MPI Tutorial

Purushotham Bangalore, Ph.D.
Anthony Skjellum, Ph.D.

Department of Computer and Information Sciences
University of Alabama at Birmingham

Overview

- Message Passing Interface - MPI
  - Point-to-point communication
  - Collective communication
  - Communicators
  - Datatypes
  - Topologies
  - Inter-communicators
  - Profiling

Message Passing Interface (MPI)

- A message-passing library specification
  - Message-passing model
  - Not a compiler specification
  - Not a specific product
- For parallel computers, clusters, and heterogeneous networks
- Designed to aid the development of portable parallel software libraries
- Designed to provide access to advanced parallel hardware for
  - End users
  - Library writers
  - Tool developers

Message Passing Interface - MPI

- MPI-1 standard widely accepted by vendors and programmers
  - MPI implementations available on most modern platforms
  - Huge number of MPI applications deployed
  - Several tools exist to trace and tune MPI applications
- MPI provides rich set of functionality to support library writers, tools developers and application programmers

MPI Salient Features

- Point-to-point communication
- Collective communication on process groups
- Communicators and groups for safe communication
- User defined datatypes
- Virtual topologies
- Support for profiling

A First MPI Program

```
#include <stdio.h>
#include <mpi.h>
main( int argc, char **argv )
{
   MPI_Init ( &argc, &argv );
   printf ( "Hello World!\n" );
   MPI_Finalize ( );
}
```

```
program main
include 'mpif.h'
integer ierr
call MPI_INIT( ierr )
print *, 'Hello world!'
call MPI_FINALIZE( ierr )
end
```


Starting the MPI Environment

- **`MPI_INIT ( )`**

  Initializes MPI environment. This function must be called and must be the first MPI function called in a program (exception: **`MPI_INITIALIZED`**)

  Syntax

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

MPI_INIT ( IERROR )
INTEGER IERROR
```


Exiting the MPI Environment

- **`MPI_FINALIZE (  )`**

  Cleans up all MPI state. Once this routine has been called, no MPI routine (even **`MPI_INIT`**) may be called

  Syntax

```
int MPI_Finalize ( );

MPI_FINALIZE  ( IERROR )
INTEGER IERROR
```

C and Fortran Language Considerations, I.

- MPI_INIT: The C version accepts the argc and argv variables that are provided as arguments to main ( )
- Error codes: Almost all MPI Fortran subroutines have an integer return code as their last argument. Almost all C functions return an integer error code
- Types: Opaque objects are given type names in C. Opaque objects are usually of type INTEGER in Fortran (exception: binary-valued variables are of type LOGICAL)
- Inter-language interoperability is not guaranteed

C and Fortran Language Considerations, II.

- Bindings
  - C
    - All MPI names have an MPI_ prefix
    - Defined constants are in all capital letters
    - Defined types and functions have one capital letter after the prefix; the remaining letters are lowercase
  - Fortran
    - All MPI names have an MPI_ prefix
    - No capitalization rules apply

Finding Out About the Parallel Environment

- Two of the first questions asked in a parallel program are:
  - "How many processes are there?"
  - "Who am I?"
- "How many" is answered with the function call **`MPI_COMM_SIZE()`**
- "Who am I" is answered with the function call **`MPI_COMM_RANK()`**
  - The rank is a number between **`zero`** and **`(size - 1)`**

Example 1 (Fortran)

```
program main
include 'mpif.h'
integer rank, size, ierr
call MPI_INIT( ierr )
call MPI_COMM_RANK( MPI_COMM_WORLD, rank, ierr )
call MPI_COMM_SIZE ( MPI_COMM_WORLD, size, ierr )
print *, 'Process ', rank, ' of ', size, ' is alive'
call MPI_FINALIZE( ierr )
end
```

Example 1 (C)

```
#include <mpi.h>
main( 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 ( "Process %d of %d is alive\n", rank,
     size );
   MPI_Finalize ( );
}
```


Communicator

- Communication in **MPI** takes place with respect to communicators
- **MPI_COMM_WORLD** is one such predefined communicator (something of type **"MPI_COMM"**) and contains group and context information
- **MPI_COMM_RANK** and **MPI_COMM_SIZE** return information based on the communicator passed in as the first argument
- Processes may belong to many different communicators

Environment Setup

Using the CIS Cluster

- Login
  - ssh everest00.cis.uab.edu
  - Add the following lines at the end of your .bashrc file:
    export PATH=/opt/mpipro/bin:${PATH}
    export LD_LIBRARY_PATH=/opt/mpipro/lib64:${LD_LIBRARY_PATH}
  - Logout and login again
- Compile
  - mpicc –o program program.c
  - mpic++ –o program program.cc
- Submit
  - qsub myscript.sge
- Monitor
  - qstat –u <userid>
- See User Guide for more details
  - http://www.cis.uab.edu/ccl/resources/everest/EverestGridNodeUserGuide.php

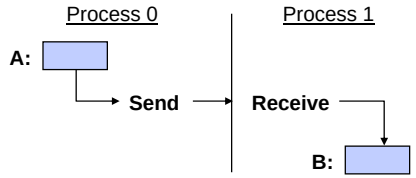
Sample SGE script

```
#!/bin/bash
#
#$ -cwd
#$ -j y
#$ -S /bin/bash
#
#$ -pe mpi 4
MPI_DIR=/opt/mpipro/bin
EXE="/home/puri/examples/psum 1000"
$MPI_DIR/mpirun -np $NSLOTS -machinefile $TMPDIR/machines $EXE
```


Point-to-Point Communications

Sending and Receiving Messages

- Basic message passing process



- Questions
  - To whom is data sent?
  - Where is the data?
  - What type of data is sent?
  - How much of data is sent?
  - How does the receiver identify it?

Message Organization in MPI

- Message is divided into data and envelope
- data
  - buffer
  - count
  - datatype
- envelope
  - process identifier (source/destination rank)
  - message tag
  - communicator

Generalizing the Buffer Description

- Specified in MPI by starting address, count, and datatype, where datatype is as follows:
  - Elementary (all C and Fortran datatypes)
  - Contiguous array of datatypes
  - Strided blocks of datatypes
  - Indexed array of blocks of datatypes
  - General structure
- Datatypes are constructed recursively
- Specifying application-oriented layout of data allows maximal use of special hardware
- Elimination of length in favor of count is clearer
  - Traditional: send 20 bytes
  - MPI: send 5 integers

MPI C Datatypes

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

MPI Fortran Datatypes

MPI FORTRAN	FORTRAN datatypes
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_REAL8	REAL*8
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER
MPI_BYTE	
MPI_PACKED	

Process Identifier

- MPI communicator consists of a *group* of processes
  - Initially “all” processes are in the group
  - MPI provides group management routines (to create, modify, and delete groups)
- All communication takes place among members of a group of processes, as specified by a communicator
- Naming a process
  - **`destination`** is specified by **`( rank, group )`**
  - Processes are named according to their rank in the group
  - Groups are enclosed in “communicator”
  - **`MPI_ANY_SOURCE`** wildcard rank permitted in a receive

Message Tag

- Tags allow programmers to deal with the arrival of messages in an orderly manner
- MPI tags are guaranteed to range from 0 to 32767
- The upper bound on tag value is provided by the attribute MPI_TAG_UB
- **`MPI_ANY_TAG`** can be used as a wildcard value

MPI Basic Send/Receive

- Thus the basic (blocking) send has become:
  `MPI_Send ( start, count, datatype, dest, tag, comm )`
- And the receive has become:
  `MPI_Recv( start, count, datatype, source, tag, comm, status )`
- The source, tag, and the count of the message actually received can be retrieved from status

Bindings for Send and Receive

```
int MPI_Send( void *buf, int count, MPI_Datatype
type, int dest, int tag, MPI_Comm comm )

MPI_SEND( BUF, COUNT, DATATYPE, DEST, TAG, COMM,
IERR )
   <type> BUF( * )
   INTEGER COUNT, DATATYPE, DEST, COMM, IERR

int MPI_Recv( void *buf, int count, MPI_Datatype
datatype, int source, int tag, MPI_Comm comm,
MPI_Status *status )

MPI_RECV( BUF, COUNT, DATATYPE, SOURCE, TAG, COMM,
STATUS, IERR )
   <type> BUF ( * )
   INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM,
   STATUS( MPI_STATUS_SIZE ), IERR
```


Getting Information About a Message

- The following functions can be used to get information about a message

```
MPI_Status status;
MPI_Recv( . . . , &status );

tag_of_received_message = status.MPI_TAG;
src_of_received_message = status.MPI_SOURCE;
MPI_Get_count( &status, datatype, &count);
```

- MPI_TAG and MPI_SOURCE are primarily of use when MPI_ANY_TAG and/or MPI_ANY_SOURCE is used in the receive
- The function MPI_GET_COUNT may be used to determine how much data of a particular type was received

Getting Information About a Message (Fortran)

- The following functions can be used to get information about a message

```
INTEGER status(MPI_STATUS_SIZE)
call MPI_Recv( . . . , status, ierr )

tag_of_received_message = status(MPI_TAG)
src_of_received_message = status(MPI_SOURCE)
call MPI_Get_count(status, datatype, count, ierr)
```

- MPI_TAG and MPI_SOURCE are primarily of use when MPI_ANY_TAG and/or MPI_ANY_SOURCE is used in the receive
- The function MPI_GET_COUNT may be used to determine how much data of a particular type was received

Example-2, I.

```
      program main
      include 'mpif.h'
      integer rank, size, to, from, tag, count, i, ierr, src, dest
      integer integer st_source, st_tag, st_count, status(MPI_STATUS_SIZE)
      double precision data(100)

      call MPI_INIT(ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
      print *, 'Process ', rank, ' of ', size, ' is alive'
      dest = size - 1
      src = 0
C
      if (rank .eq. src) then
        to     = dest
        count  = 100
        tag    = 2001
        do 10 i=1, 100
 10       data(i) = i
         call MPI_SEND(data, count, MPI_DOUBLE_PRECISION, to,tag,
              MPI_COMM_WORLD, ierr)
```


Example-2, II.

```
      else if (rank .eq. dest) then
        tag   = MPI_ANY_TAG
        count = 100
        from  = MPI_ANY_SOURCE
        call MPI_RECV(data, count, MPI_DOUBLE_PRECISION, from,
     +       tag, MPI_COMM_WORLD, status, ierr)
        call MPI_GET_COUNT(status, MPI_DOUBLE_PRECISION,
     +       st_count, ierr)
        st_source = status(MPI_SOURCE)
        st_tag    = status(MPI_TAG)
C
        print *, 'Status info: source = ', st_source,
     +       ' tag = ', st_tag, ' count = ', st_count
        print *, rank, ' received', (data(i),i=1,10)
      endif

      call MPI_FINALIZE(ierr)
      stop
      end
```


Example-2, I.

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

int main(int argc, char **argv) {
  int i, rank, size, dest;
  int to, src, from, count, tag;
  int st_count, st_source, st_tag;
  double data[100];
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  printf("Process %d of %d is alive\n", rank, size);

  dest = size - 1;
  src = 0;
```

Example-2, II.

```
if (rank == src) {
  to = dest; count = 100; tag = 2001;
  for (i = 0; i < 100; i++)
    data[i] = i;
  MPI_Send(data,count,MPI_DOUBLE,to,tag,MPI_COMM_WORLD);

} else if (rank == dest) {
  tag = MPI_ANY_TAG; count = 100; from = MPI_ANY_SOURCE;
  MPI_Recv(data,count,MPI_DOUBLE,from,tag,MPI_COMM_WORLD,
          &status);

  MPI_Get_count(&status, MPI_DOUBLE, &st_count);
  st_source= status.MPI_SOURCE;
  st_tag= status.MPI_TAG;
  printf("Status info: source = %d, tag = %d, count = %d\n",
         st_source, st_tag, st_count);
  printf(" %d received: ", rank);
}
 MPI_Finalize();
 return 0;
}
```

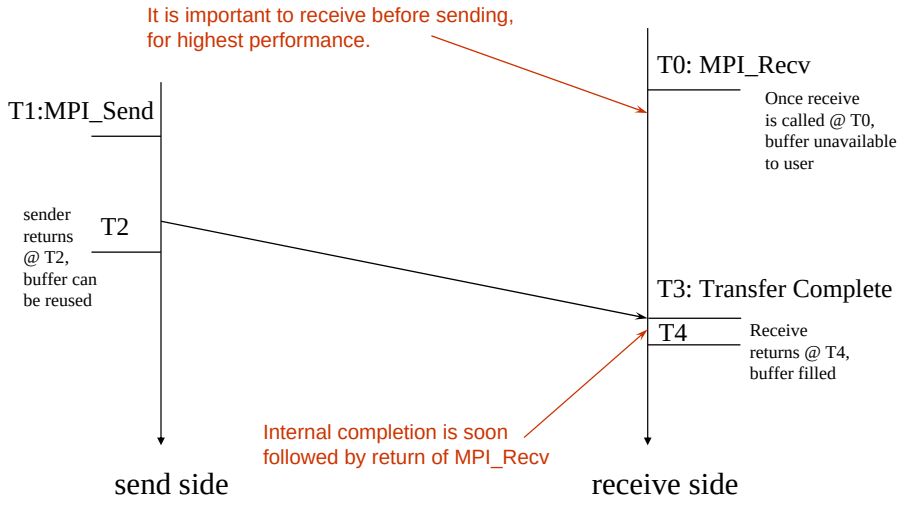

Lab 1

- Objective: Pass a message around a ring **n** times.
- Write a program to do the following:
  - Process 0 should read in a single integer (`>0`) from standard input
  - Use **`MPI_SEND`** and **`MPI_RECV`** to pass the integer around a ring
  - Use the user-supplied integer to determine how many times to pass the message around the ring
  - Process 0 should decrement the integer each time it is received
  - All processes should exit when they receive a “0”
- Refer to the MPI function index

Blocking Communication

- So far we have discussed *blocking* communication
  - **`MPI_SEND`** does not complete until buffer is empty (available for reuse)
  - **`MPI_RECV`** does not complete until buffer is full (available for use)
- A process sending data will be blocked until data in the send buffer is emptied
- A process receiving data will be blocked until the receive buffer is filled
- Completion of communication generally depends on the message size and the system buffer size
- Blocking communication is simple to use but can be prone to deadlocks

Blocking Send-Receive Diagram (Receive before Send)



Non-Blocking Communication

- Non-blocking operations return (immediately) "request handles" that can be waited on and queried

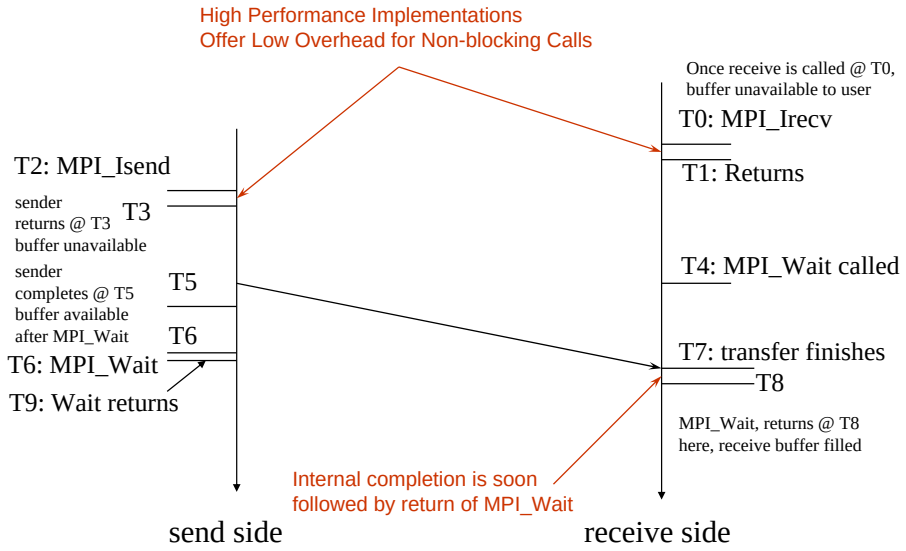
```
MPI_ISEND( start, count, datatype, dest, tag,
  comm, request )
MPI_IRECV( start, count, datatype, src, tag,
  comm, request )
MPI_WAIT( request, status )
```

- Non-blocking operations allow overlapping computation and communication
- One can also test without waiting using **`MPI_TEST`**

```
MPI_TEST( request, flag, status )
```

- Anywhere you use **`MPI_Send`** or **`MPI_Recv`**, you can use the pair of **`MPI_Isend/MPI_Wait`** or **`MPI_Irecv/MPI_Wait`**

Non-Blocking Send-Receive



Multiple Completions

- It is often desirable to wait on multiple requests
- An example is a worker/manager program, where the manager waits for one or more workers to send it a message

```
MPI_WAITALL( count, array_of_requests,
  array_of_statuses )
MPI_WAITANY( count, array_of_requests, index,
  status )
MPI_WAITSOME( incount, array_of_requests,
  outcount, array_of_indices, array_of_statuses )
```

- There are corresponding versions of *test* for each of these viz., **`MPI_Testall, MPI_Testany, MPI_Testsome`**

Probing the Network for Messages

- **`MPI_PROBE`** and **`MPI_IPROBE`** allow the user to check for incoming messages without actually receiving them
- **`MPI_IPROBE`** returns **`"flag == TRUE"`** if there is a matching message available. **`MPI_PROBE`** will not return until there is a matching receive available

```
MPI_IPROBE (source, tag, communicator,
flag, status)
MPI_PROBE ( source, tag, communicator,
status )
```

Message Completion and Buffering

- A send has completed when the user supplied buffer can be reused

```
*buf = 3;
MPI_Send ( buf, 1, MPI_INT, ... );
*buf = 4; /* OK, receiver will always receive 3 */
```

```
*buf = 3;
MPI_Isend(buf, 1, MPI_INT, ...);
*buf = 4; /* Undefined whether the receiver will get 3 or 4 */
MPI_Wait ( ... );
```

- The send mode used (standard, ready, synchronous, buffered) may provide additional information
- Just because the send completes does not mean that the receive has completed
  - Message may be buffered by the system
  - Message may still be in transit

Example-3, I.

```
      program main
      include 'mpif.h'

      integer ierr, rank, size, tag, num, next, from
      integer stat1(MPI_STATUS_SIZE), stat2(MPI_STATUS_SIZE)
      integer req1, req2

      call MPI_INIT(ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)

      tag = 201
      next = mod(rank + 1, size)
      from = mod(rank + size - 1, size)
      if (rank .EQ. 0) then
        print *, "Enter the number of times around the ring"
        read *, num

        print *, "Process  0 sends", num, " to 1"
        call MPI_ISEND(num, 1, MPI_INTEGER, next, tag,
     $       MPI_COMM_WORLD, req1, ierr)
        call MPI_WAIT(req1, stat1, ierr)
      endif
```


Example-3, II.

```
10    continue
      call MPI_IRECV(num, 1, MPI_INTEGER, from, tag, MPI_COMM_WORLD, req2, ierr)
      call MPI_WAIT(req2, stat2, ierr)
      print *, "Process ", rank, " received ", num, " from ", from
      if (rank .EQ. 0) then
        num = num - 1
        print *, "Process  0 decremented num"
      endif
      print *, "Process", rank, " sending", num, " to", next
      call MPI_ISEND(num, 1, MPI_INTEGER, next, tag, MPI_COMM_WORLD, req1, ierr)
      call MPI_WAIT(req1, stat1, ierr)
      if (num .EQ. 0) then
        print *, "Process", rank, " exiting"
        goto 20
      endif
      goto 10

20    if (rank .EQ. 0) then
        call MPI_IRECV(num, 1, MPI_INTEGER, from, tag, MPI_COMM_WORLD, req2, ierr)
        call MPI_WAIT(req2, stat2, ierr)
      endif
      call MPI_FINALIZE(ierr)
      end
```


Example-3, I.

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

int main(int argc, char **argv){
  int num, rank, size, tag, next, from;
  MPI_Status status1, status2;
  MPI_Request req1, req2;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank( MPI_COMM_WORLD, &rank);
  MPI_Comm_size( MPI_COMM_WORLD, &size);
  tag = 201;
  next = (rank+1) % size;
  from = (rank + size - 1) % size;
  if (rank == 0) {
    printf("Enter the number of times around the ring: ");
    scanf("%d", &num);

    printf("Process %d sending %d to %d\n", rank, num, next);
    MPI_Isend(&num, 1, MPI_INT, next, tag,
          MPI_COMM_WORLD,&req1);
    MPI_Wait(&req1, &status1);
  }
```

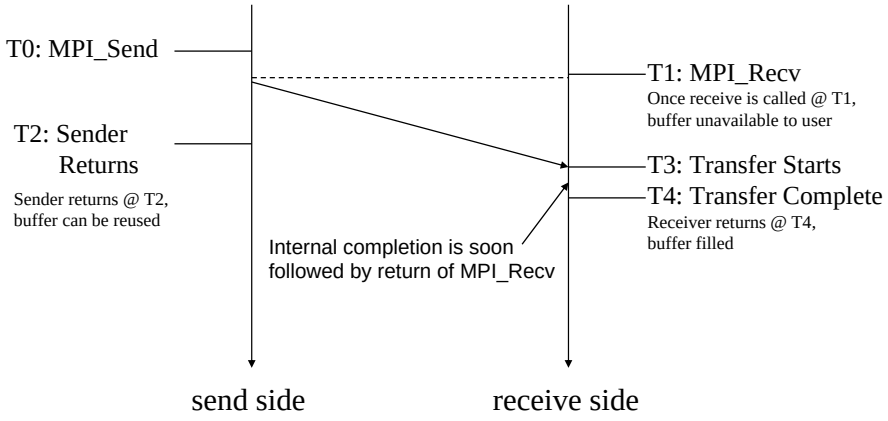
Example-3, II.

```
do {
  MPI_Irecv(&num, 1, MPI_INT, from, tag, MPI_COMM_WORLD, &req2);
  MPI_Wait(&req2, &status2);
  printf("Process %d received %d from process %d\n",rank,num,from);

  if (rank == 0) {
    num--;
    printf("Process 0 decremented number\n");
  }
  printf("Process %d sending %d to %d\n", rank, num, next);
  MPI_Isend(&num, 1, MPI_INT, next, tag, MPI_COMM_WORLD, &req1);
  MPI_Wait(&req1, &status1);
 } while (num != 0);

 if (rank == 0) {
  MPI_Irecv(&num, 1, MPI_INT, from, tag, MPI_COMM_WORLD, &req2);
  MPI_Wait(&req2, &status2);
 }

 MPI_Finalize();
 return 0;
}
```

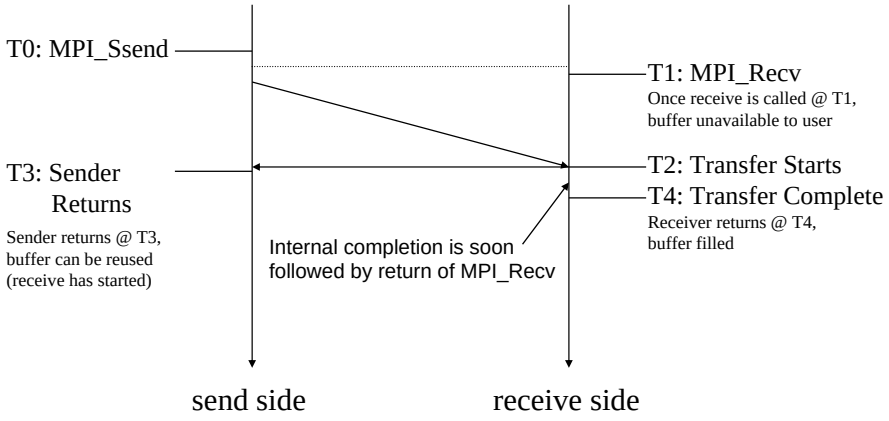

Send Modes

- Standard mode **(MPI_Send, MPI_Isend)**
  - The standard **MPI Send**, the send will not complete until the send buffer is empty
- Synchronous mode **(MPI_Ssend, MPI_Issend)**
  - The send does not complete until after a matching receive has been posted
- Buffered mode **(MPI_Bsend, MPI_Ibsend)**
  - User supplied buffer space is used for system buffering
  - The send will complete as soon as the send buffer is copied to the system buffer
- Ready mode **(MPI_Rsend, MPI_Irsend)**
  - The send will send eagerly under the assumption that a matching receive has already been posted (an erroneous program otherwise)

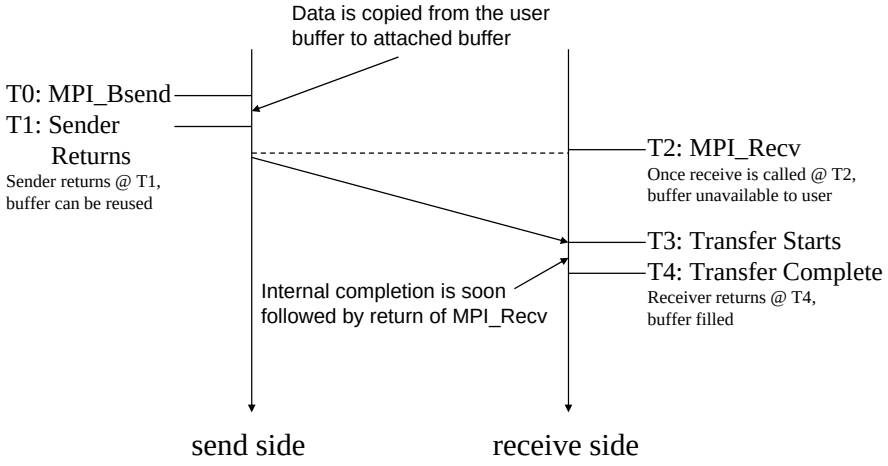
Standard Send-Receive



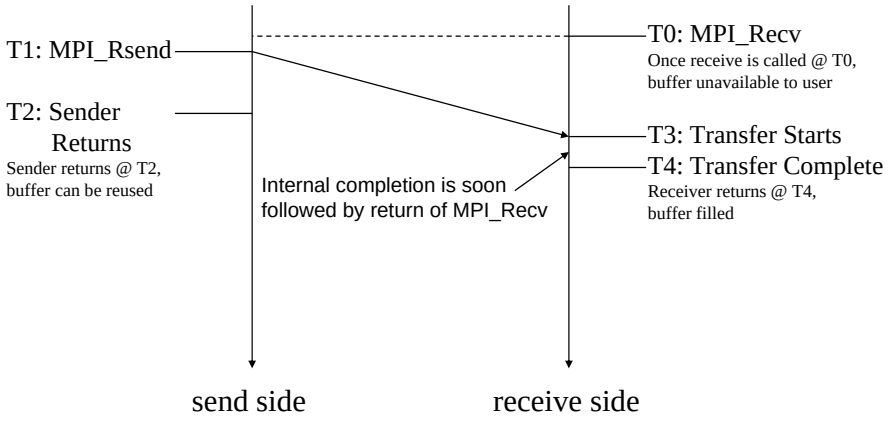
Synchronous Send-Receive



Buffered Send-Receive



Ready Send-Receive



Other Point to Point Features

- Persistent communication requests
  - Saves arguments of a communication call and reduces the overhead from subsequent calls
  - The INIT call takes the original argument list of a send or receive call and creates a corresponding communication request (e.g., **`MPI_SEND_INIT, MPI_RECV_INIT`**)
  - The START call uses the communication request to start the corresponding operation (e.g., **`MPI_START, MPI_STARTALL`**)
  - The REQUEST_FREE call frees the persistent communication request (**`MPI_REQUEST_FREE`**)
- Send-Receive operations
  - **`MPI_SENDRECV, MPI_SENDRECV_REPLACE`**
- Cleaning pending communication
  - **`MPI_CANCEL`**

Persistent Communication Example: Example 4

Example 3 using persistent communication requests

```
MPI_Recv_init(&num, 1, MPI_INT, from, tag, MPI_COMM_WORLD, &req2);
MPI_Send_init(&num, 1, MPI_INT, next, tag, MPI_COMM_WORLD, &req1);

do {
  MPI_Start(&req2);
  MPI_Wait(&req2, &status2);
  printf("Process %d received %d from process %d\n",rank,num,from);

  if (rank == 0) {
    num--;
    printf("Process 0 decremented number\n");
  }

  printf("Process %d sending %d to %d\n", rank, num, next);
  MPI_Start(&req1);
  MPI_Wait(&req1, &status1);

} while (num != 0);
```

Lab 2

- Objective: To write a function to send a message from process 0 to all other processes.
- You should assume that all processes in the communicator will call your function “at the same time.”
- The function should look something like:
  - **`user_function(void *buffer, int count, MPI_Datatype datatype, MPI_Comm comm)`**
- Process 0 should use a loop **`dest = 1 ... size-1`** **`MPI_Isend(buffer, count, datatype, dest, 0, comm, &reg[i]);`**
- **`MPI_WAITALL`** should be used to wait for the completion of all the sends.
- Processes **`1`** through **`size-1`** should use **`MPI_IRECV`** and **`MPI_WAIT`** to receive the message.

Lab 2 - Driver Program

```
program main
include 'mpif.h'

integer      ierr, rank, size, number
number=0

call MPI_Init (ierr)
call MPI_Comm_rank (MPI_COMM_WORLD, rank, ierr)
call MPI_Comm_size (MPI_COMM_WORLD, size, ierr)

if (rank.eq.0) then
   print*, “Enter the number to broadcast: ”
   read*, number
endif

call user_broadcast(number, 1, MPI_INT, MPI_COMM_WORLD);
print*, “In Process ”,rank,” the number is ”, number

call MPI_Finalize (ierr);
end
```


Lab 2 - Driver Program

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

main(int argc, char **argv)
{
   int      rank, size;
   int      number=0;

   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &rank);
   MPI_Comm_size (MPI_COMM_WORLD, &size);

   if (rank == 0) {
      printf("[%d] Enter the number to broadcast: ", rank);
      scanf ("%d", &number);
   }

   user_broadcast(&number, 1, MPI_INT, MPI_COMM_WORLD);
   printf("In Process %d the number is %d\n", rank, number) ;

   MPI_Finalize ();
}
```


Collective Communications

Collective Communications

- Communication is coordinated among a group of processes, as specified by communicator, not on all processes
- All collective operations are blocking and no message tags are used
- All processes in the communicator group must call the collective operation
- Collective and point-to-point messaging are separated by different “contexts”
- Three classes of collective operations
  - Data movement
  - Collective computation
  - Synchronization

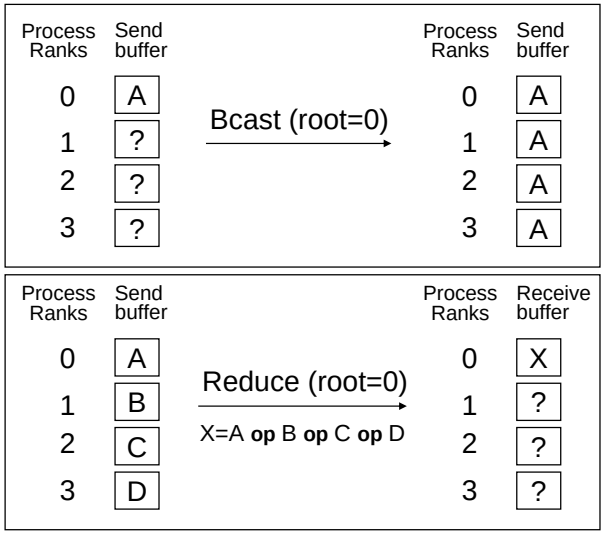
MPI Basic Collective Operations

- Two simple collective operations

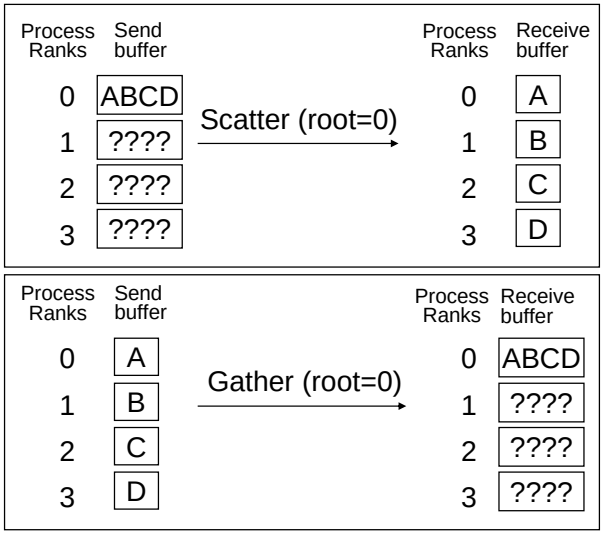
```
MPI_BCAST( start, count, datatype, root, comm )
MPI_REDUCE( start, result, count, datatype, operation,
  root, comm )
```

- The routine **MPI_BCAST** sends data from one process to all others
- The routine **MPI_REDUCE** combines data from all processes, using a specified operation, and returns the result to a single process

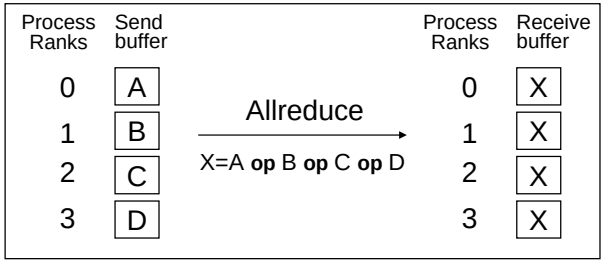
Broadcast and Reduce

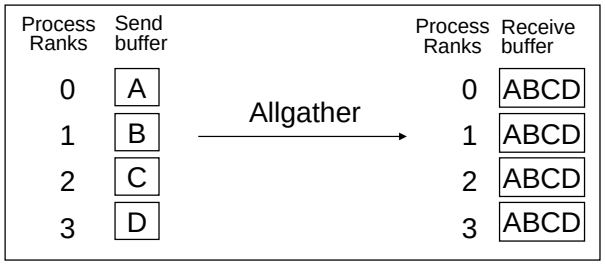


Scatter and Gather

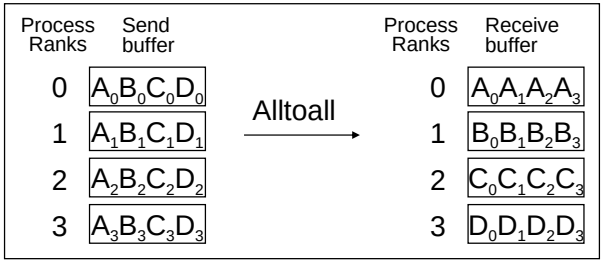


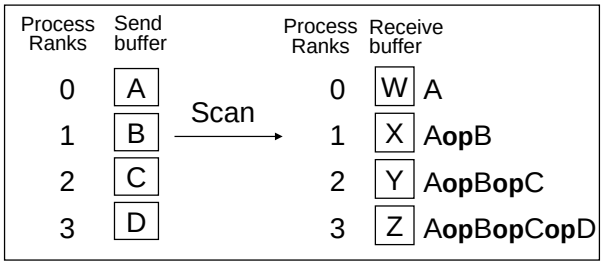
Allreduce and Allgather





Alltoall and Scan





MPI Collective Routines

- Several routines:
  MPI_ALLGATHER **MPI_ALLGATHERV** **MPI_BCAST**
  MPI_ALLTOALL **MPI_ALLTOALLV** **MPI_REDUCE**
  MPI_GATHER **MPI_GATHERV** **MPI_SCATTER**
  MPI_REDUCE_SCATTER **MPI_SCAN**
  MPI_SCATTERV **MPI_ALLREDUCE**
- **All** versions deliver results to all participating processes
- **"V"** versions allow the chunks to have different sizes
- **MPI_ALLREDUCE, MPI_REDUCE, MPI_REDUCE_SCATTER,** and **MPI_SCAN** take both built-in and user-defined combination functions

Built-In Collective Computation Operations

MPI Name	Operation
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_PROD	Product
MPI_SUM	Sum
MPI_LAND	Logical and
MPI_LOR	Logical or
MPI_LXOR	Logical exclusive or (xor)
MPI_BAND	Bitwise and
MPI_BOR	Bitwise or
MPI_BXOR	Bitwise xor
MPI_MAXLOC	Maximum value and location
MPI_MINLOC	Minimum value and location

User defined Collective Computation Operations

MPI_OP_CREATE(user_function, commute_flag, user_op)
MPI_OP_FREE(user_op)

The user_function should look like this:

```
user_function (invec, inoutvec, len, datatype)
```

The user_function should perform the following:

```
do i = 1, len
   inoutvec(i) = invec(i) op inoutvec(i)
end do
```

```
for ( i = 0; i < len; i++)
   inoutvec[i] = invec[i] op inoutvec[i];
```


Synchronization

- **`MPI_BARRIER ( comm )`**
- Function blocks until all processes in “comm” call it
- Often not needed at all in many message-passing codes
- When needed, mostly for highly asynchronous programs or ones with speculative execution

Example 5, I.

```
      program main
      include 'mpif.h'

      integer   iwidth, iheight, numpixels, i, val, my_count, ierr
      integer   rank, comm_size, sum, my_sum
      real      rms
      character recvbuf(65536), pixels(65536)

      call MPI_INIT(ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD, comm_size, ierr)

      if (rank.eq.0) then
        iheight = 256
        iwidth = 256
        numpixels = iwidth * iheight
C       Read the image
        do i = 1, numpixels
          pixels(i) = char(i)
        enddo
C       Calculate the number of pixels in each sub image
        my_count = numpixels / comm_size
      endif
```


Example 5, II.

```
C     Broadcasts my_count to all the processes
      call MPI_BCAST(my_count, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)

C     Scatter the image
      call MPI_SCATTER(pixels, my_count, MPI_CHARACTER, recvbuf,
     $     my_count, MPI_CHARACTER, 0, MPI_COMM_WORLD, ierr)

C     Take the sum of the squares of the partial image
      my_sum = 0
      do i=1,my_count
        my_sum = my_sum + ichar(recvbuf(i))*ichar(recvbuf(i))
      enddo

C     Find the global sum of the squares
      call MPI_REDUCE( my_sum, sum,  1, MPI_INTEGER, MPI_SUM, 0,
     $     MPI_COMM_WORLD, ierr)

C     rank 0 calculates the root mean square
      if (rank.eq.0) then
        rms = sqrt(real(sum)/real(numpixels))
        print *, 'RMS = ', rms
      endif
```

Example 5, III.

```
C     Rank 0 broadcasts the RMS to the other nodes
      call MPI_BCAST(rms, 1, MPI_REAL, 0, MPI_COMM_WORLD, ierr)

C     Do the contrast operation
      do i=1,my_count
        val = 2*ichar(recvbuf(i)) - rms
        if (val.lt.0) then
          recvbuf(i) = char(0)
        else if (val.gt.255) then
          recvbuf(i) = char(255)
        else
          recvbuf(i) = char(val)
        endif
      enddo

C     Gather back to root
      call MPI_GATHER(recvbuf, my_count, MPI_CHARACTER, pixels,
     $     my_count,  MPI_CHARACTER, 0, MPI_COMM_WORLD, ierr)

      call MPI_FINALIZE(ierr)
      stop
      end
```


Example 5, I.

```
#include <mpi.h>
#include <stdio.h>
#include <math.h>

int main(int argc, char *argv[])
{
  int width = 256, height = 256, rank, comm_size,
  int sum, my_sum, numpixels, my_count, i, val;
  unsigned char pixels[65536], recvbuf[65536];
  double rms;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &comm_size);

  if (rank == 0) {
    numpixels = width * height;

    /* Load the Image */
    for (i=0; i<numpixels; i++) pixels[i] = i + 1;

    /* Calculate the number of pixels in each sub image */
    my_count = numpixels / comm_size;
  }
```


Example 5, II.

```
/* Broadcasts my_count to all the processes */
 MPI_Bcast(&my_count, 1, MPI_INT, 0, MPI_COMM_WORLD);

/* Scatter the image */
 MPI_Scatter(pixels, my_count, MPI_UNSIGNED_CHAR, recvbuf, my_count,
                 MPI_UNSIGNED_CHAR, 0, MPI_COMM_WORLD);

 /* Take the sum of the squares of the partial image */
 my_sum = 0;
 for (i=0; i< my_count; i++) {
   my_sum += recvbuf[i] * recvbuf[i];
 }

 /* Find the global sum of the squares */
 MPI_Reduce( &my_sum, &sum,  1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

 /* rank 0 calculates the root mean square */
 if (rank == 0) {
   rms = sqrt ((double) sum / (double) numpixels);
   printf("RMS = %lf\n", rms);
 }
```


Example 5, III.

```
/* Rank 0 broadcasts the RMS to the other nodes */
  MPI_Bcast(&rms, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

/* Do the contrast operation */
  for (i=0; i< my_count; i++) {
    val = 2*recvbuf[i] - rms;
    if (val < 0 )
      recvbuf[i] = 0;
    else if (val > 255)
      recvbuf[i] = 255;
    else
      recvbuf[i] = val;
  }

  /* Gather back to root */
  MPI_Gather(recvbuf, my_count, MPI_UNSIGNED_CHAR, pixels, my_count,
                  MPI_UNSIGNED_CHAR, 0, MPI_COMM_WORLD);

  /* Dump the Image (only in process 0) */
  MPI_Finalize();
  return 0;
}
```

Lab 3

- Modify example 5 to handle vectors of uneven sizes by using **`MPI_SCATTERV`** and **`MPI_GATHERV`** operations instead of **`MPI_SCATTER`** and **`MPI_GATHER`** operations respectively.
- Hints:
  - Refer to the MPI function index for the description of **`MPI_SCATTERV`** and **`MPI_GATHERV`**
  - Instead of broadcasting **`my_count`**, broadcast **`numpixels`**
  - Compute the **`counts`** array and the **`displacements`** array in each process (print the arrays while debugging)
  - Replace **`MPI_SCATTER`** and **`MPI_GATHER`** functions with **`MPI_SCATTERV`** and **`MPI_GATHERV`** functions respectively.

Communicators

Communicators

- All MPI communication is based on a communicator which contains a context and a group
- Contexts define a safe communication space for message-passing
- Contexts can be viewed as system-managed tags
- Contexts allow different libraries to co-exist
- The group is just a set of processes
- Processes are always referred to by unique rank in group

Pre-Defined Communicators

- MPI-1 supports three pre-defined communicators:
  - MPI_COMM_WORLD
  - MPI_COMM_NULL
  - MPI_COMM_SELF
- Only MPI_COMM_WORLD is used for communication
- Predefined communicators are needed to “get things going” in MPI

Uses of MPI_COMM_WORLD

- Contains all processes available at the time the program was started
- Provides initial safe communication space
- Simple programs communicate with MPI_COMM_WORLD
- Complex programs duplicate and subdivide copies of MPI_COMM_WORLD
- MPI_COMM_WORLD provides the basic unit of MIMD concurrency and execution lifetime for MPI-2

Uses of MPI_COMM_NULL

- An invalid communicator
- Cannot be used as input to any operations that expect a communicator
- Used as an initial value of communicators to be defined
- Returned as a result in certain cases
- Value that communicator handles are set to when freed

Uses of MPI_COMM_SELF

- Contains only the local process
- Not normally used for communication (since only to oneself)
- Holds certain information:
  - hanging cached attributes appropriate to the process
  - providing a singleton entry for certain calls (especially MPI-2)

Duplicating a Communicator: MPI_COMM_DUP

- It is a collective operation. All processes in the original communicator must call this function
- Duplicates the communicator group, allocates a new context, and selectively duplicates cached attributes
- The resulting communicator is not an exact duplicate. It is a whole new separate communication universe with similar structure

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

MPI_COMM_DUP( COMM, NEWCOMM, IERR )
INTEGER COMM, NEWCOMM, IERR
```

Subdividing a Communicator with MPI_COMM_SPLIT

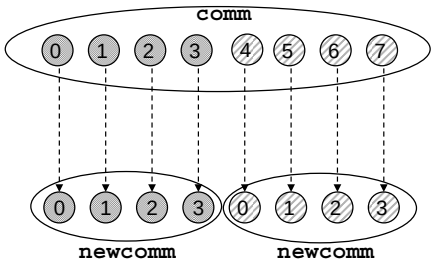
- MPI_COMM_SPLIT partitions the group associated with the given communicator into disjoint subgroups
- Each subgroup contains all processes having the same value for the argument color
- Within each subgroup, processes are ranked in the order defined by the value of the argument key, with ties broken according to their rank in old communicator

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

MPI_COMM_SPLIT( COMM, COLOR, KEY, NEWCOMM, IERR )
INTEGER COMM, COLOR, KEY, NEWCOMM, IERR
```


Subdividing a Communicator: Example 1

- To divide a communicator into two non-overlapping groups

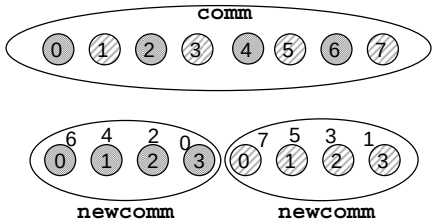
```
color = (rank < size/2) ? 0 : 1 ;
MPI_Comm_split(comm, color, 0, &newcomm) ;
```



Subdividing a Communicator: Example 2

- To divide a communicator such that
  - all processes with even ranks are in one group
  - all processes with odd ranks are in the other group
  - maintain the reverse order by rank

```
color = (rank % 2 == 0) ? 0 : 1 ;
key = size - rank ;
MPI_Comm_split(comm, color, key, &newcomm) ;
```



Subdividing a Communicator with MPI_COMM_CREATE

- Creates a new communicators having all the processes in the specified group with a new context
- The call is erroneous if all the processes do not provide the same handle
- MPI_COMM_NULL is returned to processes not in the group
- MPI_COMM_CREATE is useful if we already have a group, otherwise a group must be built using the group manipulation routines

```
int MPI_Comm_create( MPI_Comm comm, MPI_Group group, MPI_Comm
*newcomm )

MPI_COMM_CREATE( COMM, GROUP, NEWCOMM, IERR )
INTEGER COMM, GROUP, NEWCOMM, IERR
```

Group Manipulation Routines

- To obtain an existing group, use
  MPI_COMM_GROUP (comm, group) ;
- To free a group, use
  MPI_GROUP_FREE (group) ;
- A new group can be created by specifying the members to be included/excluded from an existing group using the following routines
  - MPI_GROUP_INCL: specified members are included
  - MPI_GROUP_EXCL: specified members are excluded
  - MPI_GROUP_RANGE_INCL and MPI_GROUP_RANGE_EXCL: a range of members are included or excluded
  - MPI_GROUP_UNION and MPI_GROUP_INTERSECTION: a new group is created from two existing groups
- Other routines: MPI_GROUP_COMPARE, MPI_GROUP_TRANSLATE_RANKS

Tools for Writing Libraries

- MPI is specifically designed to make it easier to write message-passing libraries
- Communicators solve tag/source wild-card problem
- Attributes provide a way to attach information to a communicator

Private Communicators

- One of the first things that a library should normally do is create a private communicator
- This allows the library to send and receive messages that are known only to the library

```
MPI_Comm_dup( old_comm, &new_comm );
```


Attributes

- Attributes are data that can be attached to one or more communicators
- Attributes are referenced by keyval. Keyvals are created with MPI_KEYVAL_CREATE
- Attributes are attached to a communicator with MPI_ATTR_PUT and their values accessed by MPI_ATTR_GET

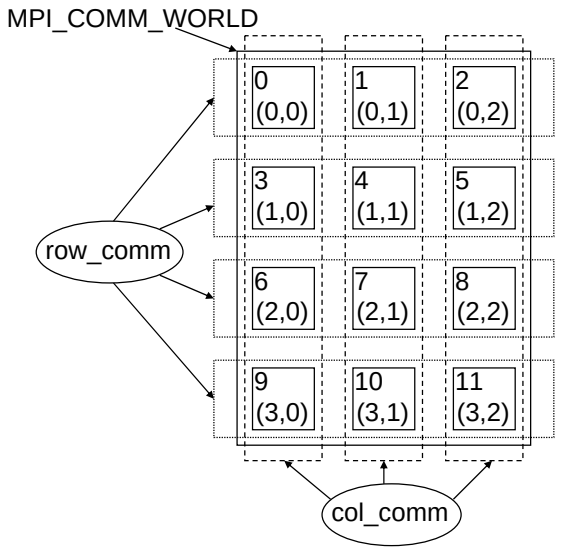
Example 6

```
      program main
      include 'mpif.h'

      integer ierr, row_comm, col_comm
      integer myrank, size, P, Q, p, q

      P = 4
      Q = 3

      call MPI_INIT(ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)

C     Determine row and column position
      p = myrank/Q
      q = mod(myrank,Q)

C     Split comm into row and column comms
      call MPI_Comm_split(MPI_COMM_WORLD, p, q, row_comm, ierr)
      call MPI_Comm_split(MPI_COMM_WORLD, q, p, col_comm, ierr)

      print*, "My coordinates are[",myrank,"] ",p,q
      call MPI_Finalize(ierr)
      stop
      end
```


Example 6

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

main(int argc, char **argv)
{
   MPI_Comm row_comm, col_comm;
   int myrank, size, P=4, Q=3, p, q;

   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &myrank);
   MPI_Comm_size (MPI_COMM_WORLD, &size);

   /* Determine row and column position */
   p = myrank / Q;
   q = myrank % Q;   /* pick a row-major mapping */

   /* Split comm into row and column comms */
   MPI_Comm_split(MPI_COMM_WORLD, p, q, &row_comm); /* color by row, rank by column */
   MPI_Comm_split(MPI_COMM_WORLD, q, p, &col_comm);  /* color by column, rank by row */

   printf("[%d]:My coordinates are (%d,%d)\n",myrank,p,q);
   MPI_Finalize();
}
```


Example 6



Lab 4, I.

- Build a function which creates a hierarchy of communicators over a grid. The function prototype is as follows:
  BUILD_GRID(X, Y, COMM_IN, COMM_GRID, COMM_ROW, COMM_COL)
  - X, Y, and COMM_IN are input arguments. The others are output arguments.
  - X, Y: The size of the GRID. X times Y should equal the size of COMM_IN or return an error.
  - COMM_GRID: Use MPI_COMM_DUP to create a duplicate of COMM_IN.
  - COMM_ROW: Will consist of processes in the same row ordered by column index.
  - COMM_COL: Will consist of processes in the same column ordered by row index.

Lab 4, II.

- The row and column that a process belongs to can be determined using the following formulae:
  - row = rank / Y
  - col = rank % Y
- The main program should do the following:
  - read the grid size X and Y in process 0
  - broadcast the grid size from process 0 to all the other processes
  - call the function BUILD_GRID from all the processes
  - use the communicators comm_row and comm_col to compute the sum of the ranks (rank in grid communicator) in the row and column communicators, respectively.
  - print the row sum and column sum in each process along with it's rank in grid communicator

Datatypes

Datatypes

- MPI datatypes have two main purposes:
  - Heterogeneity --- parallel programs between different processors
  - Noncontiguous data --- structures, vectors with non-unit stride, etc.
- Basic/primitive datatypes, corresponding to the underlying language, are predefined
- The user can construct new datatypes at run time; these are called derived datatypes
- Datatypes can be constructed recursively
- Avoids explicit packing/unpacking of data by user
- A derived datatype can be used in any communication operation instead of primitive datatype
  - MPI_SEND (buf, 1, mytype, …..)
  - MPI_RECV (buf, 1, mytype, …..)

Datatypes (continued)

- A general datatype is an opaque object that specifies
  - a sequence of basic datatypes: {type0, …, typen-1}
  - a sequence of integer (byte) displacements: {disp0, …, dispn-1}
- The sequence of basic datatypes is called the type signature of the datatype
  - typesig = {type0, …, typen-1}
- The sequence of pairs of type signature and displacement (i.e., (typei, dispi)) is called a type map
  - typemap = {(type0, disp0), …, (typen-1, dispn-1)}
- The typesig and typemap provide the information required to assemble data when a general datatype is used in a communication operation

Datatypes in MPI

- Elementary: Language-defined types
  - MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION, etc.
- Vector: Separated by constant “stride”
  - MPI_TYPE_VECTOR
- Contiguous: Vector with stride of one
  - MPI_TYPE_CONTIGUOUS
- Hvector: Vector, with stride in bytes
  - MPI_TYPE_HVECTOR
- Indexed: Array of indices (for scatter/gather)
  - MPI_TYPE_INDEXED
- Hindexed: Indexed, with indices in bytes
  - MPI_TYPE_HINDEXED
- Struct: General mixed types (for C structs etc.)
  - MPI_TYPE_STRUCT

Primitive Datatypes in MPI (C)

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

Primitive Datatypes in MPI (FORTRAN)

MPI FORTRAN	FORTRAN datatypes
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER
MPI_BYTE	
MPI_PACKED	

Example: Building Structures

```
struct {
   char     display[50]; /* Name of display */
   int      maxiter;     /* max # of iterations */
   double   xmin, ymin;  /* lower left corner of rectangle */
   double   xmax, ymax;  /* upper right corner */
   int      width;       /* of display in pixels */
   int      height;      /* of display in pixels */
} cmdline;

/* set up 4 blocks */
int         blockcounts[4] = {50,1,4,2};
MPI_Datatype types[4]={MPI_CHAR, MPI_INT, MPI_DOUBLE, MPI_INT};
MPI_Aint     displs[4];
MPI_Datatype cmdtype;

/* initialize types and displs with addresses of items */
MPI_Address(&cmdline.display, &displs[0]);
MPI_Address(&cmdline.maxiter, &displs[1]);
MPI_Address(&cmdline.xmin,    &displs[2]);
MPI_Address(&cmdline.width,   &displs[3]);
for (i = 3; i >= 0; i--)
   displs[i] -= displs[0];

MPI_Type_struct(4, blockcounts, displs, types, &cmdtype);
MPI_Type_commit(&cmdtype);
```

Example: Building Structures

```
character         display(50)
integer           maxiter
double precision  xmin, ymin
double precision  xmax, ymax
integer           width
integer           height
common /cmdline/ display,maxiter,xmin,ymin,xmax,ymax,width,height

integer blockcounts(4), types(4), displs(4), cmdtype
data    blockcounts/50,1,4,2/
data    types/MPI_CHARACTER,MPI_INTEGER,MPI_DOUBLE_PRECISION,
$       MPI_INTEGER/

call MPI_Address(display, displs(1), ierr)
call MPI_Address(maxiter, displs(2), ierr)
call MPI_Address(xmin, displs(3), ierr)
call MPI_Address(width, displs(4), ierr)
do i = 4, 1, -1
  displs(i) = displs(i) - displs(1)
end do

call MPI_Type_struct(4, blockcounts, displs, types, cmdtype, ierr)
call MPI_Type_commit(cmdtype, ierr)
```


Structures

- Structures are described by
  - number of blocks
  - array of number of elements (array_of_len)
  - array of displacements or locations (array_of_displs)
  - array of datatypes (array_of_types)

```
MPI_TYPE_STRUCT(count,
                array_of_len,
                array_of_displs,
                array_of_types,
                newtype);
```


Example: Building Vectors

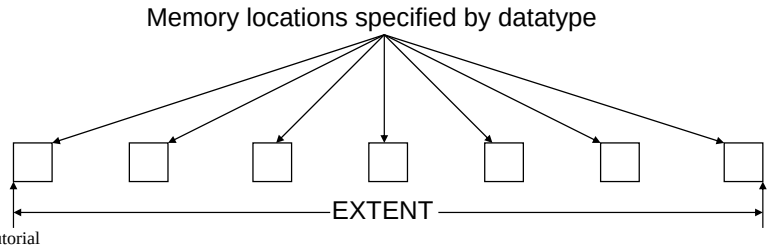
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31	32	33	34	35
36	37	38	39	40	41	42
43	44	45	46	47	48	49

- To specify this column (in row order), use
  MPI_TYPE_VECTOR(count, blocklen, stride, oldtype, newtype)
  MPI_TYPE_COMMIT(newtype)

- The exact code for this is
  MPI_TYPE_VECTOR(7, 1, 7, MPI_DOUBLE, newtype);
  MPI_TYPE_COMMIT(newtype);

Extents

- The extent of a datatype is (normally) the distance between the first and last member.
- We can set an artificial extent by using MPI_UB and MPI_LB in MPI_TYPE_STRUCT
- The routine MPI_TYPE_EXTENT must be used to obtain the size of a datatype (not sizeof in C) since datatypes are opaque objects
  - MPI_TYPE_EXTENT (datatype, extent)



Vectors Revisited

- To create a datatype for an arbitrary number of elements in a column of an array stored in row-major format, use

```
int displs[2], sizeofdouble;
int blens[2] = {1, 1};
MPI_Datatype types[2] = {MPI_DOUBLE, MPI_UB};
MPI_Datatype coltype;
MPI_Type_extent(MPI_DOUBLE, &sizeofdouble);
displs[0] = 0;
displs[1] = number_in_columns * sizeofdouble;

MPI_Type_struct(2, blens, displs, types, &coltype);
MPI_Type_commit(&coltype);
```

- To send **`n`** elements, we can use

```
MPI_Send(buf, n, coltype, ...);
```

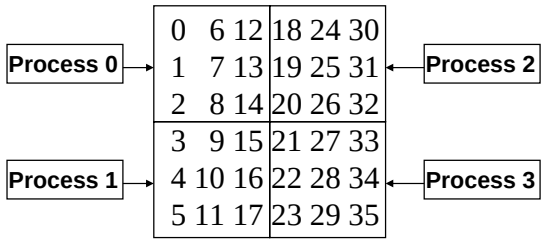

Structures Revisited

- When sending an array of structures, it is important to ensure that MPI and the compiler have the same value for the size of each structure
- Most portable way to do this is to use **MPI_UB** in the structure definition for the end of the structure. In the previous example, this would be:

```
MPI_Datatype types[5] = {MPI_CHAR, MPI_INT, MPI_DOUBLE,
                         MPI_INT, MPI_UB};

/* initialize types and displs */
MPI_Address(&cmdline.display, &displs[0]);
MPI_Address(&cmdline.maxiter, &displs[1]);
MPI_Address(&cmdline.xmin,    &displs[2]);
MPI_Address(&cmdline.width,   &displs[3]);
MPI_Address(&cmdline[1],      &displs[4]);
for (i = 4; i >= 0; i--)
  displs[i] -= displs[0];

MPI_Type_struct(5, blockcounts, displs, types, &cmdtype);
MPI_Type_commit(&cmdtype);
```


Interleaving Data

- We can interleave data by moving the upper bound value inside the data
- To distribute a matrix among 4 processes, we can create a block datatype and use **MPI_SCATTERV**



NOTE: Scatterv does the following for all processes (i = 0 to size-1)

```
send(buf+displs(i)*extent(sendtype), sendcounts(i),
     sendtype,.....)
```


An Interleaved Datatype - C Example

- Define a vector datatype

```
MPI_Type_vector (3, 3, 6, MPI_DOUBLE, &vectype);
```

- Define a block whose extent is just one entry

```
int sizeofdouble;
int blens[2] = {1, 1};
MPI_Type_extent (MPI_DOUBLE, &sizeofdouble);
int indices[2] = {0, sizeofdouble};
MPI_Datatype types[2] = {vectype, MPI_UB};

MPI_Type_struct (2, blens, indices, types, &block);
MPI_Type_commit (&block);

int len[4] = {1,1,1,1};
int displs[4] = {0,3,18,21};
MPI_Scatterv(sendbuf, len, displs, block,
        recvbuf, 9, MPI_DOUBLE, 0, comm);
```

An Interleaved Datatype - C Example

- Define a vector datatype
  MPI_Type_vector (3, 3, 6, MPI_DOUBLE, &vectype);
- Define a block whose extent is just one entry

```
int sizeofdouble;
int blens[2] = {1, 1};
MPI_Type_extent (MPI_DOUBLE, &sizeofdouble);
int indices[2] = {0, 3*sizeofdouble};
MPI_Datatype types[2] = {vectype, MPI_UB};

MPI_Type_struct (2, blens, indices, types, &block);
MPI_Type_commit (&block);

int len[4] = {1,1,1,1};
int displs[4] = {0,1,6,7};
MPI_Scatterv(sendbuf, len, displs, block,
        recvbuf, 9, MPI_DOUBLE, 0, comm);
```


Example 7, I.

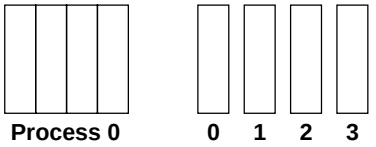
```
program main
include 'mpif.h'

integer N, M
parameter (N=12, M=48)

real a(N,N), row(M)
integer rank, i, j, size, bsize,blens(2), displ(2), sizeofreal
integer types(2), temptype, recvtype

call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)

bsize = N/size
do i = 1, M
  row(i) = rank*M + i
enddo

if (rank.eq.0) then
  do i = 1, N
    do j = 1, N
      a(i,j) = 0.0
    enddo
  enddo
endif
```


Example 7, II.

```
call MPI_TYPE_VECTOR(N,bsize,N,MPI_REAL,temptype,ierr)
blens(1) = 1
blens(2) = 1
call MPI_TYPE_EXTENT(MPI_REAL, sizeofreal, ierr)
displ(1) = 0
displ(2) = bsize*sizeofreal
types(1) = temptype
types(2) = MPI_UB
call MPI_TYPE_STRUCT(2, blens, displ, types, recvtype, ierr)
call MPI_TYPE_COMMIT(recvtype, ierr)

call MPI_GATHER(row, M, MPI_REAL, a, 1, recvtype, 0,
$    MPI_COMM_WORLD, ierr)

if (rank.eq.0) then
  do i = 1, N
    print *, (a(i,j), j=1,N)
  enddo
endif

call MPI_FINALIZE(ierr)
end
```


Example 7, I.

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

#define N    12
#define M    36

main(int argc, char **argv)
{
  float a[N][N], column[M];
  int   rank, i, j, size, bsize, blens[2];
  MPI_Aint    displ[2];
  MPI_Datatype types[2], temptype, recvtype;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  bsize = N/size;

  for (i=0; i < M; i++)
    column[i] = rank*M + i + 1.0;

  if (rank == 0)
    for (i=0; i < N; i++)
      for (j=0; j < N; j++)
        a[i][j] = 0.0;
```

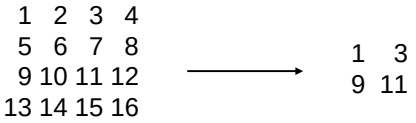


Example 7, II.

```
MPI_Type_vector(N,bsize,N,MPI_FLOAT,&temptype);
blens[0] = 1;
blens[1] = 1;
displ[0] = 0;
displ[1] = bsize*sizeof(float);
types[0] = temptype;
types[1] = MPI_UB;
MPI_Type_struct (2, blens, displ, types, &recvtype);
MPI_Type_commit(&recvtype);

MPI_Gather(column, M, MPI_FLOAT, a, 1, recvtype, 0, MPI_COMM_WORLD);

if (rank == 0) {
  for (i=0; i < N; i++) {
    for (j=0; j < N; j++)
      printf("%f ", a[i][j]);
    printf("\n");
  }
}

MPI_Finalize();
}
```


Lab 5

- Create a datatype called submatrix that consists of elements in alternate rows and alternate columns of the given original matrix.
- Use MPI_SENDRECV to send the submatrix from a process to itself and print the results. To test this program you can run the program on just one processor.
- First build a newvector type that has alternate elements in a row (column, for Fortran programmers)
- Use the newvector type to build the submatrix type



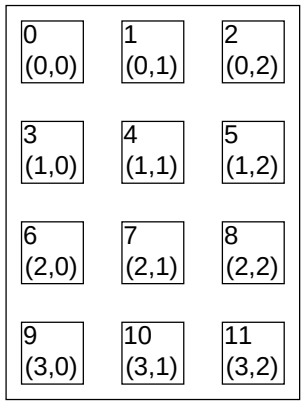
Topologies

Topologies

- MPI provides routines to provide structure to collections of processes
- Topologies provide a mapping from application to physical description of processors
- These routines allow the MPI implementation to provide an ordering of processes in a topology that makes logical neighbors close in the physical interconnect (e.g., grey code for hypercubes)
- Provides routines that answer the question: Who are my neighbors?

Cartesian Topologies

4 x 3 cartesian grid



Defining a Cartesian Topology

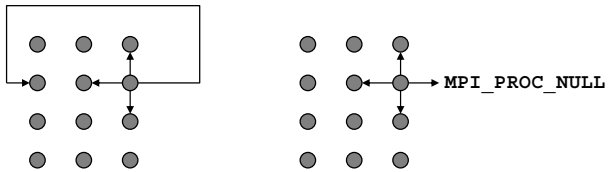
- The routine MPI_CART_CREATE creates a Cartesian decomposition of the processes

MPI_CART_CREATE(MPI_COMM_WORLD, ndim, dims, periods, reorder, comm2d)

  - ndim - no. of cartesian dimensions
  - dims - an array of size ndims to specify no. of processes in each dimension
  - periods - an array of size ndims to specify the periodicity in each dimension
  - reorder - flag to specify ordering of ranks for better performance
  - comm2d - new communicator with the cartesian information cached

The Periods Argument

- In the non-periodic case, a neighbor may not exist, which is indicated by a rank of MPI_PROC_NULL
- This rank may be used in send and receive calls in MPI
- The action in both cases is as if the call was not made



Defining a Cartesian Topology

```
ndim      = 2
dims(1)   = 4
dims(2)   = 3
periods(1) = .false.
periods(2) = .false.
reorder   = .true.
call MPI_CART_CREATE(MPI_COMM_WORLD, ndim, dims,
$                    periods, reorder, comm2d, ierr)
```

```
ndim      = 2;
dims[0]   = 4;
dims[1]   = 3;
periods[0] = 0;
periods[1] = 0;
reorder   = 1;
MPI_CART_CREATE(MPI_COMM_WORLD, ndim, dims,
                periods, reorder, &comm2d);
```

Finding Neighbors

- MPI_CART_CREATE creates a new communicator with the same processes as the input communicator, but with the specified topology
- The question, Who are my neighbors, can be answered with MPI_CART_SHIFT
- The values returned are the ranks, in the communicator comm2d, of the neighbors shifted by +/- 1 in the two dimensions
- The values returned can be used in a MPI_SENDRECV call as the ranks of source and destination

```
MPI_CART_SHIFT(comm, direction,  displacement,
                     src_rank, dest_rank)
MPI_CART_SHIFT(comm2d, 0,  1, nbrtop, nbrbottom)
MPI_CART_SHIFT(comm2d, 1,  1, nbrleft, nbrright)
```


Partitioning a Cartesian Topology

- A cartesian topology can be divided using MPI_CART_SUB on the communicator returned by MPI_CART_CREATE
- MPI_CART_SUB is closely related to MPI_COMM_SPLIT
- To create a communicator with all processes in dimension-1, use

```
remain_dims(1) = .false.
remain_dims(2) = .true.
MPI_Cart_sub(comm2d, remain_dims, comm_row, ierr)
```

```
remain_dims[0] = 0;
remain_dims[1] = 1;
MPI_Cart_sub(comm2d, remain_dims, &comm_row);
```


Partitioning a Cartesian Topology (continued)

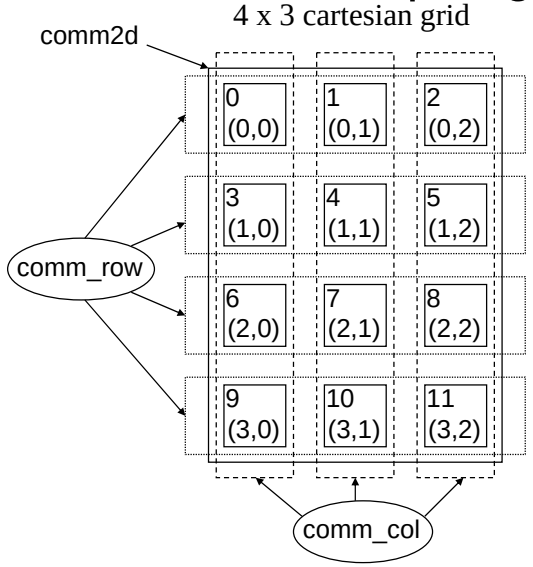
- To create a communicator with all processes in dimension-0, use

```
remain_dims(1) = .true.
remain_dims(2) = .false.
MPI_Cart_sub(comm2d, remain_dims, comm_col, ierr)
```

```
remain_dims[0] = 1;
remain_dims[1] = 0;
MPI_Cart_sub(comm2d, remain_dims, &comm_col);
```


Cartesian Topologies

4 x 3 cartesian grid



Other Topology Routines

- MPI_CART_COORDS: Returns the cartesian coordinates of the calling process given the rank
- MPI_CART_RANK: Translates the cartesian coordinates to process ranks as they are used by the point-to-point routines
- MPI_DIMS_CREATE: Returns a good choice for the decomposition of the processors
- MPI_CART_GET: Returns the cartesian topology information that was associated with the communicator
- MPI_GRAPH_CREATE: allows the creation of a general graph topology
- Several routines similar to cartesian topology routines for general graph topology

Example 8, I.

```
      program topology
      include "mpif.h"

      integer NDIMS
      parameter (NDIMS = 2)
      integer dims(NDIMS), local(NDIMS)
      logical periods(NDIMS), reorder, remain_dims(2)
      integer comm2d, row_comm, col_comm, rowsize, colsize
      integer nprow, npcol, myrow, mycol, numnodes, ierr
      integer left, right, top, bottom, sum_row, sum_col

      call MPI_INIT( ierr )
      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, numnodes, ierr )

      dims(1) = 0
      dims(2) = 0
      call MPI_DIMS_CREATE( numnodes, NDIMS, dims, ierr )
      nprow = dims(1)
      npcol = dims(2)

      periods(1) = .TRUE.
      periods(2) = .TRUE.
      reorder = .TRUE.

      call MPI_CART_CREATE( MPI_COMM_WORLD, NDIMS, dims, periods,
     $                      reorder, comm2d, ierr )
```


Example 8, II.

```
call MPI_CART_COORDS( comm2d, myrank, DIMS, local, ierr )

myrow = local(1)
mycol = local(2)

remain_dims(1) = .FALSE.
remain_dims(2) = .TRUE.
call MPI_CART_SUB( comm2d, remain_dims, row_comm, ierr )

remain_dims(1) = .TRUE.
remain_dims(2) = .FALSE.
call MPI_CART_SUB( comm2d, remain_dims, col_comm, ierr )

call MPI_Comm_size(row_comm, rowsize,ierr)
call MPI_Comm_size(col_comm, colsize,ierr)
if (myrank.eq.0) print*, rowsize = ',rowsize,' colsize = ',colsize

call MPI_CART_SHIFT(comm2d, 1, 1, left, right, ierr)
call MPI_CART_SHIFT(comm2d, 0, 1, top, bottom, ierr)

print *,'myrank[',myrank,'] (p,q) = (',myrow,mycol,' )'
print *,'myrank[',myrank,'] left ',left,' right ',right
print *,'myrank[',myrank,'] top ',top,' bottom ',bottom

call MPI_Finalize(ierr)
end
```


Example 8, I

```
#include <mpi.h>
#include <stdio.h>
typedef enum{FALSE, TRUE} BOOLEAN;
#define N_DIMS 2

main(int argc, char **argv)
{
   MPI_Comm comm_2d, row_comm, col_comm;
   int myrank, size, P, Q, p, q, reorder, left, right, bottom, top, rowsize, colsize;
   int dims[N_DIMS],        /* number of dimensions */
      local[N_DIMS],        /* local row and column positions */
      period[N_DIMS],       /* aperiodic flags */
      remain_dims[N_DIMS];  /* sub-dimension computation flags */

   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &myrank);
   MPI_Comm_size (MPI_COMM_WORLD, &size);

   /* Generate a new communicator with virtual topology */
   dims[0] = dims[1] = 0;
   MPI_Dims_create( size, N_DIMS, dims );
   P = dims[0];
   Q = dims[1];
   reorder = TRUE;
   period[0] = period[1] = TRUE;
   MPI_Cart_create(MPI_COMM_WORLD, N_DIMS, dims, period, reorder, &comm_2d);
```

Example 8, II.

```
/* Determine the position in the grid and split comm2d into row and col comms */
  MPI_Cart_coords(comm_2d, myrank, N_DIMS, local);
  p = local[0];    q = local[1];

  /* Get row and column communicators using cartesian sub-topology */
  remain_dims[0] = FALSE;
  remain_dims[1] = TRUE;
  MPI_Cart_sub(comm_2d, remain_dims, &row_comm);

  remain_dims[0] = TRUE;
  remain_dims[1] = FALSE;
  MPI_Cart_sub(comm_2d, remain_dims, &col_comm);

  MPI_Comm_size(row_comm, &rowsize);
  MPI_Comm_size(col_comm, &colsize);

  MPI_Cart_shift(comm_2d, 1, 1, &left, &right);
  MPI_Cart_shift(comm_2d, 0, 1, &top, &bottom);

  printf("(%d,%d)[%d] left = %d right = %d top = %d bottom = %d\n",
         p, q, myrank, left, right, top, bottom);
  if (myrank == 0)
     printf("Grid size = %dX%d, rowsize = %d colsize = %d \n", P, Q, rowsize, colsize);

  MPI_Finalize();
}
```


Example 8 - Output

- Grid size = 4X3
- rowsize = 3 colsize = 4
- (0,0)[0] left = 2 right = 1 top = 9 bottom = 3
- (1,0)[3] left = 5 right = 4 top = 0 bottom = 6
- (0,1)[1] left = 0 right = 2 top = 10 bottom = 4
- (3,0)[9] left = 11 right = 10 top = 6 bottom = 0
- (2,1)[7] left = 6 right = 8 top = 4 bottom = 10
- (2,0)[6] left = 8 right = 7 top = 3 bottom = 9
- (2,2)[8] left = 7 right = 6 top = 5 bottom = 11
- (3,2)[11] left = 10 right = 9 top = 8 bottom = 2
- (1,2)[5] left = 4 right = 3 top = 2 bottom = 8
- (3,1)[10] left = 9 right = 11 top = 7 bottom = 1
- (0,2)[2] left = 1 right = 0 top = 11 bottom = 5
- (1,1)[4] left = 3 right = 5 top = 1 bottom = 7

Lab 6

- Repeat Lab 4 using topology functions with the period flag set TRUE
- Shift the row_sum computed along the column communicator using MPI_SENDRECV
- Similarly shift the col_sum computed along the row communicator using MPI_SENDRECV
- Display the new row_sum and col_sum along with the cartesian coordinates
- Use MPI_CART_SHIFT to determine the neighbors along the row and column communicators

Inter-communicators

Inter-communicators

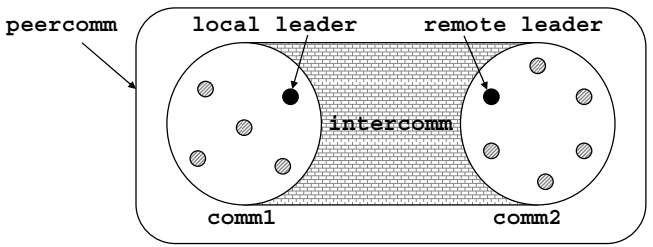
- Intra-communication: communication between processes that are members of the same group
- Inter-communication: communication between processes in different groups (say, local group and remote group)
- Both inter- and intra-communication have the same syntax for point-to-point communication
- Inter-communicators can be used only for point-to-point communication (no collective and topology operations with inter-communicators)
- A target process is specified using its rank in the remote group
- Inter-communication is guaranteed not to conflict with any other communication that uses a different communicator

Inter-communicator Accessor Routines

- To determine whether a communicator is an intra-communicator or an inter-communicator
  - MPI_COMM_TEST_INTER(comm, flag)
    flag = true, if comm is an inter-communicator
    flag = false, otherwise
- Routines that provide the local group information when the communicator used is an inter-communicator
  - MPI_COMM_SIZE, MPI_COMM_GROUP, MPI_COMM_RANK
- Routines that provide the remote group information for inter-communicators
  - MPI_COMM_REMOTE_SIZE, MPI_COMM_REMOTE_GROUP

Inter-communicator Create

- MPI_INTERCOMM_CREATE creates an inter-communicator by binding two intra-communicators
  - MPI_INTERCOMM_CREATE(local_comm, local_leader,
    peer_comm, remote_leader,
    tag, intercomm)

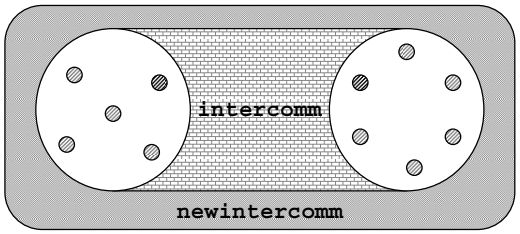


Inter-communicator Create (continued)

- Both the local and remote leaders should
  - belong to a peer communicator
  - know the rank of the other leader in the peer communicator
- Members of each group should know the rank of their leader
- An inter-communicator create operation involves
  - collective com munication among processes in local group
  - collective communication among processes in remote group
  - point-to-point communication between local and remote leaders
- To exchange data between the local and remote groups after the inter-communicator is created, use

```
MPI_SEND(..., 0, intercomm)
MPI_RECV(buf, ..., 0, intercomm);
MPI_BCAST(buf, ..., localcomm);
```

Inter-communicator Merge

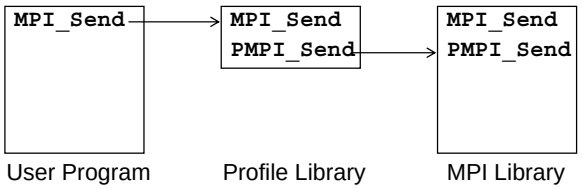


- MPI_INTERCOMM_MERGE creates an intra-communicator by merging the local and remote groups of an inter-communicator
  - MPI_INTERCOMM_MERGE(intercomm, high, newintracomm)
- The process groups are ordered based on the value of **high**
- All processes in one group should have the same value for **high**

Profiling Interface

Profiling Interface

- The objective of the MPI profiling interface is to assist profiling tools to interface their code to different MPI implementations
- Profiling tools can obtain performance information without access to the underlying MPI implementation
- All MPI routines have two entry points: MPI_.. and PMPI_..
- Users can use the profiling interface without modification to the source code by linking with a profiling library
- A log file can be generated by replacing -lmpi with -llmpi -lpmpi -lm



Timing MPI Programs

- MPI_WTIME returns a floating-point number of seconds, representing elapsed wall-clock time since some time in the past

  double MPI_Wtime(void)
  DOUBLE PRECISION MPI_WTIME( )

- MPI_WTICK returns the resolution of MPI_WTIME in seconds. It returns, as a double precision value, the number of seconds between successive clock ticks.

  double MPI_Wtick(void)
  DOUBLE PRECISION MPI_WTICK( )

Output Servers

- Portable, moderate- to high-performance output capability for distributed memory programs
- Master-slave approach
  - Reserve one “master” processor for I/O
  - Waits to receive messages from workers
  - Instead of printing, workers send data to master
  - Master receives messages and assembles single, global output file
- MPI-2 I/O functions

Case Study

2-D Laplace solver

- Mathematical Formulation
- Numerical Method
- Implementation
  - Topologies for structured meshes
  - MPI datatypes
  - Optimizing point-to-point communication

Mathematical Formulation

- The poisson equation can be written as

$$\nabla^2 u = f(x, y) \text{ in the interior } (1)$$

$$u(x, y) = g(x, y) \text{ on the boundary } (2)$$

- Using a 5-point finite difference Laplace scheme eq. (1) can be discretized as

$$\frac{u_{i-1,j} + u_{i,j+1} + u_{i,j-1} + u_{i+1,j} - 4u_{i,j}}{h^2} = f_{i,j}, \mathrm{h} = \Delta\mathrm{x} = \Delta\mathrm{y}$$

Numerical Method

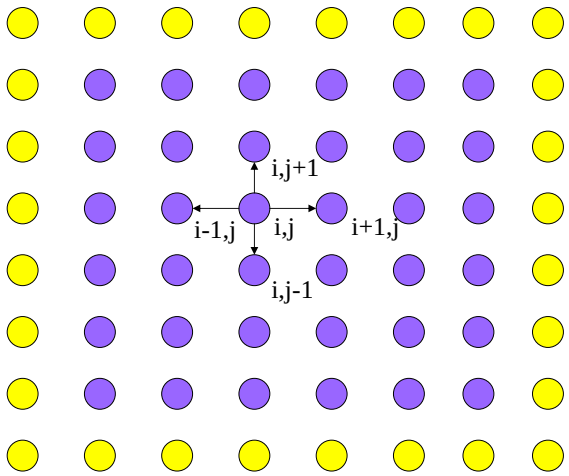
- The Poisson equation can be solved using a Jacobi iteration

$$u^{k+1}_{i,j} = \frac{1}{4}(u^{k}_{i-1,j} + u^{k}_{i,j+1} + u^{k}_{i,j-1} + u^{k}_{i+1,j} - h^2 f_{i,j})$$

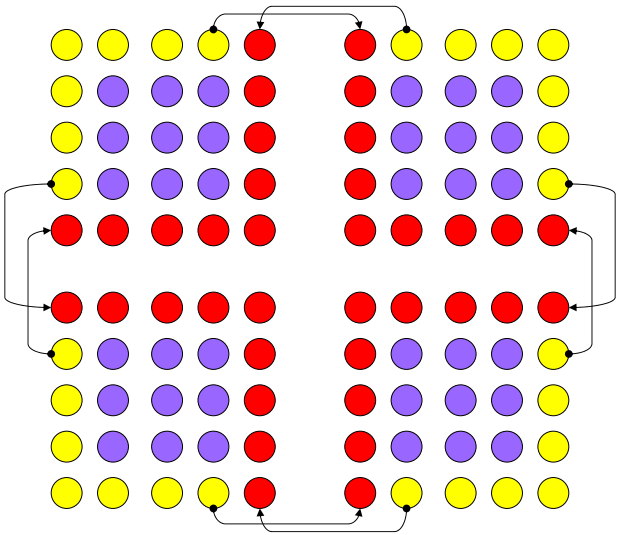
- A simple algorithm to solve the poisson equation is shown below

```
Initialize right hand side
Setup an initial solution guess
do
    for all the grid points
        compute u^{k+1}_{i,j}
    compute norm
until convergence
printout solution
```


Implementation Details



Parallel Implementation



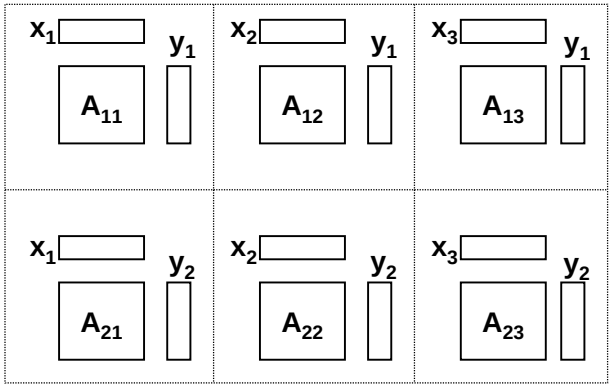
Parallel Algorithm

- Initialize right hand side
- Setup an initial solution guess
- do
- for all the grid points
- compute $u^{k+1}_{i,j}$
- exchange data across process boundaries
- compute norm
- until convergence
- printout solution

Extra Slides

Parallel Linear Algebra

Topology Functions in Linear Algebra Libraries



Example 8, I.

```
void create_2dgrid(MPI_Comm comm_in, MPI_Comm *comm_2d, MPI_Comm *row_comm,
                   MPI_Comm *col_comm) {
  int dims[N_DIMS],         /* number of dimensions */
      period[N_DIMS],       /* aperiodic flags */
      remain_dims[N_DIMS];  /* sub-dimension computation flags */
  int size, reorder;
  MPI_Comm_size (comm_in, &size);
  /* Generate a new communicator with virtual topology */
  dims[0] = dims[1] = 0;
  MPI_Dims_create( size, N_DIMS, dims );
  reorder = TRUE;
  period[0] = period[1] = TRUE;
  MPI_Cart_create(comm_in, N_DIMS, dims, period, reorder, comm_2d);

  /* Get row and column communicators using cartesian sub-topology */
  remain_dims[0] = FALSE;
  remain_dims[1] = TRUE;
  MPI_Cart_sub(*comm_2d, remain_dims, row_comm);

  remain_dims[0] = TRUE;
  remain_dims[1] = FALSE;
  MPI_Cart_sub(*comm_2d, remain_dims, col_comm);
}
```


Example 8, II.

```
/* Compute [z <- alpha * A * x + beta * y] */
void pdgemv(double alpha, double a[M][N], double x[N], double beta,
            double y[M], double z[M], MPI_Comm comm) {
  int i, j;
  double u[M];

  /* Compute part of [A * x] */
  for (i = 0; i < M; i++) {
    u[i] = 0.0;
    for (j = 0; j < N; j++)
      u[i] += a[i][j]*x[j];
  }

  /* Obtain complete [A * x] */
  MPI_Allreduce(u, z, M, MPI_DOUBLE, MPI_SUM, comm);

  /* Update z */
  for (i = 0; i < M; i++)
    z[i] = alpha * z[i] + beta * y[i];
}
```

Example 8, I

```
      subroutine create_2dgrid(commin, comm2d, row_comm, col_comm)
      include 'mpif.h'
      integer commin, comm2d, row_comm, col_comm
      integer NDIMS
      parameter (NDIMS = 2)
      integer dims(NDIMS), numnodes, ierr
      logical periods(NDIMS), reorder, remain_dims(2)
      call MPI_COMM_SIZE( commin, numnodes, ierr )
      dims(1) = 0
      dims(2) = 0
      call MPI_DIMS_CREATE( numnodes, NDIMS, dims, ierr )
C     Create a cartesian grid
      periods(1) = .TRUE.
      periods(2) = .TRUE.
      reorder = .TRUE.
      call MPI_CART_CREATE(commin, NDIMS, dims, periods, reorder, comm2d, ierr )
C     Divide the 2-D cartesian grid into row and column communicators
      remain_dims(1) = .FALSE.
      remain_dims(2) = .TRUE.
      call MPI_CART_SUB( comm2d, remain_dims, row_comm, ierr )
      remain_dims(1) = .TRUE.
      remain_dims(2) = .FALSE.
      call MPI_CART_SUB( comm2d, remain_dims, col_comm, ierr )
      return
      end
```


Example 8, II.

```
      subroutine pdgemv(alpha, a, x, beta, y, z, comm)
      include 'mpif.h'
      integer M, N
      parameter (M = 2, N = 3)
      double precision alpha, a(M,N), x(N), beta, y(M), z(M)
      integer comm
      integer i, j, ierr
      double precision u(M)
C     Compute part of A * x
      do i = 1, M
        u(i) = 0.0
        do j = 1, N
          u(i) = u(i) + a(i,j)*x(j)
        enddo
      enddo
C     Obtain complete A * x
      call MPI_Allreduce(u, z, M, MPI_DOUBLE_PRECISION, MPI_SUM,  comm, ierr)
C     Update z
      do i = 1, M
        z(i) = alpha * z(i) + beta * y(i)
      enddo
      return
      end
```


Lab 6

- Compile and run example 8 for different grid sizes.
- Modify example 8 to determine the maximum value in vector z
- Perform the operation
  $A = \alpha * z * x^T + A$ (rank-1 update)
  where α is the maximum value in vector z determined above
- Display the matrix A

Solution: Lab 6, I.

```
/* Create a 2-D cartesian topology */
create_2dgrid(MPI_COMM_WORLD, &comm_2d, &row_comm, &col_comm);

/* Determine the position in the grid */
MPI_Cart_coords(comm_2d, myrank, N_DIMS, local);
p = local[0];     q = local[1];

/* Initialize the matrix A and vectors x and y */
init_data(a, x, y, p, q);

/* Compute [z <- alpha * A * x + beta * y] */
pdgemv(alpha, a, x, beta, y, z, row_comm);

/* Obtain the maximum value in vector z. First obtain local max */
max = z[0];
for (i = 1; i < M; i++)
  if (z[i] > max)  max = z[i];

/* Now find global max */
MPI_Allreduce(&max, &globalmax, 1, MPI_DOUBLE, MPI_MAX, col_comm);

/* Compute [A <- alpha * z * x^T + A] */
pdger(globalmax, z, x, a);
```

Solution: Lab 6, II.

```
/* Compute [A <- alpha * z * x^T + A] */
void pdger(double alpha, double z[M], double x[N], double a[M][N]) {
   int i, j;

   for (i = 0; i < M; i++)
     for ( j = 0; j < N; j++)
       a[i][j] += alpha * z[i]*x[j];
}
```


Solution: Lab 6, I.

```
C     Create a 2-D cartesian topology
      call create_2dgrid(MPI_COMM_WORLD, comm2d, row_comm, col_comm)
C     Obtain local co-ordinates
      call MPI_CART_COORDS( comm2d, myrank, NDIMS, local, ierr )
      myrow = local(1)
      mycol = local(2)
C     Initialize matrix A and vectors x and y
      call init_data(a, x, y, myrow, mycol)

      alpha = 1.0
      beta = -1.0
C     Compute z <- alpha * A * x + beta * y
      call pdgemv(alpha, a, x, beta, y, z, row_comm)
C     Obtain the maximum value in vector z.   First find the local max
      max = z(1)
      do i = 2, M
        if (z(i).gt.max) max = z(i)
      enddo

C     Now find the global max
      call MPI_ALLREDUCE(max, globalmax, 1, MPI_DOUBLE_PRECISION,
     $     MPI_MAX, col_comm, ierr)
C     Compute A <- alpha * z * x^T + A
      call pdger(globalmax, z, x, a)
```


Solution: Lab 6, II.

```
C     Compute A <- alpha * z * x^T + A
      subroutine pdger(alpha, z, x, a)
      integer M, N
      parameter (M = 2, N = 3)
      double precision alpha, a(M,N), x(N), z(M)

      implicit none
      integer i, j

      do i = 1, M
        do j = 1, N
          a(i,j) = a(i,j) + alpha * z(i) * x(j)
        enddo
      enddo

      return
      end
```