5 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 [1] for multiplying two square matrices.

5.1 Fox's Algorithm

We assume that the factor matrices $A = (a_{ij})$ and $B = (b_{ij})$ have order n. We also assume that the number of processes, p, is a perfect square, whose square root evenly divides n. Say $p = q^2$, and $\bar{n} = n/q$. In Fox's algorithm the factor matrices are partitioned among the processes in a *block checkerboard* fashion. So we view our processes as a virtual two-dimensional $q \times q$ grid, and each process is assigned an $\bar{n} \times \bar{n}$ submatrix of each of the factor matrices. More formally, we have a mapping

$$\phi : \{0, 1, \ldots, p-1\} \longrightarrow \{(s,t) : 0 \leq s, t \leq q-1\}$$

that is both one-to-one and onto. This defines our grid of processes: process i belongs to the row and column given by $\phi(i)$. Further, the process with rank $\phi^{-1}(s,t)$ is assigned the submatrices

$$A_{st} = \begin{pmatrix} a_{s*\bar{n},t*\bar{n}} & \cdots & a_{(s+1)*\bar{n}-1,t*\bar{n}} \\ \vdots & & \vdots \\ a_{s*\bar{n},(t+1)*\bar{n}-1} & \cdots & a_{(s+1)*\bar{n}-1,(t+1)*\bar{n}-1} \end{pmatrix},$$

and

$$B_{st} = \begin{pmatrix} b_{s*\bar{n},t*\bar{n}} & \cdots & b_{(s+1)*\bar{n}-1,t*\bar{n}} \\ \vdots & & \vdots \\ b_{s*\bar{n},(t+1)*\bar{n}-1} & \cdots & b_{(s+1)*\bar{n}-1,(t+1)*\bar{n}-1} \end{pmatrix}.$$

For example, if $p = 9, \phi(x) = (x/3, x \bmod 3)$, and $n = 6$, then A would be partitioned as follows.

Process 0 $A_{00} = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix}$	Process 1 $A_{01} = \begin{pmatrix} a_{02} & a_{03} \\ a_{12} & a_{13} \end{pmatrix}$	Process 2 $A_{02} = \begin{pmatrix} a_{04} & a_{05} \\ a_{14} & a_{15} \end{pmatrix}$
Process 3 $A_{10} = \begin{pmatrix} a_{20} & a_{21} \\ a_{30} & a_{31} \end{pmatrix}$	Process 4 $A_{11} = \begin{pmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{pmatrix}$	Process 5 $A_{12} = \begin{pmatrix} a_{24} & a_{25} \\ a_{34} & a_{35} \end{pmatrix}$
Process 6 $A_{20} = \begin{pmatrix} a_{40} & a_{41} \\ a_{50} & a_{51} \end{pmatrix}$	Process 7 $A_{21} = \begin{pmatrix} a_{42} & a_{43} \\ a_{52} & a_{53} \end{pmatrix}$	Process 8 $A_{22} = \begin{pmatrix} a_{44} & a_{45} \\ a_{54} & a_{55} \end{pmatrix}$

In Fox's algorithm, the block submatrices, A_{rs} and B_{st}, $s = 0, 1, \ldots, q-1$, are multiplied and accumulated on process $\phi^{-1}(r, t)$. The basic algorithm is:

```
for(step = 0; step < q; step++) {
    1.  Choose a submatrix of A from each row of processes.
    2.  In each row of processes broadcast the submatrix
        chosen in that row to the other processes in
        that row.
    3.  On each process, multiply the newly received
        submatrix of A by the submatrix of B currently
        residing on the process.
    4.  On each process, send the submatrix of B to the
        process directly above.  (On processes in the
        first row, send the submatrix to the last row.)
}
```

The submatrix chosen in the rth row is $A_{r,u}$, where

$$u = (r + \mathsf{step}) \bmod q.$$

5.2 Communicators

If we try to implement Fox's algorithm, it becomes apparent that our work 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 the pseudo-code

```
2.  In each row of processes broadcast the submatrix
    chosen in that row to the other processes in
    that row,
```

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

```
4.  On each process, send the submatrix of B to the
    process directly above.  (On processes in the
    first row, send the submatrix to the last row.)
```

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. For example, MPI_COMM_WORLD is an intra-communicator, and we would like for each row and each column of processes in Fox's algorithm to form an intra-communicator. Inter-communicators, as the name implies, are used for sending messages between processes belonging to *disjoint* intra-communicators. For example, an inter-communicator would be useful in an environment that allowed one to dynamically create processes: a newly created set of processes that formed an intra-communicator could be linked to the original set of processes (e.g., MPI_COMM_WORLD) by an inter-communicator. We will only discuss intra-communicators. The interested reader is referred to [4] for details on the use of inter-communicators.

A minimal (intra-)communicator is composed of

- a *Group*, and
- 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 can be thought of as a system-defined tag that is attached to a group. So two processes that belong to the same group and that use the same context can communicate. This pairing of a group with a context is the most basic form of a communicator. Other data can be associated to 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 5.5.

5.3 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 $\phi(x) = (x/q, x \bmod q)$. So the first row of processes consists of the processes with ranks 0, 1, ..., $q-1$. (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 MPI_GROUP_WORLD;
MPI_Group first_row_group;
MPI_Comm first_row_comm;
int row_size;
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, &MPI_GROUP_WORLD);

/* Create the new group */
MPI_Group_incl(MPI_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 required 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 implicitly 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 MPI_GROUP_WORLD */
if (my_rank < q) {
    MPI_Comm_rank(first_row_comm,
        &my_rank_in_first_row);
    /* Allocate space for A_00, order = n_bar */
    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 of MPI, 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, MPI_Group* group)
```

simply returns the group underlying the communicator comm.

The second command

```
int MPI_Group_incl(MPI_Group old_group, int new_group_size,
    int* ranks_in_old_group, MPI_Group* new_group)
```

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, MPI_Group new_group,
    MPI_Comm* new_comm)
```

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 must call MPI_Comm_create with the same arguments. The *Standard* [4] gives three reasons for this:

1. It allows the implementation to layer MPI_Comm_create on top of regular collective communications.

2. It provides additional safety.

3. It permits implementations to avoid communication related to context creation.

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.

5.4 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 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, and 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, and 5, and on processes 6, 7, and 8 it will consist of processes 6, 7, and 8.

The syntax of MPI_Comm_split is

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

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.

5.5 Topologies

Recollect that it is possible to associate additional information — information beyond the group and context — with a communicator. This additional information is said to be *cached* with the communicator, and one of the most important pieces of information that can be cached with a communicator 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 defined by a virtual topology, and the actual underlying physical structure of the parallel machine.

There are essentially two types of virtual topologies that can be created in MPI — a *cartesian* or *grid* topology and a *graph* topology. Conceptually, the former is subsumed by the latter. 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 2.

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. The 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 dimensions[2];
int wrap_around[2];
int reorder = 1;

dimensions[0] = dimensions[1] = q;
wrap_around[0] = wrap_around[1] = 1;
MPI_Cart_create(MPI_COMM_WORLD, 2, dimensions,
    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 it will have a two-dimensional cartesian coordinate system associated. 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 been changed in grid_comm.

The "inverse" to MPI_Cart_coords is MPI_Cart_rank.

```
int 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 process_rank.

The syntax of MPI_Cart_create is

```
int MPI_Cart_create(MPI_Comm old_comm,
    int number_of_dims, int* dim_sizes, int* periods,
    int reorder, MPI_Comm* cart_comm)
```

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 periods. The first of these, number_of_dims, contains the number of dimensions in the cartesian coordinate system. The next two, dim_sizes and periods, are arrays with order equal to number_of_dims. The array dim_sizes specifies the order of each dimension, and periods specifies whether each dimension is circular or linear.

The processes in cart_comm are ranked in *row-major* order. That is, the first row consists of processes $0, 1, \ldots,$ dim_sizes[0]-1, the second row consists of processes dim_sizes[0], dim_sizes[0] $+ 1, \ldots,$ 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, int* coordinates,
    int* rank);
int MPI_Cart_coords(MPI_Comm comm, int rank,
    int number_of_dims, int* coordinates)
```

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.

5.6 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 varying_coords[2];
MPI_Comm row_comm;

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

The call to MPI_Cart_sub creates q new communicators. The varying_coords argument 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*. Hence we assigned varying_coords[0] the value 0 — the first coordinate doesn't vary — and we assigned varying_coords[1] the value 1 — the second coordinate

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 varying_coords.

```
MPI_Comm col_comm;

varying_coords[0] = 1; varying_coords[1] = 0;
MPI_Cart_sub(grid_comm, varying_coord, 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 (or varying) one or more dimensions of the old communicators. Also note that MPI_Cart_sub is, like MPI_Comm_split, a collective operation.

5.7 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 among the various functions.

```
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 communicator */
} GRID_INFO_TYPE;

/* We assume space for grid has been allocated in the
 * calling routine.
```

```
 */
void Setup_grid(GRID_INFO_TYPE* grid) {
    int old_rank;
    int dimensions[2];
    int periods[2];
    int coordinates[2];
    int varying_coords[2];

    /* Set up Global Grid Information */
    MPI_Comm_size(MPI_COMM_WORLD, &(grid->p));
    MPI_Comm_rank(MPI_COMM_WORLD, &old_rank);
    grid->q = (int) sqrt((double) grid->p);
    dimensions[0] = dimensions[1] = grid->q;
    periods[0] = periods[1] = 1;
    MPI_Cart_create(MPI_COMM_WORLD, 2, dimensions, periods,
        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 and column communicators */
    varying_coords[0] = 0; varying_coords[1] = 1;
    MPI_Cart_sub(grid->comm, varying_coords,
        &(grid->row_comm));
    varying_coords[0] = 1; varying_coords[1] = 0;
    MPI_Cart_sub(grid->comm, varying_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 lo-

cal matrices. Specifically, we'll assume she has supplied a type definition for LOCAL_MATRIX_TYPE, a corresponding derived type, DERIVED_LOCAL_-MATRIX, 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, GRID_INFO_TYPE* grid,
    LOCAL_MATRIX_TYPE* local_A,
    LOCAL_MATRIX_TYPE* local_B,
    LOCAL_MATRIX_TYPE* local_C) {
    LOCAL_MATRIX_TYPE* temp_A;
    int step;
    int bcast_root;
    int n_bar;  /* order of block submatrix = n/q */
    int source;
    int dest;
    int tag = 43;
    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 (step = 0; step < grid->q; step++) {
        bcast_root = (grid->my_row + step) % grid->q;
        if (bcast_root == grid->my_col) {
            MPI_Bcast(local_A, 1, DERIVED_LOCAL_MATRIX,
                bcast_root, grid->row_comm);
            Local_matrix_multiply(local_A, local_B,
                local_C);
```

```
        } else {
            MPI_Bcast(temp_A, 1, DERIVED_LOCAL_MATRIX,
                bcast_root, grid->row_comm);
            Local_matrix_multiply(temp_A, local_B,
                local_C);
        }
        MPI_Send(local_B, 1, DERIVED_LOCAL_MATRIX, dest, tag,
            grid->col_comm);
        MPI_Recv(local_B, 1, DERIVED_LOCAL_MATRIX, source, tag,
            grid->col_comm, &status);
    } /* for */

} /* Fox */
```