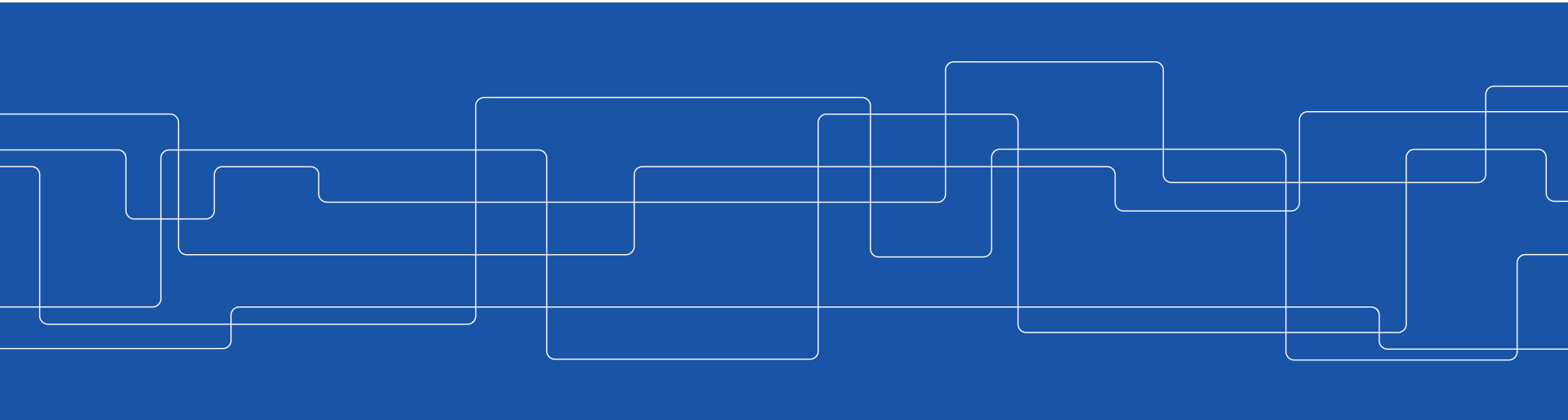




Case Study: MPI: Message Passing Interface.

Vladimir Vlassov





Outline

- The MPI library: functions, types;
- MPI programming concepts: processes, communicators, etc.;
- Basic MPI functions, blocking/non-blocking send/receive;
- Collective operations: barriers, broadcast, gather/scatter, reduce, etc.;
- Communicators.



References

- Chapter 4.6 Case Study: MPI, in “*Distributed Systems – Concepts and Design*”, Coulouris et al, Addison Wesley
- MPICH - A Portable Implementation of MPI
<http://www.mpich.org/>
- MPI Tutorials and Other documents
<http://www.mpich.org/documentation/guides/>
<http://www.mcs.anl.gov/research/projects/mpi/learning.html>

MPI: Message Passing Interface

- **MPI** is a message-passing library specification for multiprocessors, clusters, and heterogeneous networks
 - Not a compiler specification, not a specific product.
 - Message-passing model and API
 - Designed
 - To allow the development of parallel software libraries
 - To provide access to advanced parallel hardware for end users, library writers, and tool developers.
 - MPI is a de facto standard for message passing systems
 - Language bindings: Fortran and C
- **MPICH** and **LAM** – are the two most popular implementations of MPI today
 - MPICH is more generally usable on various platforms
 - LAM – for TCP/IP networks
- Another message-passing programming environment: **PVM** (Parallel Virtual Machine)



The MPI Library (API)

The MPI library is large: about 130 functions

- extensive functionality and flexibility
- message passing (point-to-point, collective)
- process groups and topologies

The MPI library is “small”: only six functions allow writing many programs.

- `MPI_Init`, `MPI_Finalize`,
`MPI_Comm_size`, `MPI_Comm_rank`,
`MPI_Send`, `MPI_Recv`

All MPI functions, constants, and data types have the prefix `MPI_`

C Data Types Mapping in MPI

`mpi.h`

MPI datatype	C datatype
<code>MPI_CHAR</code>	<code>signed char</code>
<code>MPI_SHORT</code>	<code>signed short int</code>
<code>MPI_INT</code>	<code>signed int</code>
<code>MPI_LONG</code>	<code>signed long int</code>
<code>MPI_UNSIGNED_CHAR</code>	<code>unsigned char</code>
<code>MPI_UNSIGNED_SHORT</code>	<code>unsigned short int</code>
<code>MPI_UNSIGNED</code>	<code>unsigned int</code>
<code>MPI_UNSIGNED_LONG</code>	<code>unsigned long int</code>
<code>MPI_FLOAT</code>	<code>float</code>
<code>MPI_DOUBLE</code>	<code>double</code>
<code>MPI_LONG_DOUBLE</code>	<code>long double</code>
<code>MPI_BYTE</code>	
<code>MPI_PACKED</code>	

MPI Programming Concepts: Processes

MPI process

- The smallest unit of computation, e.g., a Unix process.
- Uniquely identified by its rank in a process group (communicator).
- Processes communicate with tagged messages within process groups identified by communicators (communication contexts)
- A proc can send messages either synchronously or asynchronously.
- The proc is responsible for receiving messages directed to it.
- The proc uses either blocking or non-blocking receive/send.
- The proc can participate in a collective operation such as broadcast, gather, or barrier, that involves all processes in a group.

MPI application

- a static set of processes that interact to solve a problem.
- `mpiexec -np 2 hello` – starts two `hello` processes.



Communicators. Process groups. Ranks

Communicator

- An opaque object that defines a process group and a communication context for the group. Delimits scope of communication.
 - Separate groups of processes working on sub-problems, each with a specific communication context identified by a communicator.

Process Group

- A virtual set of processes identified by a communicator.
 - Grouped for collective and point-to-point communications.
 - All communication (not just collective operations) takes place in groups.
 - Each process has a unique index (rank) in a given group.

Rank

- A unique identifier (index) of a process within a communicator (in a proc group)
- Ranks are in the range 0, ..., size-1 (where size is the size of the group)



Developing, Building and Running an MPI Application

- **Install** one of the MPI implementations, e.g., MPICH-2
- **Develop** your application in an SPMD-style – one program for all processes
 - The number of processes is specified at run time (could be 1)
 - Use the number of proc and proc ranks to determine process tasks
 - In C: `#include <mpi.h>`
- **Compile and build:**
`mpicc -o myprog myprog.c`
 - For large projects, develop a `Makefile`
 - To compile C++: `mpicc -cc=g++ -o myprog myprog.cpp`
 - The option `-help` shows all options to `mpicc`.
- **Run:**
`mpiexec -np 2 myprog`
 - The option `-help` shows all options to `mpiexec`.
 - Alternatively, you can call `mpirun` (which is a link to `mpiexec`)



Process Managers (1/2)

Process managers (e.g., Hydra, MPD) are external distributed agents that spawn and manage parallel jobs.

- A process manager communicates with MPICH processes using a predefined interface called PMI (process management interface).
- You can use any process manager if they follow the same wire protocol.
- Three known implementations of the PMI wire protocol: "**simple**," "**smpd**," and "**slurm**" (default: **simple**)

MPICH provides several different process managers, e.g.

- **Hydra**, **MPD**, **Gforker**, and **Remshell** follow the "simple" PMI wire protocol.



Process Managers (2/2)

MPD has been the traditional default process manager for MPICH till the 1.2.x release series.

Starting the 1.3.x series, **Hydra is the default process manager.**

See “*Using the Hydra Process Manager*” at

https://github.com/pmodels/mpich/blob/main/doc/wiki/how_to/Using_the_Hydra_Process_Manager.md



Basic MPI Functions

`int MPI_Init(int *argc, char **argv[])`

- Start MPI, and enroll the process in the MPI application.

`int MPI_Finalize()`

- Stop (exit) MPI

`int MPI_Comm_size(MPI_Comm comm, int *size)`

- Determine the number of processes in the group `comm`.
 - `comm` – communicator, e.g., `MPI_COMM_WORLD`
 - `size` – number of processes in the group (returned)

`int MPI_Comm_rank(MPI_Comm comm, int *rank)`

- Determine the rank of the calling process in the group `comm`.
 - `comm` – communicator, e.g., `MPI_COMM_WORLD`
 - `rank` – the rank (returned) is a number between zero and `size-1`

Example: “Hello World”

```
#include "mpi.h"
#include <stdio.h>
int main( argc, argv )
{
    int argc;
    char **argv;
    {
        int rank, size;
        MPI_Init( &argc, &argv );
        MPI_Comm_rank( MPI_COMM_WORLD, &rank );
        MPI_Comm_size( MPI_COMM_WORLD, &size );
        printf( "Hello world! I'm %d of %d\n", rank, size );
        MPI_Finalize();
        return 0;
    }
}
```

Basic (Blocking) Send

```
int MPI_Send(void *buf, int count, MPI_datatype dt,  
             int dest, int tag, MPI_Comm comm)
```

- Send a message with the given tag to the given destination in the given communicator
 - buf** send buffer
 - count** number of items in the buffer
 - dt** data type of items
 - dest** destination process rank
 - tag** message tag
 - comm** communicator
- Returns integer result code as for all MPI functions, normally **MPI_SUCCESS**
- **datatype** can be elementary, a continuous array of data types, stridden blocks of data types, an indexed array of data types, and a general structure.

Basic (Blocking) Receive

```
int MPI_Recv( void *buf, int count, MPI_datatype dt,  
              int source, int tag, MPI_Comm comm,  
              MPI_Status *status)
```

- Receive a message with the given tag from the given source in the given communicator
 - buf** receive buffer (loaded)
 - count** max number of entries in the buffer
 - dt** data type of entries
 - source** source process rank
 - tag** message tag
 - comm** communicator
 - status** status (returned).
- “Wildcard” values are provided for tag (**MPI_ANY_TAG**) and source (**MPI_ANY_SOURCE**).

Inspecting Received Message

- If wildcard values are used for tag and sources, the received message can be inspected via an **MPI_Status** structure with three components **MPI_SOURCE**, **MPI_TAG**, and **MPI_ERROR**.

```
MPI_Status status;  
MPI_Recv( ..., &status );  
int tag_received = status.MPI_TAG;  
int rank_of_source = status.MPI_SOURCE;  
MPI_Get_count( &status, datatype, &count );
```

- **MPI_Get_count** is used to determine how much data of a particular type has been received.

Communication Modes of Blocking Send

Standard blocking send (non-local) `MPI_Send`

- Implementation defined buffering: If the message is buffered, the send may complete before a matching receive is invoked.

Buffered blocking send (local) `MPI_Bsend`

- Can be started whether or not a matching receive has been posted;
- May complete before a matching receive is posted.

Synchronous blocking send (non-local) `MPI_Ssend`

- Can be started whether or not a matching receive has been posted;
- Completes when a matching receive is posted and has started to receive.

Ready blocking send (non-local) `MPI_Rsend`

- Can be started only if the matching receive is already posted;
- Otherwise, the outcome is undefined.

Example 1: Exchange of Messages (Always Succeeds)

```
MPI_Comm_rank(comm, &rank);  
if (rank == 0) {  
    MPI_Send(sendbuf, count, MPI_REAL, 1, tag, comm);  
    MPI_Recv(recvbuf, count, MPI_REAL, 1, tag, comm, &status);  
}  
if (rank == 1) {  
    MPI_Recv(recvbuf, count, MPI_REAL, 0, tag, comm, &status);  
    MPI_Send(sendbuf, count, MPI_REAL, 0, tag, comm);  
}
```

- This program will succeed even if no buffer space for data is available.
- In this example, the standard send operation can be replaced with asynchronous send.

Example 2: Exchange of Messages (Always Deadlocks)

```
MPI_Comm_rank(comm, &rank);  
if (rank == 0) {  
    MPI_Recv(recvbuf, count, MPI_REAL, 1, tag, comm, &status);  
    MPI_Send(sendbuf, count, MPI_REAL, 1, tag, comm);  
}  
if (rank == 1) {  
    MPI_Recv(recvbuf, count, MPI_REAL, 0, tag, comm, &status);  
    MPI_Send(sendbuf, count, MPI_REAL, 0, tag, comm);  
}
```

- This program will always deadlock!
- The same holds for any other send mode.

Example 3: Exchange of Messages (Relies on Buffering)

```
MPI_Comm_rank(comm, &rank);  
if (rank == 0)  
{  
    MPI_Send(sendbuf, count, MPI_REAL, 1, tag, comm);  
    MPI_Recv(recvbuf, count, MPI_REAL, 1, tag, comm, &status);  
}  
if (rank == 1)  
{  
    MPI_Send(sendbuf, count, MPI_REAL, 0, tag, comm);  
    MPI_Recv(recvbuf, count, MPI_REAL, 0, tag, comm, &status);  
}
```

- For the program to complete, at least one of the two messages must be buffered.
- The program can succeed only if the communication system can buffer at least count words of data.

Non-Blocking Communication Operations

Non-blocking send initiates sending:

```
int MPI_Isend(void* buf, int count, MPI_Datatype type,  
             int dest, int tag, MPI_Comm comm,  
             MPI_Request *request)
```

Non-blocking receive initiates receiving:

```
int MPI_Irecv(void* buf, int count, MPI_Datatype type,  
             int source, int tag, MPI_Comm comm,  
             MPI_Request *request)
```

- A request object is returned in **request** to identify the operation.

Non-Blocking Operations (cont'd)

To query the status of communication or to wait for its completion:

```
int MPI_Test( MPI_Request *request, int *flag,  
              MPI_Status *status)
```

- Returns immediately with **flag = true** if the operation identified by **request** has been completed,
Otherwise returns immediately with **flag = false**.

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

- Returns when the operation identified by **request** completes.

Example: Non-Blocking Send/Receive

```
MPI_Comm_rank(comm, &rank);
if (rank = 0)
{
    MPI_Isend(a, 10, MPI_REAL, 1, tag, comm, request);
    /*** do some computation to mask latency ***/
    MPI_Wait(request, &status);
}
if (rank = 1)
{
    MPI_Irecv(a, 10, MPI_REAL, 0, tag, comm, request);
    /*** do some computation to mask latency ***/
    MPI_Wait(request, &status);
}
}
```



Probing for Pending Messages

Non-blocking/blocking check for an incoming message
without receiving it

MPI_Iprobe(source, tag, comm, flag, status)

- polls for pending messages

MPI_Probe(source, tag, comm, status)

- returns when a message is pending

Collective Operations

A **collective operation** is executed by having **all** processes in the communicator **call the same** communication routine with matching arguments.

- Several collective routines have a single originating or receiving process – the **root**.
- Some arguments in the collective functions are “significant only at the root” and are ignored by all participants except the root.



Collective Communication

Collective synchronization (barrier):

```
int MPI_Barrier( MPI_Comm comm )
```

Broadcast from **buf** of **root** to all processes:

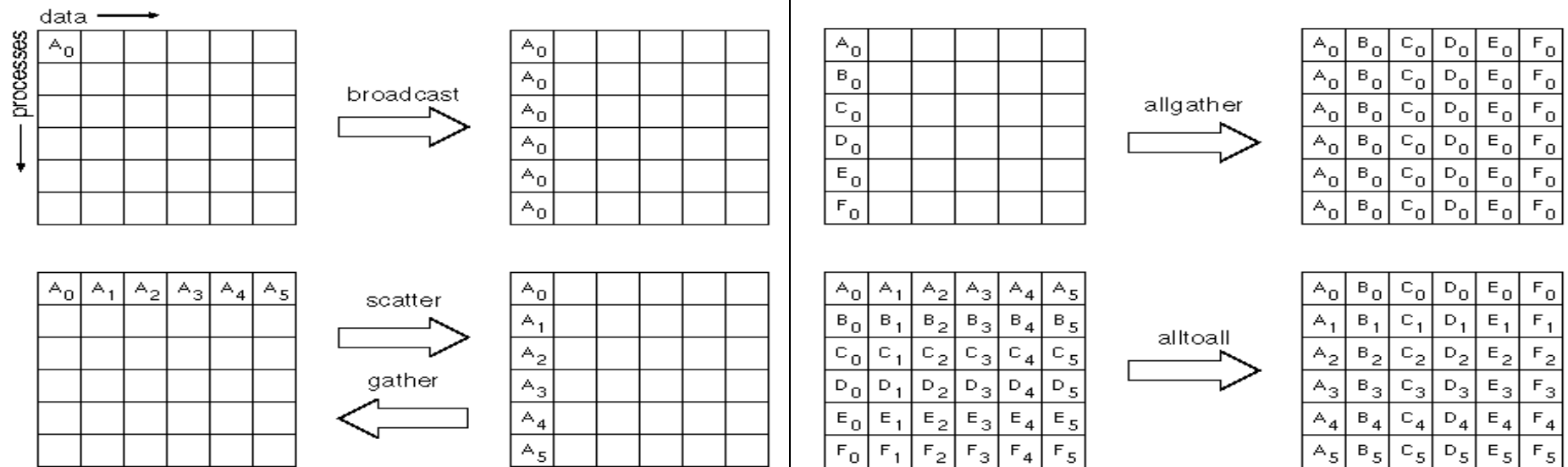
```
int MPI_Bcast( void *buf, int count, MPI_datatype dt, int root, MPI_Comm comm )
```

Collective data transfer:

```
int MPI_Gather( void *sendbuf, int sendcount, MPI_datatype sendtype,  
               void *recvbuf, int recvcount, MPI_datatype recvtype,  
               int root, MPI_Comm comm )
```

```
int MPI_Scatter( void *sendbuf, int sendcount, MPI_datatype sendtype,  
                void *recvbuf, int recvcount, MPI_datatype recvtype,  
                int root, MPI_Comm comm )
```

Collective Data Transfer Operations





Example 1: Broadcast

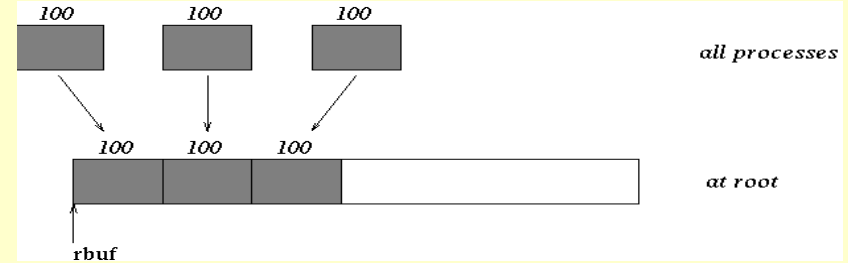
Broadcast 100 integers from process 0 (root) to every process in the group.

```
MPI_Comm comm;  
    int array[100];  
    int root=0;  
    ...  
    MPI_Bcast( array, 100, MPI_INT, root, comm);  
}
```

Example 2: Gather

Gather 100 integers from every process in group to root.

```
MPI_Comm comm;  
int gsize, sendarray[100];  
int root, myrank, *rbuf;  
...  
MPI_Comm_rank(comm, myrank);  
MPI_Comm_size(comm, &gsize);  
...  
if (myrank == root) rbuf = (int *)malloc(gsize*100*sizeof(int));  
MPI_Gather(sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```

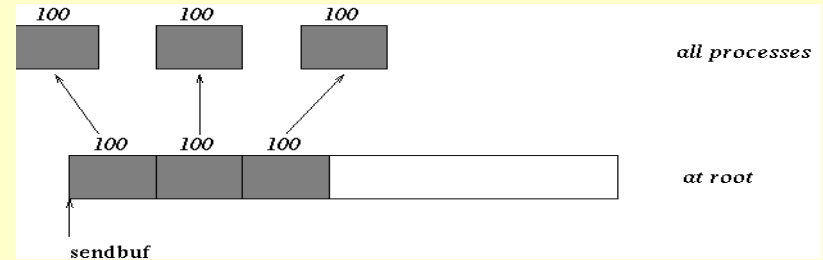


Example 3: Scatter

The reverse of Example 2 (previous slide): Scatter sets of 100 integers from the root to each process in the group.

```

MPI_Comm comm;
int gsize,*sendbuf;
int root, rbuf[100];
...
MPI_Comm_rank(comm, myrank);
MPI_Comm_size(comm, &gsize);
if ( myrank == root) {
    sendbuf = (int *)malloc(gsize*100*sizeof(int));
    ... // fill the send buffer with data to be scattered
}
MPI_Scatter(sendbuf, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
    
```



Global Reduction Operations

Reduce data from send buffers of all participating processes into a receive buffer of the **root** proc using operation **op**

```
int MPI_Reduce( void *sendbuf, void *recvbuf, int count,
                MPI_datatype dt, MPI_Op op, int root, MPI_Comm comm)
```

Reduce data from send buffers of all participating processes into receive buffers of all the processes using operation **op**

```
int MPI_Allreduce( void * sendbuf, void * recvbuf, int count,
                   MPI_datatype dt, MPI_Op op, MPI_Comm comm )
```

Available operations (**MPI_Op op**): **MPI_MAX**, **MPI_MIN**, **MPI_SUM**, **MPI_LAND**, **MPI_BOR**, ...

Timing Functions

`double MPI_Wtime(void)`

- Returns a floating-point number of seconds, representing elapsed wall-clock time since some time in the past. The time is “local” on the host.

`double MPI_Wtick(void)`

- Returns the resolution of `MPI_WTIME` in seconds, the number of seconds between successive clock ticks.

Example:

```
{
    double starttime, endtime;
    starttime = double MPI_Wtime();
    .... stuff to be timed ...
    endtime   = double MPI_Wtime();
    printf("That took %f seconds\n", endtime - starttime);
}
```




Modularity: Communicators

A **communicator** identifies a process group and provides a context for all communication within the group

- Communicator acts as an extra tag on messages

The communicator **MPI_COMM_WORLD** identifies all running processes of the MPI application.

Create/Destroy Communicators

- Create new communicator: same group, new context:

```
int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
```

- Create new communicators based on colors, ordered by keys:

```
int MPI_Comm_split( MPI_Comm comm, int color, int key, MPI_Comm *newcomm )
```

color identifies a group, **key** – rank in the group

- Create an inter-communicator from two intra-communicators

```
int MPI_Intercomm_create( MPI_Comm local_comm, int local_leader,  
                          MPI_Comm peer_comm, int remoteleader,  
                          int tag, MPI_Comm *inter_comm)
```

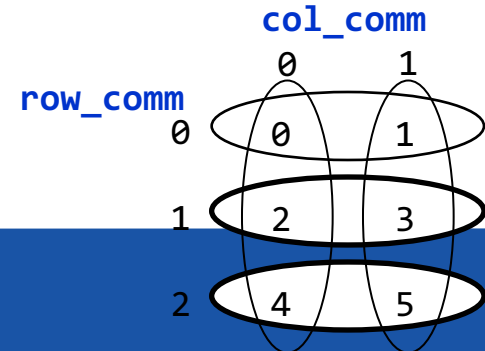
- Destroy a communicator

```
int MPI_Comm_free(MPI_Comm *comm)
```

Example MPI_Comm_split

```
// 2D topology with nrow rows and mcol (2) columns
// Split 3x2 grid into 2 communicators
// one (row_comm) corresponds to 3 rows;
// another (col_comm) - to 2 columns
irow = myID / mcol;           // logical row number
jcol = myID % mcol;           // logical column number
int row_comm, col_comm;
MPI_Comm_split(MPI_COMM_WORLD, irow, jcol, &row_comm);
MPI_Comm_split(MPI_COMM_WORLD, jcol, irow, &col_comm);
```

myID	0	1	2	3	4	5
irow	0	0	1	1	2	2
jcol	0	1	0	1	0	1





Communicators (cont'd)

One use of communicators is for calling parallel library routines in different context:

```
MPI_Comm *newcomm;  
...  
MPI_Comm_dup(comm, newcomm);  
transpose(newcomm, matrix); /* call library function */  
MPI_Comm_free(newcomm);
```



Example: Compute PI

```
#include "mpi.h"
#include <math.h>
int main(int argc, char *argv[]) {
    int done = 0, n, myid, numprocs, i, rc;
    double PI25DT = 3.141592653589793238462643, mypi, pi, h, sum, x, a;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    while (!done) {
        if (myid == 0) {
            printf("Enter the number of intervals: (0 quits) "); scanf("%d",&n);
        }
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
        if (n == 0) break;
        h = 1.0 / (double) n;
        sum = 0.0;
        for (i = myid + 1; i <= n; i += numprocs) {
            x = h * ((double)i - 0.5);
            sum += 4.0 / (1.0 + x*x);
        }
        mypi = h * sum;
        MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
        if (myid == 0) printf("pi is approximately %.16f, Error is %.16f\n", pi, fabs(pi - PI25DT));
    }
    MPI_Finalize();
}
```