_Comm   comm;       /* Communicator for entire grid */
    MPI_Comm   row_comm;   /* Communicator for my row      */
    MPI_Comm   col_comm;   /* Communicator for my col      */
    int        q;          /* Order of grid                */
    int        my_row;     /* My row number                */
    int        my_col;     /* My column number             */
    int        my_rank;    /* My rank in the grid comm     */
} GRID_INFO_T;

/* We assume space for grid has been allocated in the
 * calling routine.
 */
void Setup_grid(
         GRID_INFO_T*  grid  /* out */) {
    int old_rank;
    int dimensions[2];
    int wrap_around[2];
    int coordinates[2];
    int free_coords[2];

    /* Set up Global Grid Information */
    MPI_Comm_size(MPI_COMM_WORLD, &(grid->p));
    MPI_Comm_rank(MPI_COMM_WORLD, &old_rank);

    /* We assume p is a perfect square */
    grid->q = (int) sqrt((double) grid->p);
    dimensions[0] = dimensions[1] = grid->q;
    /* We want a circular shift in second dimension. */
    /* Don't care about first                        */
    wrap_around[0] = wrap_around[1] = 1;
    MPI_Cart_create(MPI_COMM_WORLD, 2, dimensions,
        wrap_around, 1, &(grid->comm));
    MPI_Comm_rank(grid->comm, &(grid->my_rank));
    MPI_Cart_coords(grid->comm, grid->my_rank, 2,
        coordinates);
    grid->my_row = coordinates[0];
    grid->my_col = coordinates[1];

    /* Set up row communicators */
    free_coords[0] = 0;
    free_coords[1] = 1;
    MPI_Cart_sub(grid->comm, free_coords,
        &(grid->row_comm));

    /* Set up column communicators */
    free_coords[0] = 1;
    free_coords[1] = 0;
    MPI_Cart_sub(grid->comm, free_coords,
        &(grid->col_comm));
} /* Setup_grid */
```

Notice that since each of our communicators has an associated topology, we constructed them using the topology construction functions—MPI_Cart_create and MPI_Cart_sub—rather than the more general communicator construction functions MPI_Comm_create and MPI_Comm_split.

Now let's write the function that does the actual multiplication. We'll assume that the user has supplied the type definitions and functions for the local matrices. Specifically, we'll assume she has supplied a type definition for LOCAL_MATRIX_T, a corresponding derived type, local_matrix_mpi_t, and three functions: Local_matrix_multiply, Local_matrix_allocate, and Set_to_zero. We also assume that storage for the parameters has been allocated in the calling function, and all the parameters, except the product matrix local_C, have been initialized.

```
void Fox(
        int              n         /* in  */,
        GRID_INFO_T*     grid      /* in  */,
        LOCAL_MATRIX_T*  local_A   /* in  */,
        LOCAL_MATRIX_T*  local_B   /* in  */,
        LOCAL_MATRIX_T*  local_C   /* out */) {

    LOCAL_MATRIX_T*  temp_A; /* Storage for the sub-    */
                             /* matrix of A used during */
                             /* the current stage       */
    int             stage;
    int             bcast_root;
    int             n_bar;  /* n/sqrt(p)                */
    int             source;
    int             dest;
    MPI_Status      status;

    n_bar = n/grid->q;
    Set_to_zero(local_C);

    /* Calculate addresses for circular shift of B */
    source = (grid->my_row + 1) % grid->q;
    dest = (grid->my_row + grid->q - 1) % grid->q;

    /* Set aside storage for the broadcast block of A */
    temp_A = Local_matrix_allocate(n_bar);

    for (stage = 0; stage < grid->q; stage++) {
        bcast_root = (grid->my_row + stage) % grid->q;
        if (bcast_root == grid->my_col) {
            MPI_Bcast(local_A, 1, local_matrix_mpi_t,
                bcast_root, grid->row_comm);
            Local_matrix_multiply(local_A, local_B,
                local_C);
        } else {
            MPI_Bcast(temp_A, 1, local_matrix_mpi_t,
                bcast_root, grid->row_comm);
            Local_matrix_multiply(temp_A, local_B,
                local_C);
        }
        MPI_Sendrecv_replace(local_B, 1, local_matrix_mpi_t,
            dest, 0, source, 0, grid->col_comm, &status);
    } /* for */

} /* Fox */
```

In the last function call, we have used a new MPI function, MPI_Sendrecv_replace. It performs both the send and the receive required for the circular shift of local_B: it sends the current copy of local_B to the process in col_comm with rank dest, and then receives the copy of local_B residing on the process in col_comm with rank source. Its syntax is

```
    int MPI_Sendrecv_replace(
            void*           buffer    /* in/out */,
            int             count     /* in     */,
            MPI_Datatype    datatype  /* in     */,
            int             dest      /* in     */,
            int             send_tag  /* in     */,
            int             source    /* in     */,
        int           recv_tag  /* in     */,
        MPI_Comm      comm      /* in     */,
        MPI_Status*   status    /* out    */)
```

It sends the contents of buffer to the process in comm with rank dest and receives in buffer data sent from the process with rank source. The send uses the tag send_tag, and the receive uses the tag recv_tag. The processes involved in the send and the receive don't have to be distinct. The process dest can receive the contents of buffer with a call to MPI_Recv, and the process source can send with a call to MPI_Send. The function is called MPI_Sendrecv_replace to distinguish it from the function MPI_Sendrecv, which also performs a send and a receive, but it uses different buffers for the send and the receive.

7.9 Summary

We covered a lot of ground in this chapter. We studied two algorithms for parallel matrix multiplication, and we learned about two new ideas in MPI: communicators and topologies.

We started the chapter with a discussion of matrix multiplication and a simple algorithm for parallel matrix multiplication. The parallel algorithm mapped rows of the matrices to the processes, and we saw that this mapping would require a large amount of communication. So we explored alternative mappings. The alternative mapping we used in our second matrix multiplication function is called a *checkerboard* mapping. It mapped square submatrices to the processes rather than rows or columns.

Our initial development of Fox's algorithm made the assumption that each process stored a single element of the matrix rather than an entire submatrix. Although this is an unrealistic assumption, it is a common design technique in parallel programming, since it reduces the complexity of the initial design by reducing the number of parameters we need to work with. Care must be taken in moving from the simplified design to the final, general design: it's easy to make mistakes in computing the extra parameters. In our case, the extra parameters were the order of the submatrices and the order of the process grid.

Fox's algorithm views the processes as a virtual grid, and in order to carry out several operations in Fox's algorithm, it is convenient to view certain subgrids as communication universes. MPI provides mechanisms for the construction and manipulation of both virtual grids and communication universes. Communication universes correspond to MPI **communicators**, and virtual grids correspond to a type of MPI process topology.

There are two types of MPI communicator: intra-communicators and inter-communicators. Intra-communicators can be used in MPI_Send/Recv and in collective communication functions. Inter-communicators are used for communication between processes belonging to disjoint communicators. We discussed intra-communicators.

Basic intra-communicators consist of a **group** and a **context**. A group is an ordered collection of processes. A context is a unique, system-defined label that is associated with a group when a communicator is created. It can be viewed as a system-defined tag that is used by the system to check the communicator arguments to communication functions: a message can only be received if the context of the communicator argument used by the receiving process equals the context of the communicator argument used by the sending process. The key distinction between contexts and tags is that contexts are system defined, and hence guaranteed to be unique, even across subprograms written by different programmers.

Groups and communicators are **opaque objects**. From a practical standpoint, this means that the details of their internal representation depend on the particular implementation, and, as a consequence, they cannot be directly accessed by the user. Rather, the user accesses a **handle** that references the opaque object, and the opaque objects are manipulated by special MPI functions, for example, `MPI_Comm_create`, `MPI_Group_incl`, and `MPI_Comm_group`. Contexts are not accessed at all by MPI functions: they are implicitly defined by the system when a communicator is created.

We discussed several methods for building user-defined communicators. Our most basic approach consisted in building a group, and then having the system associate a context with the group. This proceeded in three stages. First we get the group underlying a communicator that contains the processes we want in our new communicator with

```
int MPI_Comm_group(
        MPI_Comm    comm         /* in  */,
        MPI_Group*  old_group    /* out */)
```

Then we create an array, `ranks_in_old_group`, listing the process ranks in the old group of the processes we want from our old group. That is, `ranks_in_old_group[i]` is the rank in `old_group` of the *i*th process in `new_group`. In order to create the new group, we call

```
int MPI_Group_incl(
        MPI_Group    old_group             /* in  */,
        int          new_group_size        /* in  */,
        int          ranks_in_old_group[]  /* in  */,
        MPI_Group*   new_group             /* out */)
```

Once we have our new group, we can associate a context with it by calling

```
int MPI_Comm_create(
        MPI_Comm   old_comm    /* in  */,
        MPI_Group  new_group   /* in  */,
        MPI_Comm*  new_comm    /* out */)
```

There is an important distinction between the first two functions and the third: the first two are completely local functions—they involve no communication—while the third function, MPI_Comm_create, is a collective communication function: it involves all the processes in old_comm.

Once we have built our communicator, we can use it as an argument to MPI_Send/ Recv or a collective communication function just as we've been using MPI_COMM_WORLD.

If we want to simply split the communicator into a collection of disjoint subcommunicators, we can use a single call to

```
int MPI_Comm_split(
        MPI_Comm    old_comm    /* in  */,
        int         split_key   /* in  */,
        int         rank_key    /* in  */,
        MPI_Comm*   new_comm    /* out */)
```

It creates a new communicator for each value of split_key. If two processes have the same value of split_key, they will be assigned to the same communicator. If two processes are assigned to the same new communicator by MPI_Comm_split, their relative ranks in the new communicator are determined by the value of rank_key: the process with the smaller value of rank_key will be assigned a lower rank in the communicator. This is a collective operation.

Process topologies allow us to address processes in ways that are more natural to our application. MPI provides two types of process topologies: graphs and grids. We discussed grids. In a grid topology we identify the processes with vertices in a regular rectangular grid of any dimension. For example, in a two-dimensional grid, we can associate a row and column with each process.

MPI allows user programs to associate information, or **attributes**, with communicators by a process called **caching**. Process topologies are one of the most important examples of cached attributes. In order to create a communicator with a cached grid topology, we can call

```
int MPI_Cart_create(
        MPI_Comm    old_comm          /* in  */,
        int         number_of_dims    /* in  */,
        int         dim_sizes[]       /* in  */,
        int         wrap_around[]     /* in  */,
        int         reorder           /* in  */,
        MPI_Comm*   cart_comm         /* out */)
```

This will create a new communicator, cart_comm, and is a collective operation. In addition to the group and context, cart_comm has cached information that associates a number_of_dims-dimensional coordinate system with the processes in cart_comm. In order to access this coordinate system, we can use the functions

```
int MPI_Cart_coords(
        MPI_Comm  cart_comm         /* in  */,
        int       rank              /* in  */,
        int       number_of_dims    /* in  */,
        int       coordinates[]     /* out */)

int MPI_Cart_rank(
        MPI_Comm   cart_comm        /* in  */,
        int        coordinates[]    /* in  */,
        int*       rank             /* out */);
```

The first function takes the rank of a process in cart_comm and returns its coordinates in the grid. The second returns a process's rank given its coordinates.

Since we frequently wish to partition a grid into subgrids (e.g., rows or columns in a two-dimensional grid), MPI provides a function analogous to MPI_Comm_split that can be used for creating subgrids:

```
int MPI_Cart_sub(
        MPI_Comm   cart_comm        /* in  */,
        int        free_coords[]    /* in  */,
        MPI_Comm*  new_comm         /* out */)
```

The array free_coords has order equal to the dimension of cart_comm. The new communicators new_comm are determined by free_coords. If free_coords[i] is 0, the *i*th coordinate is fixed; if it's 1, the *i*th coordinate is allowed to vary. For example, in a two-dimensional grid, if free_coords[0] is 0 and free_coords[1] is 1, the zeroth coordinate is fixed and the first coordinate varies. So MPI_Cart_sub will create a new communicator for each row of cart_comm. MPI_Cart_sub is a collective operation.

Our last MPI function in this chapter was the point-to-point communication function

```
int MPI_Sendrecv_replace(
        void*          buffer    /* in/out */,
        int            count     /* in     */,
        MPI_Datatype   datatype  /* in     */,
        int            dest      /* in     */,
        int            send_tag  /* in     */,
        int            source    /* in     */,
        int            recv_tag  /* in     */,
        MPI_Comm       comm      /* in     */,
        MPI_Status*    status    /* out    */)
```

It performs both a send and a receive, and buffer is used both for the outgoing and incoming messages. It is very convenient to use this function if we have to carry out an operation such as a circular shift of data across a group of processes.

7.10 References

Fox's algorithm is discussed in both [18] and [26]. [26] also discusses several other approaches to parallel matrix multiplication.

Communicators and topologies are discussed in detail in both the MPI Standard [28, 29] and [34]. [21] has several examples of the use of both intra- and inter-communicators. See [28, 29] for a discussion of graph topologies.

7.11 Exercises

1. Suppose that `MPI_COMM_WORLD` consists of $p = mn$ processes. Even if we don't associate a topology with this communicator, we can view it as a virtual grid with m rows and n columns by considering the first row to consist of processes $\{0, 1, \ldots, n-1\}$, the second row to consist of processes $\{n, n+1, \ldots, 2n-1\}$, etc.
   a. Use `MPI_Comm_group`, `MPI_Group_incl`, and `MPI_Comm_create` to create a communicator consisting of the processes belonging to the first column of the virtual grid.
   b. Use `MPI_Comm_split` to create n communicators. Each communicator should consist of the processes belonging to a column of the virtual grid.
   c. If you wrote a program that contained the code in both parts (a) and (b), would the two communicators containing the processes belonging to the first column of the grid be identical?
2. We suggested that we might implement communicators as follows. Groups would be arrays whose entries are the ranks of the processes in `MPI_COMM_WORLD`. Contexts would be integers, and each process would keep a list of available contexts. An actual implementation will probably store considerably more information in a communicator, but we can use this to get a feel for some of the issues involved in the use of communicators.
   a. Using the basic implementation as your starting point, suggest an implementation of `MPI_Comm_create`.
   b. Suggest an implementation of `MPI_Comm_split`.
3. When we discussed Fox's algorithm, we observed that it would be unrealistic to expect the system to provide n^2 physical processors. So we modified our original algorithm so that each process stored submatrices of order $n/\sqrt{p}$. Our basic algorithm, in which we store a single row on each process, is also unrealistic. Outline a modification to the basic algorithm so that it stores a block of n/p rows on each process and, at each stage, gathers n/p columns of B onto each process. Compare the storage requirements of the modified Fox algorithm and the modified basic algorithm.

4. A program is using a three-dimensional process topology. If the dimension sizes are l, m, and n, respectively, use `MPI_Cart_sub` to create the following sets of Cartesian communicators:

   a. m two-dimensional communicators, each consisting of ln processes
   b. lm one-dimensional communicators, each consisting of n processes
   c. lmn zero-dimensional communicators, each consisting of one process

   In addition to `MPI_COMM_WORLD`, MPI provides one predefined communicator for each process: `MPI_COMM_SELF`. Is the communicator you defined in part (c) on process 0 the same as the communicator `MPI_COMM_SELF` on process 0?

5. One reason that it is convenient to use `MPI_Sendrecv_replace` is that it takes care of buffering for us. Can you devise a "safe" implementation of our circular shift that only uses `MPI_Send` and `MPI_Recv`? Recall that a program is safe if it will run correctly even if the system provides no buffering. (Hint: split the processes into two sets; one set sends first, the other set receives first.)

7.12 Programming Assignments

1. Write the additional functions necessary to completely implement a program that uses Fox's algorithm to multiply two square matrices. Have each process generate its local submatrices (rather than reading them in). For output, have each process send its result submatrix to process 0, and have process 0 print out the submatrix. Don't try to print out a "unified" matrix.

2. Recollect that inter-communicators can be used for communication between processes belonging to disjoint intra-communicators. This ability is especially useful in a **client-server** type of program: one or more processes (the server processes) have resources that other processes (the client processes) need access to. A simple example is provided by automatic teller machines (the clients) and a bank's central database system (the server). Probably their most important use will be in *future* versions of MPI that allow for the dynamic creation of processes: inter-communicators will provide a means for newly created processes to communicate with already existing processes.

   Suppose that we wish to send messages from processes in `comm_1` to processes in `comm_2`, and `comm_1` and `comm_2` are disjoint: no process belongs to both of them. In order to do this using inter-communicators, we first need to identify a process in `comm_1` and a process in `comm_2` that both belong to a "parent" communicator `comm_0`. Then we can build the inter-communicator using `MPI_Intercomm_create`:

```
if (I belong to comm_1) {
    local_leader = rank in comm_1 of process
        belonging to both comm_0 and comm_1;
    remote_leader =  rank in comm_0 of
        local_leader of comm_2;
    MPI_Intercomm_create(comm_1, local_leader,
        comm_0, remote_leader, 0, &inter_comm);
} else /* I belong to comm_2 */ {
    local_leader = rank in comm_2 of process
        belonging to both comm_0 and comm_2;
    remote_leader =  rank in comm_0 of
        local_leader of comm_1;
    MPI_Intercomm_create(comm_2, local_leader,
        comm_0, remote_leader, 0, &inter_comm);
}
```

Now processes belonging to comm_1 can send messages to processes belonging to comm_2 using *point-to-point* communication functions. For example, process 0 (rank in comm_1) can send a message to process 0 (rank in comm_2) as follows:

```
char message[100];

if (my_rank in comm_1 == 0) {
    sprintf(message,"Greetings from comm_1!");
    MPI_Send(message, strlen(message)+1, MPI_CHAR,
        0, 0, inter_comm);
else if (my_rank in comm_2 == 0) {
    MPI_Recv(message, 100, MPI_CHAR, 0, 0,
        inter_comm, &status);
}
```

The source and destination ranks are the ranks in the *remote* communicator. Inter-communicators cannot be used for collective communication.

The syntax of MPI_Intercomm_create is

```
int MPI_Intercomm_create(
        MPI_Comm    local_comm      /* in  */,
        int         local_leader    /* in  */,
        MPI_Comm    parent_comm     /* in  */,
        int         remote_leader   /* in  */,
        int         tag             /* in  */,
        MPI_Comm*   inter_comm      /* out */)
```

Note that the type of inter-communicators is the same as the type of intra-communicators. Also note that all the processes in the first communicator should use the same arguments, and all the processes in the second communicator should use the same arguments. The function call is collective across the union of the two communicators.

Write a short program that splits the processes in MPI_COMM_WORLD into two communicators: the processes with even ranks and the processes with odd ranks. Create an inter-communicator from these two communicators and have each process in the odd-ranked communicator send a message to a process in the even-ranked communicator. Be sure you can handle the case where there's an odd number of processes in MPI_COMM_WORLD.