

MPI Parallel Programming Part III

P.Y. Wang
 Department of Computer Science 4A5
 George Mason University
 Fairfax VA 22030-4444 U.S.A.

Revisiting Fox's Algorithm

a_00	a_01	a_02
a_10	a_11	a_12
a_20	a_21	a_22

b_00	b_01	b_02
b_10	b_11	b_12
b_20	b_21	b_22

Initial Arrangements of A and B matrices

a_00		
	a_11	
		a_22

b_00	b_01	b_02
b_10	b_11	b_12
b_20	b_21	b_22

	a_01	
a_20		a_12

b_10	b_11	b_12
b_20	b_21	b_22
b_00	b_01	b_02

		a_02
a_10		
	a_21	

b_20	b_21	b_22
b_00	b_01	b_02
b_10	b_11	b_12

P₀	P₁	P₂
a_00 a_01	a_02 a_03	a_04 a_05
a_10 a_11	a_12 a_13	a_14 a_15
P₃	P₄	P₅
a_20 a_21	a_22 a_23	a_24 a_25
a_30 a_31	a_32 a_33	a_34 a_35
P₆	P₇	P₈
a_40 a_41	a_42 a_43	a_44 a_45
a_50 a_51	a_52 a_53	a_54 a_55

Pseudo-Code

for step = 0; step < \sqrt{p} ; step++

1. Choose a sub-matrix of A from each row of processes: for the r^{th} row, choose sub-matrix $(r + \text{step}) \bmod \sqrt{p}$.
2. In each row of processes, broadcast the sub-matrix chosen in that row to the other processes in that row.
3. On each process, locally multiply the newly received sub-matrix of A by the sub-matrix of B currently residing on the process
4. On each process, send the sub-matrix of B to its north process (circularly).

- To facilitate Fox's algorithm (and others like it), it would be useful to express communications for
 - broadcasting across a row
 - shifting a column north
- Thus, it would be helpful to have different *communication universes* or *communicators* which we construct ourselves

Communicators

In MPI, there are two types of communicators

- intra-communicators:
 - a collection of processes that can send message to each other and engage in collective communication operations
 - `MPI_COMM_WORLD` is an intra-communicator
- inter-communicators:
 - a facility for sending messages between processes in different intra-communicators

Intra-Communicators

A minimal intra-communicator is composed of

- a *group*:
 - an ordered collection of p processes (ranked $0, 1, \dots, p - 1$)
- a *context*:
 - a system defined tag attached to a group
 - this is a hidden synchronization variable
 - if two processes agree on source rank, dest rank, message tag, but use different communicators, they will not synchronize.

⇒ Only processes that belong to the same group and have the same context can communicate!

Creating Communicators

- Groups and communicators can be created in several ways
 - A group can be derived from an existing communicator
 - Communicators can be split into subgroups
 - A structure or topology can be imposed on the processes in a communicator, allowing a more natural addressing scheme
- To create a communicator with specific members, use

```
MPI_Comm_create ( MPI_Comm comm,
                  MPI_Group new_group,
                  MPI_Comm *new_comm )
```

Creating Groups

All group creation routines create a group by specifying the members to take from an *existing* group. For example,

- `MPI_Group_incl` specifies specific members
- `MPI_Group_excl` excludes specific members
- `MPI_Group_range_incl` and `MPI_Group_Range_excl` use ranges of members
- `MPI_Group_union` and `MPI_Group_intersection` creates a new group from two existing groups

- To get the underlying (existing) group, use
`MPI_Comm_group (MPI_comm oldcomm, MPI_Group *group)`
- To free an existing group, use
`MPI_Group_free (MPI_Group *group)`

Start with `MPI_COMM_WORLD` consisting of $p = q \times q$ processes:

```
MPI_Group MPI_GROUP_WORLD;
MPI_Group row0_group;
MPI_Comm row0_comm;
int row_size;
int process_rank[10];          /* assumes q is = 10 */

/* Create list of processes to be assigned to new communicator
   These are ranks in the old (underlying) group */
for (proc = 0; proc < q; proc++) process_rank[proc] = proc;

/* Get the underlying group */
MPI_comm_group (MPI_COMM_WORLD, &MPI_GROUP_WORLD);

/* Create the group consisting of the list of processes */
MPI_comm_group_incl (MPI_COMM_WORLD, q, process_rank, &row0_group);

/* Create the new communicator- all processes in the underlying
   communicator must participate in this creation */
MPI_comm_create (MPI_COMM_WORLD, row0_group, &row0_comm);
```

This creates the new communicator `row0_comm` which can perform collective communication operations.

Using the New Communicator

In this example, process 0 broadcasts its A sub-matrix to other process in the `row0_comm` communicator.

```
int me_in_row0;
float A[20,20]; /* Sub-matrices A[i,j] have size 20 x 20 */

if (me < q) { /* my rank in MPI_COMM_WORLD */

    MPI_Comm_rank (row0_comm, &me_in_row0);

    if (me_in_row0 == 0) {
        do whatever needs to be done before broadcasting
    }

    MPI_Bcast (A, 20*20, MPI_FLOAT, 0, row0_comm);
}
```

To use this in Fox's algorithm, we'd have to create a communicator for each row and column, which could be tedious if p is large

Simultaneous Communicator Creation

- Several communicators can be created simultaneously
- To do this, we could use


```
MPI_Comm_split ( MPI_Comm comm,
                  int split_key, int rank_key
                  MPI_Comm *new_comm )
```
- Processes with the same value of `split_key` form a new communicator
 - Suppose A and B call `MPI_Comm_split` and they have the same value of `split_key`.
 - If `rank_key` of A is less than `rank_key` of B, then the rank of A in the new communicator is less than the rank of B in the new communicator (when `split_key` is the same)

A Short Example

To form groups of rows of processes, use

```
MPI_comm my_row_comm;  
int me, my_row;  
my_row = me/q; /* p = q x q processes */  
MPI_Comm_split (MPI_COMM_WORLD, my_row, me, &my_row_comm);
```

This creates q new communicators; all of them have the same name `my_row_comm`:

- On processes $0, 1, \dots, q - 1$, the group underlying `my_row_comm` consists of processes $\{0, 1, \dots, q - 1\}$.
- On processes $q, q + 1, \dots, 2q - 1$ the group underlying `my_row_comm` consists of processes $\{q, q + 1, \dots, 2q - 1\}$, etc.

To form groups of columns of processes, we could write something like

```
MPI_Comm_Split (MPI_COMM_WORLD, column, 0, &newcomm_col);
```

To maintain the order by rank, we would use

```
MPI_Comm_rank (MPI_COMM_WORLD, &me);  
MPI_Comm_Split (MPI_COMM_WORLD, column, me,  
                &newcomm_col);
```

To free an unwanted communicator, thus freeing system resources, we can use

```
MPI_Comm_free ( MPI_Comm *comm );
```

Process Topologies

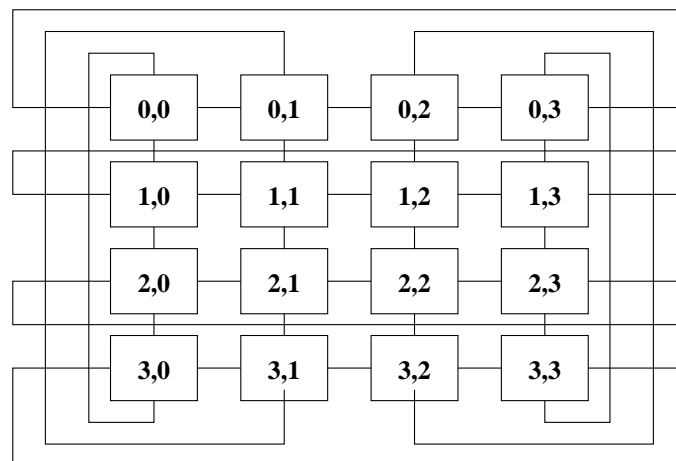
- Information can be added to communicators, giving them structure
- This information is said to be *cached* with the communicator
- A **virtual topology** is information cached with a communicator
- Topologies permit different addressing schemes to be associated with the processes belonging to a group
- In MPI, there are two types of topologies that can be created
 - a **Cartesian** or **grid** topology
 - a **graph** topology

Specifying Cartesian Topologies

To specify a Cartesian or grid topology, one needs to give

- the number of dimensions (e.g. 2)
- the size of each dimension (e.g. 4×4)
- the periodicity of each dimension (e.g. column wrap)
- the option of whether to let the system optimize the mapping of the virtual grid to the hardware by possibly reordering the processes in the underlying group (e.g. we don't care in the case of Fox's algorithm)

2-d Grid Example



```
MPI_Comm grid_com;  
int dims[2];          /* two dimensions */  
int wrap_around[2];  /* wrap-nowrap */  
int reorder = 1;     /* system reordering option */  
  
/* define sizes and periodicity */  
dims[0] = 4; dims[1] = 4;  
wrap_around[0] = 1; wrap_around[1] = 1;  
  
MPI_Cart_create (MPI_COMM_WORLD, 2, dims, wrap_around,  
                reorder, &grid_comm);
```

This creates a communicator `grid_comm` that contains all the (possibly reordered) processes of `MPI_COMM_WORLD` with an associated two-dimensional Cartesian coordinate system

Who am I?

- For a process to locate its coordinates, use the function `MPI_Cart_coords`

```
int coords[2];      /* store my coordinates here */  
int my_grid_rank;  
  
/* Get my rank in the grid (row-major order) */  
MPI_Comm_rank (grid_comm, &my_grid_rank);  
  
/* Get the corresponding Cartesian coordinates */  
MPI_Cart_coords (grid_comm, my_grid_rank, 2, coords);
```

Some Additional Cartesian Functions

- The inverse of the `MPI_Cart_coords` function is `MPI_Cart_rank (grid_comm, coords, &grid_rank)`
- The ranks of neighbors that correspond to common shift operations can be identified by using the function `MPI_Cart_shift`
- The function `MPI_Dims_create` chooses the most balanced (“square”) dimension sizes for a Cartesian coordinate system

Sample Code Skeleton

```
int mycoords[2];
int dims[2];
int periods[2] = {1,0};
int rank_prev, rank_next;
int size;
MPI_Comm comm_cart;
MPI_Comm comm_coll;

/* Create communicator with 2d topology */
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Dims_create(size, 2, dims);
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 1, &comm_cart);

/* Get local coordinates */
MPI_Comm_rank(comm_cart, &rank);
MPI_Comm_coords(comm_cart, rank, 2, mycoords);
```

```

/* Build new communicator on first column */
if (mycoords[1] == 0) {
    MPI_Comm_split(comm_cart, 0, mycoords[0], &comm_coll);
} else {
    MPI_Comm_split(comm_cart, MPI_UNDEFINED, 0, &comm_coll);
}

/* Get the ranks of next row in the same column */
MPI_Cart_shift(comm_cart, 0, 1, &rank_prev, &rank_next);

```

This will be used with subsequent *MPI_Send* and *MPI_Recv* commands in Fox's algorithm

Selecting Partitions of Grids

- To partition a grid into lower dimensions, we can create new communicators for each row (or column) of the grid
- Example:

```

int varying_coords[2]
MPI_Comm row_comm

varying_coords[0] = 0; /* do not use this dimension (fix it) */
varying_coords[1] = 1; /* use this dimension (let it vary) */

MPI_Cart_sub (grid_comm, varying_coords, &row_comm);

```

- This creates q new row communicators.

Fox's Algorithm in MPI

```

typedef struct {
    int      p;                /* Number of processes */
    MPI_Comm comm;            /* Communicator for whole grid */
    MPI_Comm row_comm;        /* Communicator for whole grid */
    MPI_Comm col_comm;        /* Communicator for whole grid */
    int      q;                /* Order of grid: p = q x q */
    int      my_row, my_col;    /* My row and column number */
    int      my_rank;          /* My rank in the grid */
} GRID_INFO_T;

void Setup_grid ( GRID_INFO_T* grid ) {

    int old_rank;
    int dimensions[2];        int wrap_around[2];
    int coordinates[2];       int free_coords[2];

    MPI_Comm_size(MPI_COMM_WORLD, &(amp;grid->p));
    MPI_Comm_rank(MPI_COMM_WORLD, &old_rank);

```

```

    /* Set up grid size as square */
    grid->q = (int) sqrt ((double) grid->p);
    dimensions[0] = dimensions[1] = grid->q;
    /* Set up grid communicator */
    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; /* First dimensions fixed */
    MPI_Cart_sub(grid->comm, free_coords, &(grid->row_comm));

    /* Set up column communicators */
    free_coords[0]= 1; free_coords[1]= 0; /* Second dimensions fixed */
    MPI_Cart_sub(grid->comm, free_coords, &(grid->col_comm));

}

```

```

void Fox ( int          n;
          GRID_INFO_T*  grid;
          LOCAL_MATRIX_T* local_A;
          LOCAL_MATRIX_T* local_B;
          LOCAL_MATRIX_T* local_C; )
{
    LOCAL_MATRIX_T* temp_A; /* Storage for the submatrix A used
                             during the current stage */

    int          stage;
    int          bcast_root;
    int          n_bar; /* n_bar = n/sqrt(p) */
    int          source, dest;
    MPI_Status   status;

    n_bar = n /grid->q;
    Set_to_zero (local_C); /* Call a routine to initialize local_C */

    /* Calculate addresses for the 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);
    }
    /* Circularly shift B north */
    MPI_Sendrecv_replace(local_B, 1, local_matrix_mpi_t,
                          dest,0,source,0,grid->col_comm,&status);
}
}

```

A Few More Words on Communication

- A convenient MPI command exists for circular shifting:
`MPI_Sendrecv_replace` (`void*` buffer,
int count, `MPI_Datatype` datatype,
int dest, int send_tag, int source, int recv_tag,
`MPI_Comm` comm, `MPI_Status*` status);
- MPI non-blocking (Immediate) Send and Receive commands exist:
`MPI_Isend` and `MPI_Irecv`

Non-Blocking Point-to-Point Communication

- `MPI_Isend`(`void*` buffer, int count, `MPI_datatype` datatype,
int dest, int tag,
`MPI_comm` comm, `MPI_Request*` request)
- `MPI_Irecv`(`void*` buffer, int count, `MPI_datatype` datatype,
int source, int tag,
`MPI_comm` comm, `MPI_Request*` request)
- `MPI_Wait`(`MPI_Request*` request, `MPI_Status*` status)
- The *request* parameter is a “handle” for the operation started by the non-blocking call
- It contains information on source or destination, the tag, the communicator, and the buffer

An Allgather Example: Blocking Communication

```

for (i = 0; i < p-1; i++) {
    send_offset = ((my_rank - i + p) % p) * blocksize;
    recv_offset = ((my_rank - i - 1 + p) % p) * blocksize;

    MPI_Send(y + send_offset, blocksize, MPI_FLOAT,
             successor, 0, ring_comm);
    MPI_Recv(y + recv_offset, blocksize, MPI_FLOAT,
             predecessor, 0, ring_comm);
}

```

- The point of using non-blocking communication is so that we can overlap communication and computation
- To do this for the above example, we will move the first computation of *send_offset* and *recv_offset* outside the loop in order to get things started

An Allgather Example: Non-blocking Communication

```

MPI_Request send_request;
MPI_Request recv_request;

send_offset = ((my_rank - i + p) % p) * blocksize;
recv_offset = ((my_rank - i - 1 + p) % p) * blocksize;

for (i = 0; i < p-1; i++) { /* post the communication operations*/
    MPI_Isend(y + send_offset, blocksize, MPI_FLOAT,
              successor, 0, ring_comm, &send_request);
    MPI_Irecv(y + recv_offset, blocksize, MPI_FLOAT,
              predecessor, 0, , ring_comm, &recv_request);

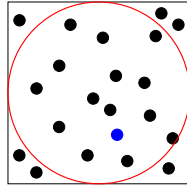
    send_offset = ((my_rank - i + p) % p) * blocksize;
    recv_offset = ((my_rank - i - 1 + p) % p) * blocksize;

    MPI_Wait(&send_request, &status);
    MPI_Wait(&recv_request, &status);
}

```

A Strange Way to Estimate π

Consider a circle of radius 1 and a square drawn around it.



- The area of the circle is π and the area of the square is 4.
- Let r be the ratio of the Area of circle / Area of square, so $r = \pi/4$, and thus $\pi = 4r$
- We can find π by generating random points (x, y) in the square and counting how many are inside the circle:

$$(x, y) : x^2 + y^2 < 1$$

Define two groups and communicators:

```

MPI_Comm world, workers;
MPI_Group world_group, workers_group;
int numprocs, myid, server, ranks[1];

MPI_Init (&argc, &argv);
world = MPI_COMM_WORLD;
MPI_Comm_size (world, &numprocs);
MPI_Comm_rank (world, &myid);
server = numprocs-1; /* use last process as the rand server */

/* Get the group of all processes */
MPI_Comm_group (world, &world_group);
ranks[0] = server;

/* Create a new group by excluding the last process */
MPI_Group_excl (world_group, 1, ranks, &worker_group);

/* Create a new communicator for the new group*/
MPI_Comm_create (world, worker_group, &workers);

MPI_Group_free (&worker_group);
MPI_Group_free (&world_group);

```

Monte Carlo computation for π

```

#include <math.h>
#include <mpi.h>
#include <mpe.h>      /* Using MPI graphics library */
#define CHUNKSIZE    1000

/* message tags */
#define REQUEST 1
#define REPLY 2

int main (int argc, char*argv[])
{
    int iter;
    int in, out, i, iters, max, ix, iy, ranks[1], done, temp;
    double x, y, Pi, error, epsilon;
    int numprocs, myid, server, totalin, totalout, workerid;
    int rands[CHUNKSIZE], request;
    MPI_Comm world, workers;
    MPI_Group world_group, workers_group;
    MPI_Status status;

```

```

    MPI_Init (&argc, &argv);
    world = MPI_COMM_WORLD;
    MPI_Comm_size (world, &numprocs);
    MPI_Comm_rank (world, &myid);
    server = numprocs-1; /* use last process as the rand server */

    if (myid == 0) sscanf (argv[1], "%lf", &epsilon);
    MPI_Bcast (&epsilon, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    /* Get the group of all processes */
    MPI_Comm_group (world, &world_group);
    ranks[0] = server;

    /* Create a new group by excluding the last process */
    MPI_Group_excl (world_group, 1, ranks, &worker_group);

    /* Create a new communicator for the new group*/
    MPI_Comm_create (world, worker_group, &workers);
    MPI_Group_free (&worker_group);

```

```
if (myid == server) {

    /* Do rand server code */

} else {

    /* Do worker code */

}

if (myid == 0) {
    printf ("\npoints: %d\nin: %d, out: %d, <ret> to exit\n",
            totalin+totalout, totalin, totalout);
    getchar();
}

MPI_Comm_free (&workers);
MPI_Finalize ();
}
```

The rand Server Code

```
do {

    MPI_Recv (&request, 1, MPI_INT, MPI_ANY_SOURCE, REQUEST,
             world, &status);

    if (request) {
        for (i = 0; i < CHUNKSIZE; i++) rands[i] = random();
        MPI_Send (rands, CHUNKSIZE, MPI_INT, status.MPI_SOURCE,
                 REPLY, world);
    }

} while (request > 0);
```

The Worker Code

```

request = 1; done = in = out = 0;
max = INT_MAX; /* max int for normalization */
MPI_Send (&request, 1, MPI_INT, server, REQUEST, world);
MPI_Comm_rank (workers, &workerid);
iter = 0;

while (!done) {
    iter++;
    request = 1;
    MPI_Recv (rands, CHUNKSIZE, MPI_INT, server, REPLY,
             world, &status);
    for (i=0; i<CHUNKSIZE; i++) {
        x = (((double) rands[i++])/max) * 2 -1;
        y = (((double) rands[i++])/max) * 2 -1;
        if (x*x + y*y < 1.0) in++; else out++;
    }

    MPI_Allreduce (&in, &totalin, 1, MPI_INT, MPI_SUM, workers);
    MPI_Allreduce (&out, &totalout, 1, MPI_INT, MPI_SUM, workers);
}

```

```

Pi = (4.0 * totalin)/(totalin + totalout);
error = fabs (Pi - 3.141592653589793238462643);
done = (error < epsilon || (totalin+totalout) > 1000000);
request = (done) ? 0 : 1;

if (myid == 0) {
    printf ("\npi = %23.20f", Pi);
    MPI_Send (&request, 1, MPI_INT, server, REQUEST, world);
} else {
    if (request) MPI_Send (&request, 1, MPI_INT, server, REQUEST,
                          world);
}
} /* endwhile */

```

Using the MPI X11 Real-Time Graphics Library

- All processes must include the `<mpe.h>` file and declare
`MPE_XGraph graph;`
- Near the beginning of the program, all processes execute
`MPE_Open_graphics (&graph, MPI_COMM_WORLD, (char *)0, -1, -1,
WINDOW_SIZE, WINDOW_SIZE, MPE_GRAPH_INDEPENDENT);`
`(char *)0` signifies the use of the default display from the user's environment
- At the end of the program, all processes execute
`MPE_Close_graphics (&graph);`
to terminate access to the display

- To draw the (x, y) point generated in the program, use
`MPE_Draw_point (graph,
(int) (WINDOW_SIZE/2 + x* WINDOW_SIZE/2),
(int) (WINDOW_SIZE/2 - y* WINDOW_SIZE/2),
MPE_BLACK);`
- To flush all the points buffered up to this point onto the display, use
`MPE_Update (graph);`
- Traffic to the X server can be cut down by buffering a large number of calls to `MPE_Draw_point`

Other MPI Features

- Graph Topologies
- Creating Processes
- Writing Libraries

- Miscellaneous Features
- LAM/MPI Extensions
 - Remote file access
 - Collective I/O
 - Signal Handling
 - Debugging and Tracing
- LAM Command Reference