Introduction to MPI

Jerry Ebalunode
Center for Advanced Computation and Data Systems
(CACDS)
http://cacds.uh.edu

http://support.cacds.uh.edu

University of Houston
Houston, TX

First Access Your Account

- Log into your accounts
  - Username or login = hpc_user**X**
  - Where x = sign in serial number 1 – 47
  - Password = **cacds2014**
- Use your web browser
  - Firefox, Chromium or Google chrome
- Slides could be downloaded from URL below

  /share/apps/tutorials/intro2mpi.pdf

Getting Started

Use the terminal to download intro2mpi.zip file to your home directory

- Run the following commands

cd

Now copy tutorial files to your home directory:

cp /share/apps/tutorials/intro2mpi.zip

unzip intro2mpi.zip

cd intro2mpi

module add openmpi

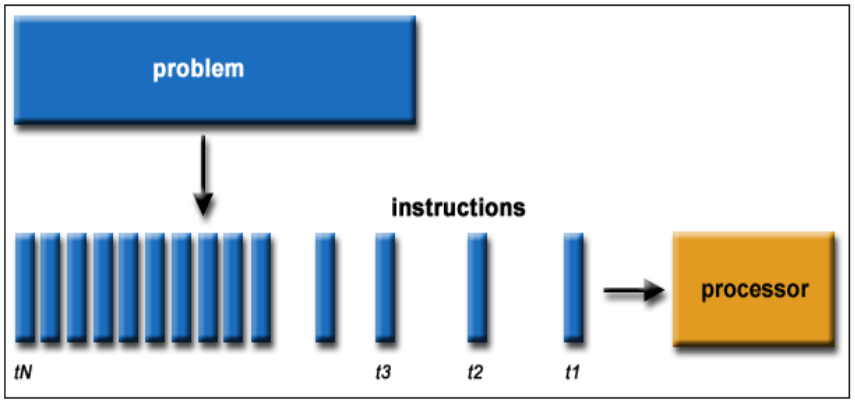
Overview

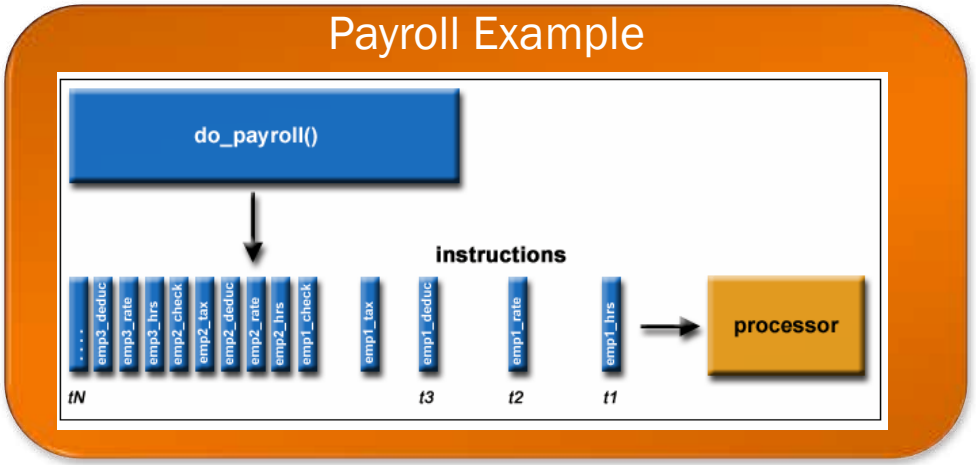
- Parallel Computing
- What is MPI?
- How do I run an MPI program?
- What does a simple MPI program look like?
- Basic MPI routines explained --examples and exercises.
- More MPI routines and capabilities reviewed.

What is Parallel Computing?

Serial Computing:

- Traditionally, software has been written for serial computation:
- A problem is broken into a discrete series of instructions
- Instructions are executed sequentially one after another
- Executed on a single processor
- Only one instruction may execute at any moment in time

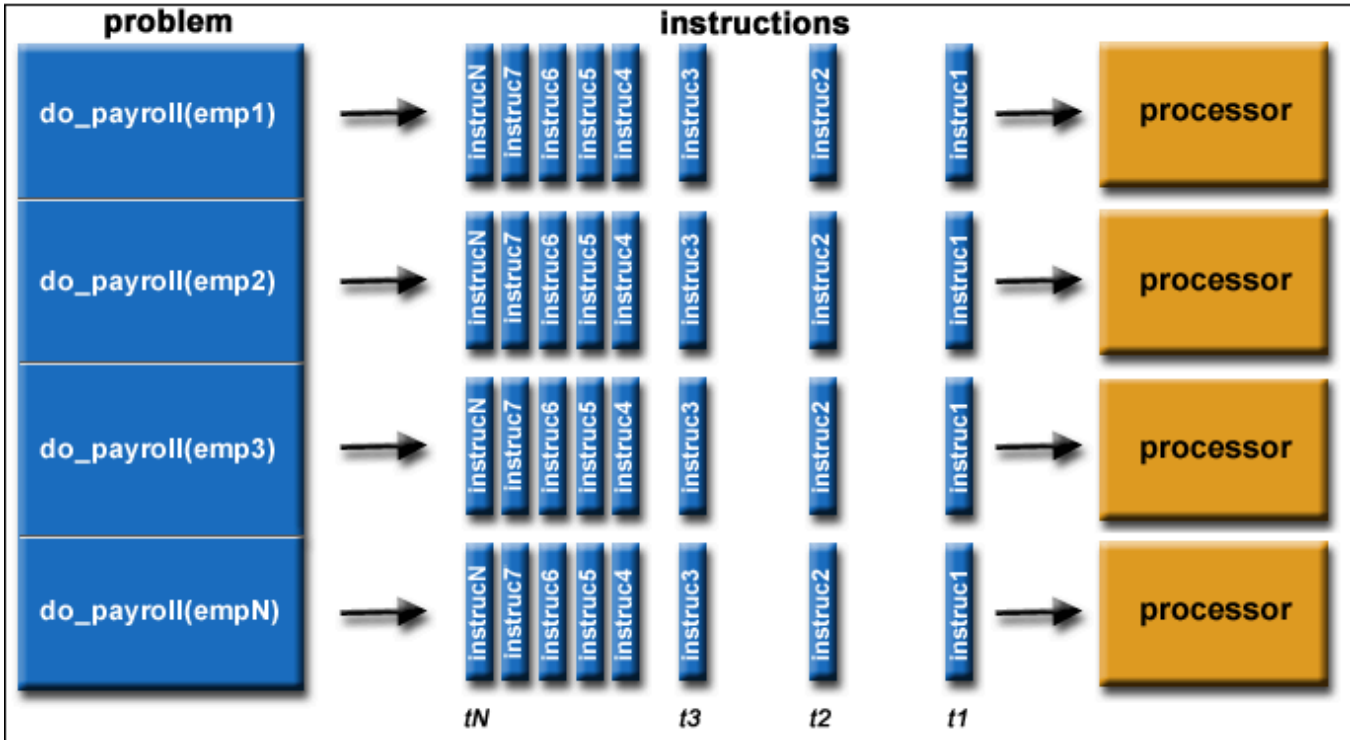




What is Parallel Computing? II

Parallel Computing is the simultaneous use of multiple compute resources to solve a computational problem:

- A problem is broken into discrete parts that can be solved concurrently
- Each part is further broken down to a series of instructions
- Instructions from each part execute simultaneously on different processors
- An overall control/coordination mechanism is employed



What is Parallel Computing? III

- The computational problem should be able to:
  - Be broken apart into discrete pieces of work that can be solved simultaneously;
  - Execute multiple program instructions at any moment in time;
  - Be solved in less time with multiple compute resources than with a single compute resource.
- The compute resources are typically:
  - A single computer with multiple processors/cores
  - An arbitrary number of such computers connected by a network (i.e, distributed memory platforms)

Types of Parallel Computing Models

- Data Parallel - the same instructions are carried out simultaneously on multiple data items (SIMD)
- Task Parallel - different instructions on different data (MIMD)
- SPMD (single program, multiple data) not synchronized at individual operation level
- SPMD is equivalent to MIMD since each MIMD program can be made SPMD (similarly for SIMD, but not in practical sense.)

Message passing (and MPI) is for MIMD/SPMD parallelism. High Performance Fortran is an example of an SIMD interface.

What is MPI

- MPI (Message-Passing Interface) is a library of subroutines for handling communication and synchronization for programs running on parallel platforms.
- MPI addresses primarily the message-passing parallel programming model, in which data is moved from the address space of one process to that of another process through cooperative operations on each process.
- MPI targets distributed memory platforms, such as the UH Maxwell cluster, but it often delivers improved performance on shared memory platforms also (such as the SGI Altix).
- MPI programs usually follow a single program multiple data (SPMD) format.

What is MPI II

- MPI is not a language, and all MPI operations are expressed as functions, subroutines, or methods, according to the appropriate language bindings which, for C and Fortran, are part of the MPI standard.
- The goal of the MPI is to develop a widely used standard for writing message-passing programs.
  - Establish a practical, portable, efficient, and flexible standard for message passing.
- http://www.mpi-forum.org/
- Currently on version 3.0 (September, 2013)

The Message-Passing Model

- A *process* is (traditionally) a program counter and address space.
- Processes may have multiple *threads* (program counters and associated stacks) sharing a single address space. MPI is for communication among processes, which have separate address spaces.
- Inter-process communication consists of
  - Synchronization
  - Movement of data from one process's address space to another's.

What Is Included In The MPI Standard?

- The standard at version 3.0 includes support for:
  - Point-to-point communication
  - Datatypes
  - Collective operations
  - Process groups
  - Communication contexts
  - Process topologies
  - Environmental management and inquiry
  - The Info object
  - Process creation and management
  - One-sided communication
  - External interfaces
  - Parallel file I/O
  - Language bindings for Fortran and C
  - Tool support

Why Use MPI?

- MPI provides a powerful, efficient, and *portable* way to express parallel programs
- MPI was explicitly designed to enable libraries…
- … which may eliminate the need for many users to learn (much of) MPI

How Do I Run My MPI Program?

Given a program MY_MPI_PROGRAM.c where MPI is initialized and used:

Load the MPI library

module load openmpi

compile with the appropriate MPI compiler wrapper

Fortran 77 Programs ===> mpif77
Fortran 90 Programs ===> mpif90
C Programs ===> mpicc
C++ Programs ===> mpic++ or mpiCC or mpicxx

Since we have a C program we use mpicc

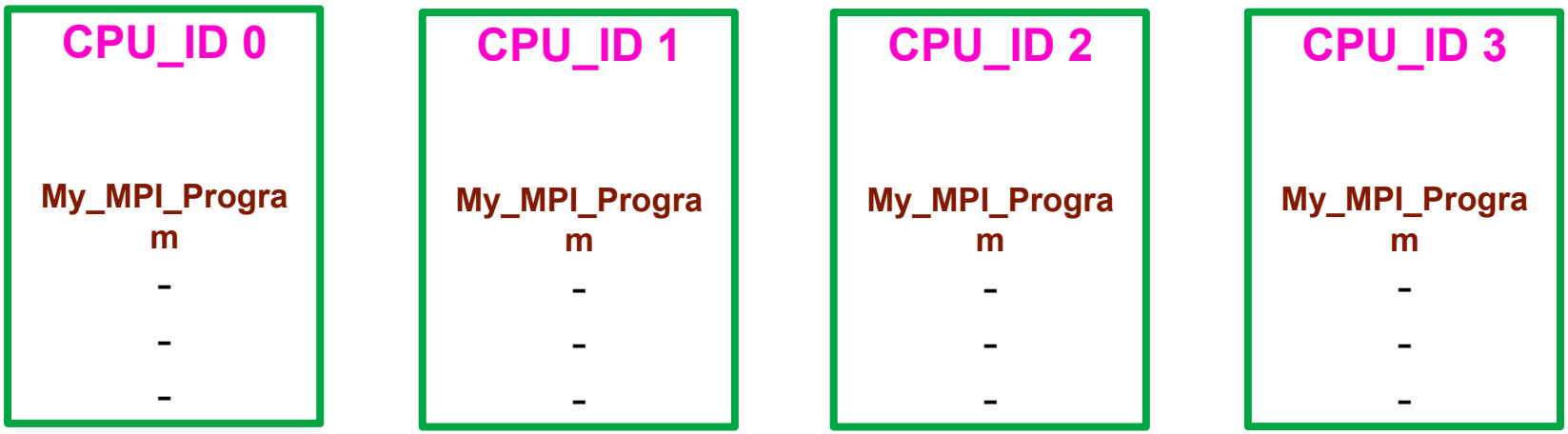
mpicc -o MY_MPI_PROGRAM MY_MPI_PROGRAM.c

Run the executable using 4 processors:

mpirun -np 4 ./MY_MPI_PROGRAM

How Does MPI Get Your Application To Run In Parallel?

- MPI spawns an identical copy of MY_MPI_PROGRAM on each of the m requested processors.
  - e.g. If m=4, there will be four identical jobs running on four processors at the same time!



A Minimal MPI Program (C) "Hello world"

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

int main( int argc, char *argv[] )
{
    MPI_Init( &argc, &argv );
    printf( "Hello, world!\n" );
    MPI_Finalize();
    return 0;
}
```

A Minimal MPI Program (Fortran)

```
program main
use MPI
integer ierr

call MPI_INIT( ierr )
print *, 'Hello, world!'
call MPI_FINALIZE( ierr )
end
```

Example 1: ("HELLO WORLD")

```
/* --Include the standard C++ I/O header file */
#include <iostream>

/* --Include the mpi header file */
#include <mpi.h>
using namespace std;

int main(int argc, char* argv[])
{
      int myid, numprocs;
/*--Initialize MPI*/
      MPI_Init(&argc, &argv);

/*--Who am I? --- get my rank=myid */
       MPI_Comm_rank( MPI_COMM_WORLD, & myid );

/*--How many processes in the global group? */
      MPI_Comm_size( MPI_COMM_WORLD, &numprocs);

      cout << "I am process  "<< myid <<" of "  << numprocs  <<" MPI
processes\n";

/*--Finalize MPI */
      MPI_Finalize();

      return 0;
}
```

Exercise 1

Try compiling this example

```
cd
cd intro2mpi
module load openmpi
mpic++ example1.c -o example
```

Now run the executable on 4 processors

```
mpirun -np 4 ./example
```

```
Output:
I am process  2 of 4 MPI
processes
I am process  0 of 4 MPI
processes
I am process  1 of 4 MPI
processes
I am process  3 of 4 MPI
processes
```

Assigning Work

- How can you assign different work to each processor, if all processors run the same program?
- The short answer:
  - MPI assigns a rank (an integer number) to each process to identify it.
  - The routine **MPI_COM_RANK** returns the rank of the calling process.
  - The routine **MPI_COM_SIZE** returns the size or number of processes in the application.
- These two parameters, size and rank, can then be used (through block if statements or otherwise) to differentiate the computations that each process will execute.

```
if (my_rank == 0)
        X = ....
        Y = ....
else
        Z1 = ...
        Z2 = ...
```

The Longer Answer

- When MPI is initialized, it creates a communicator, consisting of a group of processes and their labels. This communicator is called,

MPI_COMM_WORLD

- The number of processes (m) in this communicator is determined when we submit the MPI job:

mpirun -np m MY_MPI_JOB

- Each process can probe for the value of m by calling the MPI routine

MPI_COMM_SIZE

- Each process in the MPI_COMM_WORLD group is assigned a rank, an integer with incremental value between 0 and m-1. Each process can determine its own rank by calling the routine

MPI_COMM_RANK

What are the minimum entries in a program to run an MPI job?

There are three required entries on any MPI program:

C/C++ Programs

include 'mpi.h'

MPI_INIT()

MPI_FINALIZE()

- (No MPI program will run without these statements!)
- A template for an MPI program can be found at intro2mpi in template.f

What are the minimum entries in a program to run an MPI job?

There are three required entries on any MPI program:

Fortran Programs

```
include 'mpif.h'

call MPI_INIT(ierr)

call MPI_FINALIZE(ierr)
```

- (No MPI program will run without these statements!)
- A template for an MPI program can be found at intro2mpi in template.f

Template for MPI Programs in C
Template.c

```
                                                    /*--Include standard I/O C header file */
#include <stdio.h>

                                                    /*--Include the MPI header file */
#include "mpi.h"

int main( int argc, char* argv[])
{
                                                    /*--Declare all variables and arrays. */
     int myid, numprocs, itag;

                                                    /* --> Required statement */
     MPI_Init(&argc,&argv);
                                                    /*--Who am I? --- get my rank=myid */
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);

                                                    /*--How many processes in the global group? */
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);

                                                    /*--Finalize MPI */
                                                    /* ---> Required statement */
     MPI_Finalize();
    return 0;
}
```

Template for MPI Programs in Fortran

```
program template

                                        ! highly recommended. It will make
                                        ! debugging infinitely easier.
implicit none

                                        !--Include the mpi header file
  include 'mpif.h'

                                        ! --> Required statement

                                        !--Declare all variables and arrays.
  integer ierr, myid, numprocs, itag
  integer irc

                                        !--Initialize MPI
                                        ! --> Required statement
    call MPI_INIT( ierr )

                                        !--Who am I? --- get my rank=myid
  call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )

                                        !--How many processes in the global group?
  call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )

                                        !--Finilize MPI
                                        ! ---> Required statement
  call MPI_FINALIZE(ierr)

  stop
  end
```

Exercise 2

- Starting from template.c or template.f, write a program that given a common value of x (e.g. x=5 in all processes), computes:
  - y=x2 in process 0
  - y=x3 in process 1
  - y=x4 in process 2
  - and writes a statement from each process that identifies the process and reports the values of x and y from that process.

Exercise 2: Solution Using Three Processors

```
//program template
 //!--Define new variables used
.....
   int ip
   float  x,y
 ......
   {MPI_COMM_RANK and MPI_COMM_SIZE CALLS.....}

//!--Insert the calculations after the size (numprocs)
//!  and rank (myid) are known.
//!--Set the value of x on all processes
   x=5;

!--Define the value of y on each process .....
      if(myid == 0)
       y=pow(x,2);
      else if (myid == 1)
       y=pow(x,3);
      else  (myid == 2)
       y=pow(x,4);

//!--...and print it.

      printf("On process %f, y= %f",myid ,y);
```

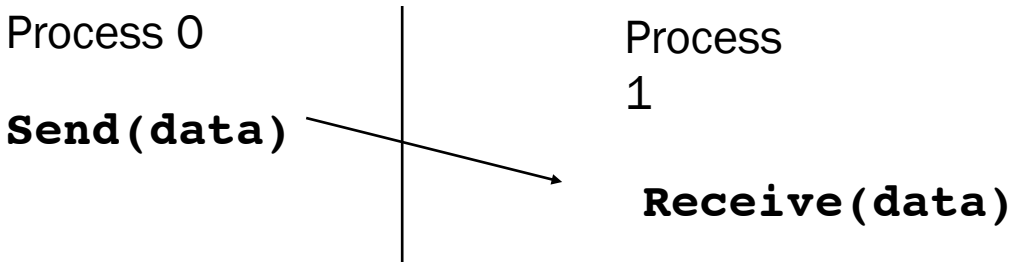
Exercise 2: Solution, a more concise version

```
//program template
.
.
//!--Define new variables used
   int ip;
   float x,y;
      .
      .
   {MPI_COMM_RANK and MPI_COMM_SIZE CALLS……}

//!--Insert the calculations after the size (numprocs)
//!  and rank (myid) are known.

//!--Set the value of x on all processes
    x=5;

//!--Define the value of y on each process and print it.

        y=pow(x,(2+myid));
      printf("On process %f, y= %f",myid ,y);

      .
      .
```

Output from exercise 2:

Cooperative Operations for Communication

- The message-passing approach makes the exchange of data *cooperative*.
- Data is explicitly *sent* by one process and *received* by another.
- An advantage is that any change in the receiving process's memory is made with the receiver's explicit participation.
- Communication and synchronization are combined.



Communication

- MPI is designed to manage codes running on distributed memory platforms. Thus data residing on other processes is accessed through MPI calls.

- Although MPI includes a large number of communication routines, most applications require only a handful of them.
- A minimal set of routines that most parallel codes run with are:

MPI_INIT

MPI_COMM_SIZE

MPI_COMM_RANK

MPI_SEND

MPI_RECV

MPI_FINALIZE

Point To Point Communications

- MPI_Send and MPI_Recv perform "point to point" communications.

- The two routines work together to complete a transfer of data from one process to another.

- One process posts a send operation, and the target process posts a receive for the data being transferred.

e.g. (pseudocode)

```
if (my_rank == 0)
 &    MPI_SEND(.....,1,..)

 if (my_rank == 1)
 &    MPI_RECV(.....,0,.....)
```

Blocking MPI Send Routine

MPI_Send(buf, count, datatype, dest, tag, comm)

- buf =initial address of send buffer (choice)
- count =number of entries to send (integer)
- datatype=datatype of each message entry (handle)
- dest =rank of destination (integer)
- tag =message tag
- comm =communicator (handle)
- ierror =return error code (integer) only for fortran
- some datatype available handles:

MPI_SHORT
MPI_INT
MPI_LONG
MPI_FLOAT
MPI_DOUBLE
MPI_CHAR
MPI_C_COMPLEX
MPI_BYTE
MPI_PACKED

Blocking MPI Receive Routine

- **MPI_Recv (buf, count, datatype, source, tag, comm, status)**
  - OUT buf initial address of receive buffer (choice)
  - IN count number of elements in receive buffer (non-negative integer)
  - IN datatype datatype of each receive buffer element (handle)
  - IN source rank of source or MPI_ANY_SOURCE (integer)
  - IN tag message tag or MPI_ANY_TAG (integer)
  - IN comm communicator (handle)
  - OUT status status object (Status)

- **Any routine that calls MPI_RECV, should declare the status array:**
  - integer status(MPI_STATUS_SIZE)
  - The status array contains information on the received message, such as its tag, source, error code. The number of entries received can also be obtained from this array.
- C implementation/signature
  - int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)

EXAMPLE 2: Point to Point Communication

```
 ….
     int   i,  dest,   numprocs, my_rank,  source;

     MPI_Status status;
          float x[5],  y[5];    //Buffers for copy and receive
          int     count   =5;

/* --the common calling arguments */

     int    tag     = 2;
/*---process 0 send data to process 1 (dest=1) */
     if (my_rank == 0) {
       for (i=1;i<=5; i++) {
          x[i-1]=(float) i;
       }
        dest    = 1;
       MPI_Send( &x[0], count,  MPI_FLOAT,  dest,  tag,   MPI_COMM_WORLD);
     }

/*---process 1 receives data from process 0 (source=0) */
     if (my_rank == 1) {
    /* float y=0.; */
       int source=0;

       MPI_Recv(&y[0],  count, MPI_FLOAT,  source,  tag, MPI_COMM_WORLD, &status);

    printf("At process %d y= %.1f %.1f %.1f %.1f %.1f \n", my_rank,y[0],
              y[1],y[2],y[3],y[4]);
     }
…..
     return 0;
    }
```

The full program can be copied from example2.c or example2.f

The Message Envelope

- How does an MPI process select which message to receive if more than one message has been sent to it?
- Each message carries verification information with it called the **message envelope**.
- The message envelope consists of the following:

source
destination
tag
communicator

- A message is received only if the arguments in the posted receive call agree with the message envelope of an incoming message.

Exercise 3

- Using the program in exercise 2, send all values of y to process 0 and compute the average of y at process 0.

- Print the result, including the process rank from where it is being printed.

Solution of Exercise 3

```
 int tag;                MPI_Status status;
 float  x, y, buff;
     .....
     ......
    {CALCULATION OF Y ON THE DIFFT. PROCESSES}
     .
/*--Average the values of y*/
    tag=1;
    if (myid == 0) {
     for (int  ip=0; ip < numprocs; ip++) {
       MPI_Recv(buff, 1, MPI_REAL, ip, tag,  MPI_COMM_WORLD, status);
       y+=buff;
      }
     y=y/float(numprocs);
    printf ("The average value of y is %d", y );
    else
    MPI_Send(y  , 1, MPI_FLOAT, 0, tag, &MPI_COMM_WORLD);
.......
```

Wildcards

- What if I want to receive a message regardless of its source and/or tag?

====> Replace the source and/or tag entry on the MPI_RECV call with a wildcard:

- Ignore source by using the MPI_ANY_SOURCE wildcard:

MPI_Recv(buf, count, datatype, & MPI_ANY_SOURCE, tag, comm,s tatus)

- Ignore tag by using the MPI_ANY_TAG wildcard:

MPI_Recv(buf, count, datatype, &source, MPI_ANY_TAG, comm, status)

- Ignore tag and source:

MPI_Recv(buf, count, datatype, &MPI_ANY_SOURCE, MPI_ANY_TAG, comm, status)

- *WILDCARDS SHOULD ONLY BE USED WHEN ABSOLUTELY NECESSARY!*

Blocking vs Non-Blocking Communications

- MPI_Send and MPI_Recv are **blocking** communications routines.

- What does it mean for MPI_Send to be a **blocking** routine?
  - Once a call to MPI_SEND is posted by a process, the call does not return control to the calling program or routine, until the buffer containing the data to be copied unto the receiving process can be safely overwritten (This insures that the message being sent is not "corrupted" before the sending is complete)

- What does it mean for MPI_Recv to be a **blocking** routine?
  - The call does not return control to the calling program until the data to be received has in fact been received.

Avoiding "Hung" Processes

A "Hung" condition occurs when one or more processes reach a call to an MPI blocked receive routine, but the message never arrives. The process will wait indefinitely and no error message will be generated. (The same will happen with a blocked send message that is never completed).

A "Hung" condition can occur when two processes exchange messages, if the exchange is not programmed carefully.

First: an example of an exchange that will never "hang":
Can you tell why? (If not compare to the example on next page).

```
//--Exchange messages
    if (myid == 0) {
      MPI_Send(a,1,MPI_FLOAT,1,tag,MPI_COMM_WORLD);
      MPI_Recv(b,1,MPI_FLOAT,1,tag,MPI_COMM_WORLD, &status);
   }
    else if (myid == 1) {
       MPI_Recv(a,1,MPI_FLOAT,0,tag,MPI_COMM_WORLD, & status);
       MPI_Send(b,1,MPI_FLOAT,0,tag,MPI_COMM_WORLD);
    }
```

The code above is an excerpt from **example3_a.c**
Please verify that this code will run to completion without a problem

Example of a hanging program

- Suppose that the order of the send and receive calls are modified as follows

```
//!--Exchange messages
    if (myid == 0) {
       MPI_Recv(b,1,MPI_FLOAT,1,tag,MPI_COMM_WORLD, &  status );
       MPI_Send(a,1,MPI_FLOAT,1,tag,MPI_COMM_WORLD );
    }
    else if (myid == 1) {
        MPI_Recv(a,1,MPI_FLOAT,0,tag,MPI_COMM_WORLD, & status );
        MPI_Send(b,1,MPI_FLOAT,0,tag,MPI_COMM_WORLD );
     }
```

- The code above is an excerpt from example3_a.c

- Will this program run as well as example3_a?
- Why?

Example of a hanging program: Why?

The program in example3_c.c will not run at all, it will hang!
(that is, it will never complete and give no error diagnostic -- other that running out of time).

Here is why:

1. Each of processes 0 and 1 calls a blocking MPI_Recv routine and expects to receive a message from the other process.
2. Neither process will continue on to the next statement until the information has been received (or is at least safely on its way).
3. At this point neither process has actually SENT any message to the other process.
4. **Thus both processes will wait indefinitely for a message that will never come….**

Example of a Program that MIGHT Hang

- The following order of the send and receive calls will work on some platforms but not others.
- IT IS NOT RECOMMENDED!

```
//!--Exchange messages
    if (myid == 0) {
     MPI_Send(a,1,MPI_FLOAT,1,tag,MPI_COMM_WORLD);
     MPI_Recv(b,1,MPI_FLOAT,1,tag,MPI_COMM_WORLD, & status);
    }
    else if (myid == 1) {
     MPI_Send(b,1,MPI_FLOAT,0,tag,MPI_COMM_WORLD);
     MPI_Recv(a,1,MPI_FLOAT,0,tag,MPI_COMM_WORLD,&status);
    }
```

- The full program can be found in **example3_b.c**
- A safe alternative is to use the **MPI_SENDRECV** routine instead.

MPI_SendRecv

MPI_SendRecv(sendbuf, sendcount, sendtype, & dest, sendtag, recvbuf, recvcount, recvtype, &source, recvtag, comm, status)

- sendbuf =initial address of send buffer (choice)
- sendcount = # of entries to send (integer)
- sendtype =type of entries in send buffer (handle)
- dest =rank of destination (integer)
- sendtag =send tag (integer)
- recvbuf =initial address of receive buffer (choice)
- recvcount=max. num. of entries to receive (integer)
- recvtype =type of entries in receive buffer (handle)
- source =rank of source (integer)
- recvtag =receive tag (integer)
- status =return status(integer)
- comm =communicator (handle)

Example 4: Using MPI_SendRecv I

Example 4:

Shows the use of the SendRecv MPI call replacing a pair of consecutive send and receive calls originating from a single process. Note that the communications here are the same as in example3_b, except that the SendRecv insures that no deadlock or hangup occurs.

- Here is an excerpt from example 4,

- //--Exchange messages

```
tag1=1;
tag2=2;
if (myid == 0)  {
MPI_SendRecv (a, 1 , MPI_FLOAT, 1, tag1, & b, 1, MPI_FLOAT, 1, tag2, & MPI_COMM_WORLD, status); }
else if (myid == 1) {
  MPI_SendRecv(b,1,MPI_FLOAT,0,tag2, &a,1,MPI_FLOAT,0,tag1, &MPI_COMM_WORLD, status );
 }
```

See example4.c

Example 5: Using MPI_SendRecv II

- MPI_SENDRECV is compatible with simple MPI_SEND and MPI_RECV routines. Here is an example that illustrates the point.

- Example 5
- Process 0 sends a to process 1 and receives b from process 2:

```
MPI_COMM   comm;          comm =MPI_COMM_WORLD;
tag1=1;   tag2=2;
if     (myid == 0)
   MPI_SendRecv(a,1,MPI_FLOAT,1,tag1, &   b,1,MPI_FLOAT,2,tag2, & comm, status);
else if (myid==1)
   MPI_Recv(a,1,MPI_FLOAT,0,tag1, &comm, status);
 else if (myid==2)
   MPI_Send(b,1,MPI_FLOAT,0,tag2,&comm);
```

- (See example5.c)

Non-blocking Communications

The most common non-blocking point-to-point MPI communication routines are:

MPI_ISend(buf, count, datatype, dest, tag, comm, request)

MPI_IRecv(buf, count, datatype, source, tag, comm, request)

- buf =initial address of send buffer (choice)
- count =number of entries to send/receive (integer)
- datatype=datatype of each entry (handle)
- dest =rank of destination process (integer)
- source =rank of source process(integer)
- tag =message tag
- comm =communicator (handle)
- request =request handle (handle)

Blocking vs Non-blocking

- **What is the difference between MPI_ISend and MPI_Send?**
  - MPI_ISend returns control to the calling routine immediately after posting the send call, before it is safe to overwrite (or use) the buffer being sent.
  - (MPI_IRecv and MPI_Recv differ in a similar way)
- **What is the advantage of using non-blocking communication routines?**
  - Performance can be improved by allowing computations that do not involve the buffer being sent (or received) to proceed simultaneously with the communication.
- **How can the communicated buffer be re-used safely?**
  - The MPI_ISend (and MPI_IRecv) return a handle: the request argument. The MPI_Wait routine can later be called in order to "complete" the request communication. MPI_Wait blocks computation until the request in question is complete and it is safe to re-use the buffer.

Blocking vs Non-blocking II

The non-blocking routines MPI_ISend and MPI_IRecv are similar to their blocking couterparts, MPI_Send and MPI_Recv.

- The difference between them is that non-blocking communications return control to the calling routine BEFORE it is safe to re-use the buffer being sent or received.
- This allows the program to proceed with computations not involving the communication buffer, while the communication completes.
- Before the program is to use the sent/received buffer, a call to MPI_Wait is necessary.
- MPI_Wait is a blocking routine. It does not return control to the calling routine until it is safe to re-use the buffer.

NON-BLOCKING: An Example

- (Pseudocode)

- Post a non-blocking send of variable a.

MPI_ISend(a,……,REQUEST1,…)

- While the communication of a takes place, compute the values of b, c, and d (which do not involve a).

```
b=x2
c=y3
d=b+c
```

- Block computation until it is safe to use a again.

MPI_Wait(REQUEST1,status)

- Use a on the computation of e, modify a, etc.

```
e=a+b
a=d
```

COLLECTIVE MPI ROUTINES

- Collective MPI routines involve simultaneous communications among all the processors in a communicator group. The communication involves a group or groups of processes.

- There are 3 types of collective communications

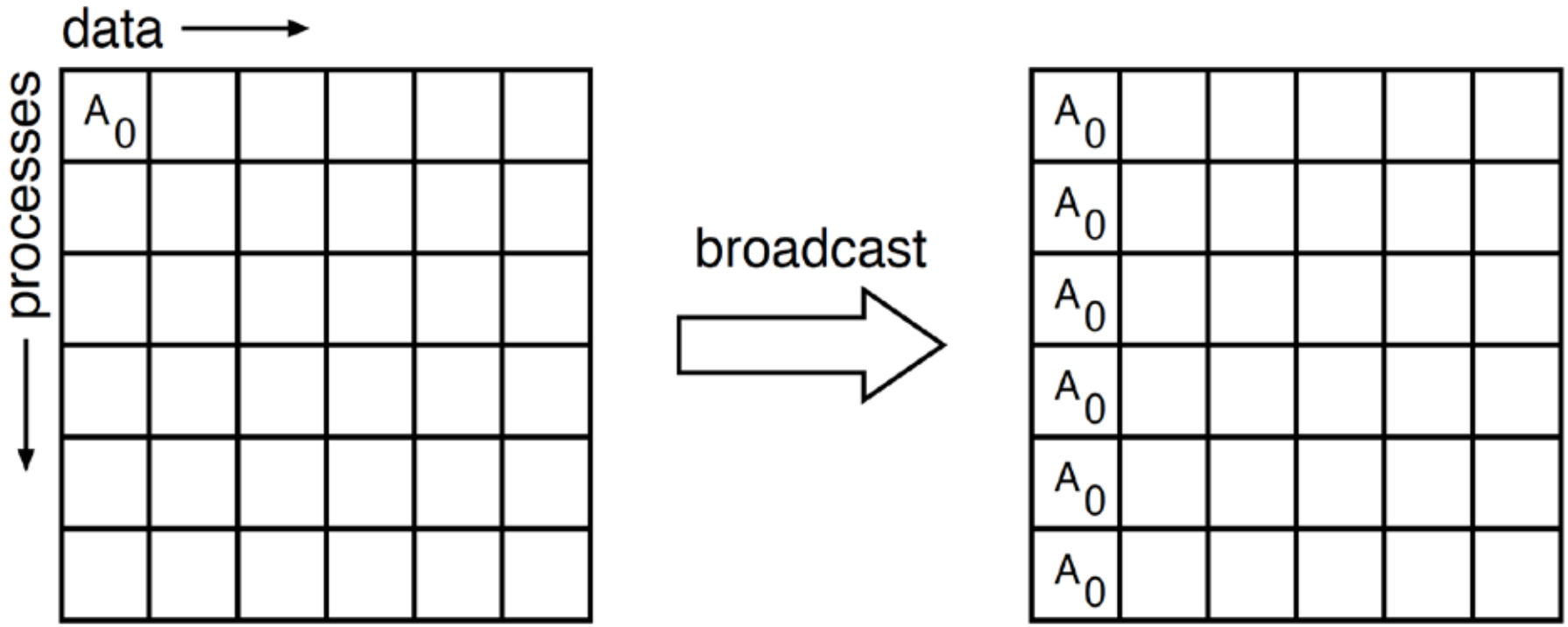
  - Barrier synchronization

  - Global communications

    - Broadcast
    - Gather
    - Scatter

  - Global Reduction Operations
    - sum
    - max
    - min

Broadcast



Broadcast Routine

The broadcast MPI routine is one of the most commonly used collective routines.

- The root process broadcasts the data in buffer to all the processes in the communicator.
- All processes must call MPI_BCAST with the same root value.

MPI_Bcast(buffer, count, datatype, root, comm)

buffer =initial address of buffer (choice)

count =number of entries in buffer (integer)

datatype=datatype of buffer (handle)

root =rank of broadcasting process (integer)

comm =communicator (handle)

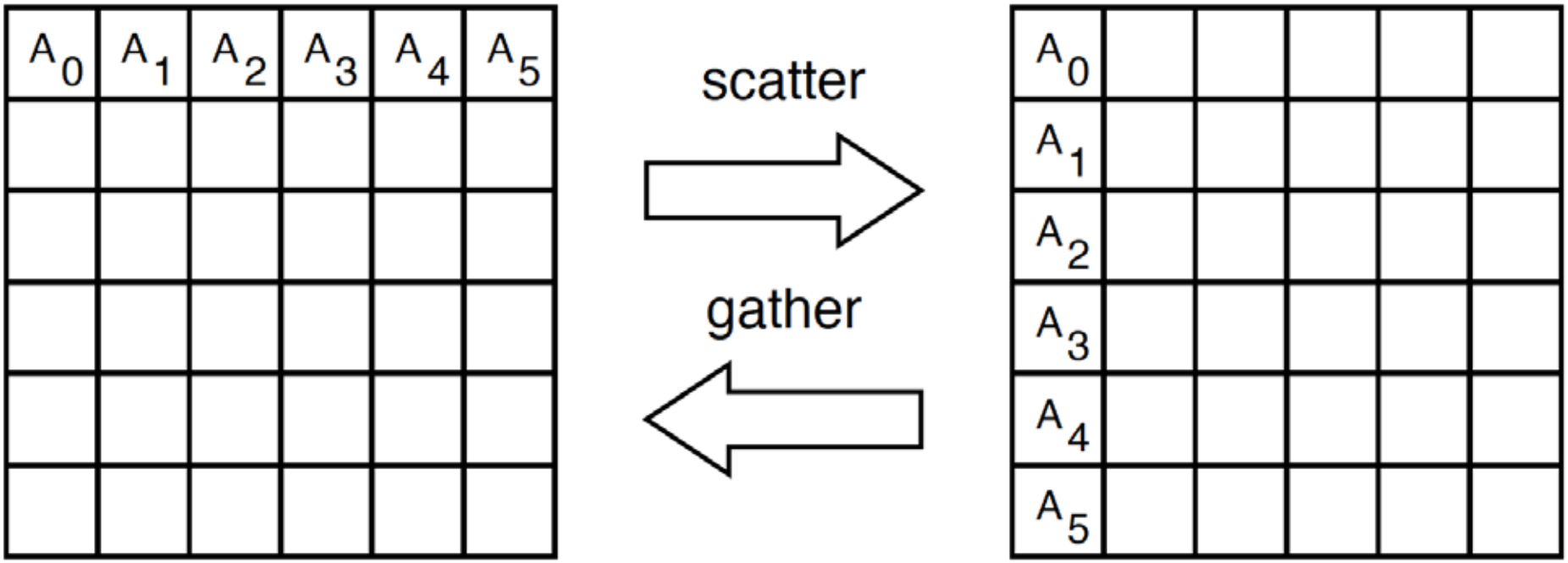
Exercise 4: Using Broadcast

- Modify the code for exercise 2 so that the value of x is defined ONLY on processor 0. Add the necessary code to broadcast the value of x to all the other processors.

Solution to Exercise 4

```
//Partial solution.
.
//!--Define new variables used
   int  ip;
   float  x, y;
      .
      .
   {MPI_COMM_RANK and MPI_COMM_SIZE CALLS.....}

//!--Insert the calculations after the size and rank
//!  are known.

//!--Set the value of x on process 0.
      if (myid == 0)    x=5;

//!--Broadcast the value of x to all processes.
      MPI_Bcast( x, 1, MPI_FLOAT, 0,MPI_COMM_WORLD );

!--Define the value of y on each process and print it.
 for (ip=0; ip<numprocs;ip++)
 {
    if(myid == ip-1)
    {
            y = pow(x,(2+myid));
            printf("On process %f, y= %f",myid, y);
    }
   }
      .
      .
```

Scatter and Gather



Scatter

Sends data from one process to all other processes in a communicator

MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

- IN sendbuf address of send buffer (choice, significant only at root)
- IN sendcount number of elements sent to each process (non-negative integer, signicant only at root)
- IN sendtype data type of send buffer elements (significant only at root) (handle)
- OUT recvbuf address of receive buffer (choice)
- IN recvcount number of elements in receive buffer (non-negative integer)
- IN recvtype data type of receive buffer elements (handle)
- IN root rank of sending process (integer)
- IN comm communicator (handle)

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

Scatter Example

```
/*Scatter sets of 100 ints from the root to each process in the
group*/
....
MPI_Comm comm;
int gsize,*sendbuf;
int root, rbuf[100];
...
MPI_Comm_size(comm, &gsize);
sendbuf = (int *)malloc(gsize*100*sizeof(int));
...
MPI_Scatter(sendbuf, 100, MPI_INT, rbuf, 100, MPI_INT, root,
comm);
```

Scatter Example II

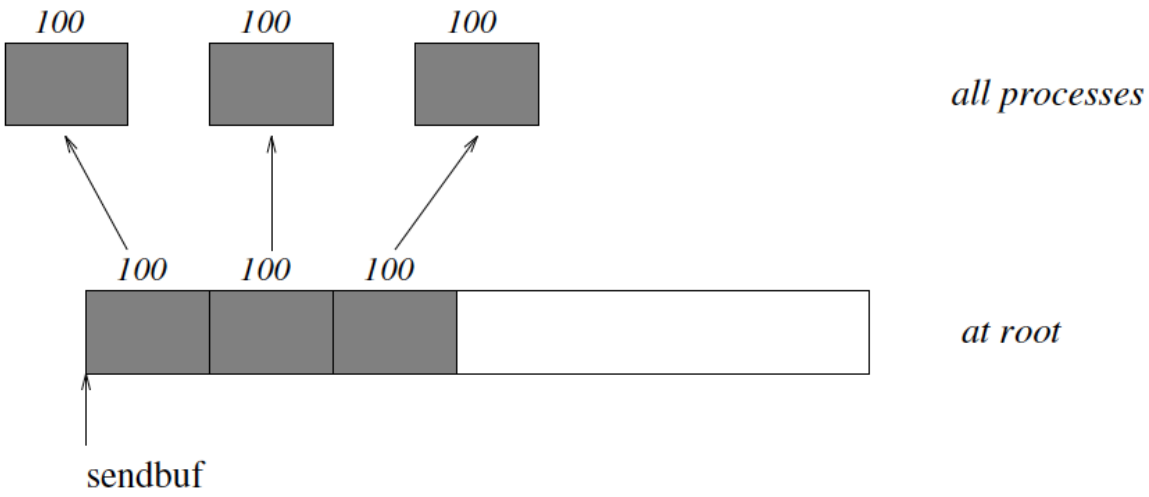


Figure 5.9: The root process scatters sets of 100 `int`s to each process in the group.

Gather

Gathers together values from a group of processes

MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

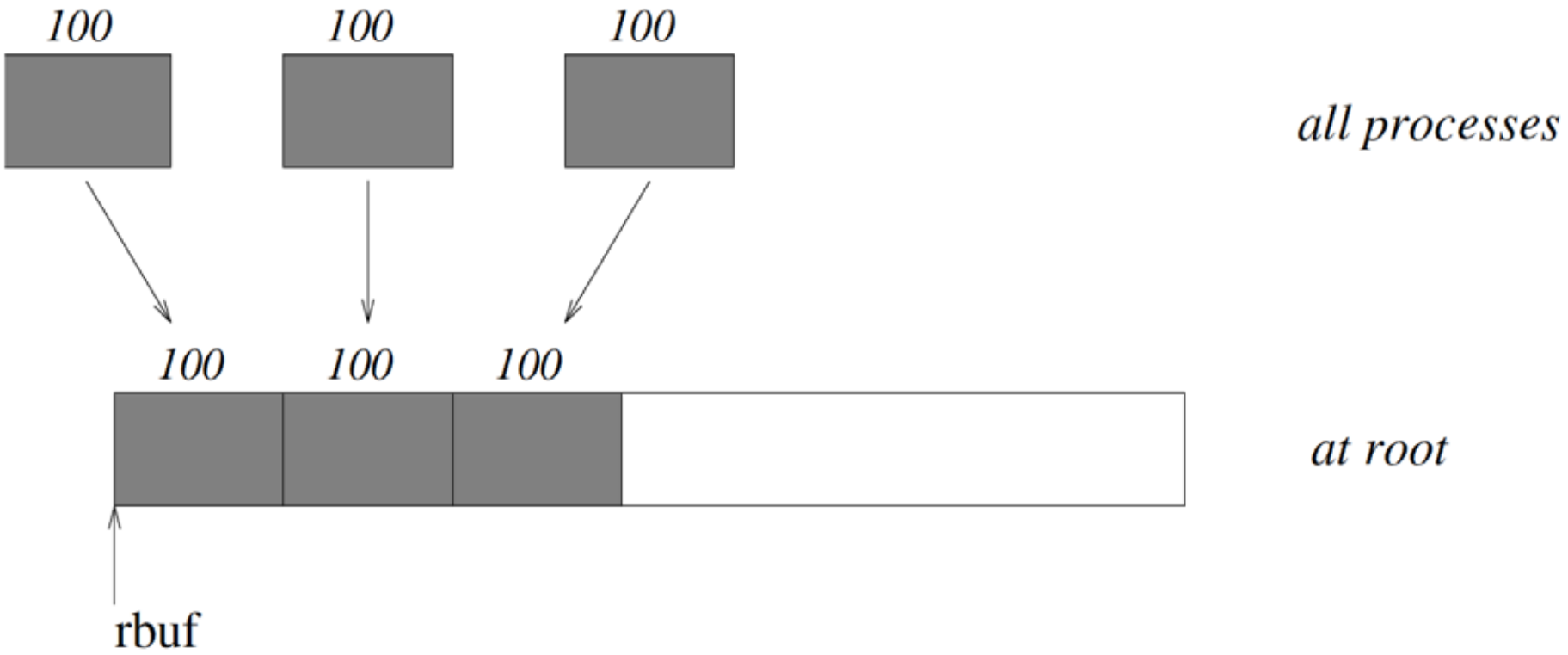
- IN sendbuf starting address of send buffer (choice)
- IN sendcount number of elements in send buffer (non-negative integer)
- IN sendtype data type of send buffer elements (handle)
- OUT recvbuf address of receive buffer (choice, significant only at root)
- IN recvcount number of elements for any single receive (non-negative integer, significant only at root)
- IN recvtype data type of recv buffer elements (significant only at root) (handle)
- IN root rank of receiving process (integer)
- IN comm communicator (handle)

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

Gather Example

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

Gather Example



The root process gathers 100 ints from each process in the group.

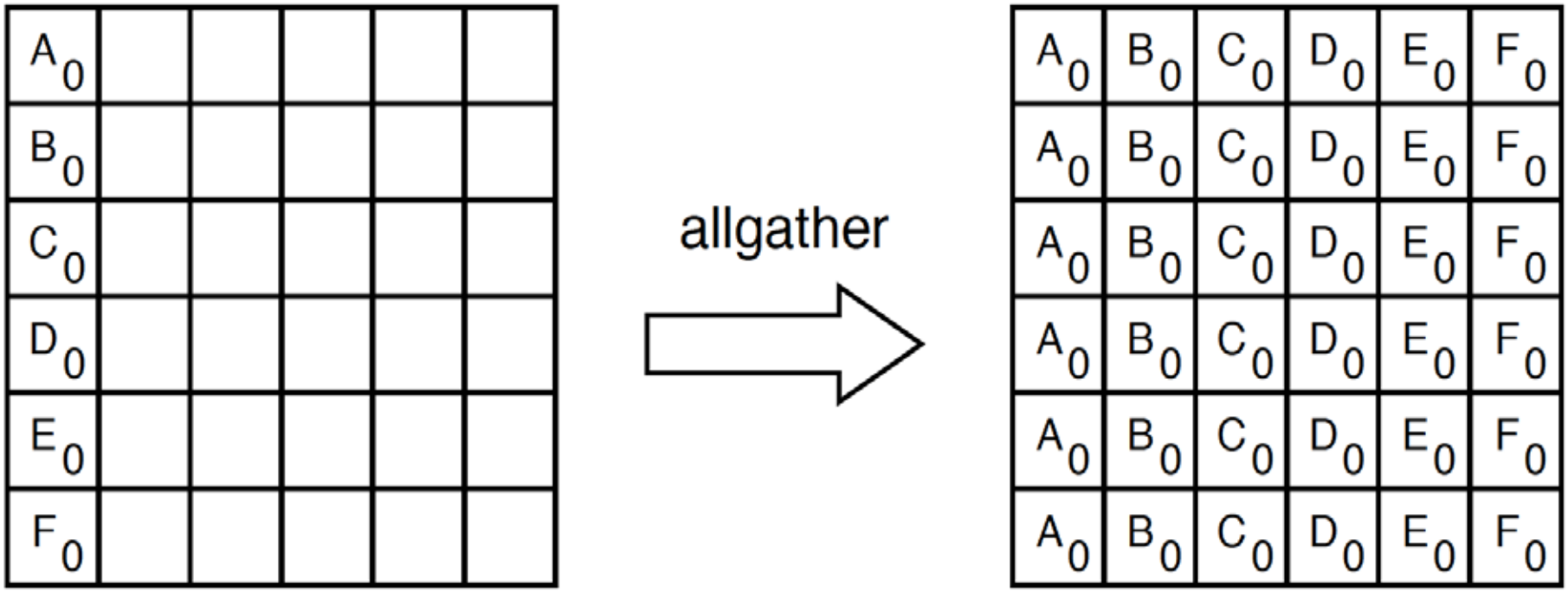
Summary

- The parallel computing community has cooperated on the development of a standard for message-passing libraries.
- MPI subsets are easy to learn and use.
- Lots of MPI learning material are available.

References

- The Standard itself:
  - at http://www.mpi-forum.org
- General MPI reference:
  - http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf
- Tutorials
  - https://computing.llnl.gov/tutorials/mpi/
- Books:
  - *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, by Gropp, Lusk, and Skjellum, MIT Press, 1994.
  - *MPI: The Complete Reference,* by Snir, Otto, Huss-Lederman, Walker, and Dongarra, MIT Press, 1996.
  - *Designing and Building Parallel Programs*, by Ian Foster, Addison-Wesley, 1995.
  - *Parallel Programming with MPI*, by Peter Pacheco, Morgan-Kaufmann, 1997.
  - *MPI: The Complete Reference Vol 1 and 2,*MIT Press, 1998(Fall).
- Other information on Web:
  - at `http://www.mcs.anl.gov/mpi`
  - pointers to lots of stuff, including other talks and tutorials, a FAQ, other MPI pages

AllGather


A0
B0
C0
D0
E0
F0
allgather
A0 B0 C0 D0 E0 F0
A0 B0 C0 D0 E0 F0
A0 B0 C0 D0 E0 F0
A0 B0 C0 D0 E0 F0
A0 B0 C0 D0 E0 F0
A0 B0 C0 D0 E0 F0

AllGather

- MPI_Allgather routine gathers data from all tasks and distribute the combined data to all tasks
- MPI_Allgather can be thought of as MPI_GATHER, but where all processes receive the result, instead of just the root. The block of data sent from the j-th process is received by every process and placed in the j-th block of the buffer recvbuf.

AllGather

MPI_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)

- IN sendbuf starting address of send buer (choice)
- IN sendcount number of elements in send buffer (non-negative integer)
- IN sendtype data type of send buffer elements (handle)
- OUT recvbuf address of receive buffer (choice)
- IN recvcount number of elements received from any process (nonnegative integer)
- IN recvtype data type of receive buer elements (handle)
- IN comm communicator (handle)

int MPI_Allgather(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)

Allgather Example

- Using MPI_ALLGATHER, we will gather 100 ints from every process in the group to every process.

```
MPI_Comm comm;
int gsize,sendarray[100];
int *rbuf;
...
MPI_Comm_size(comm, &gsize);
rbuf = (int *)malloc(gsize*100*sizeof(int));
MPI_Allgather(sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, comm);
```

- After the call, every process has the group-wide concatenation of the sets of data.

One-Sided Operations for Communication

- One-sided operations between processes include remote memory reads and writes
- Only one process needs to explicitly participate.
- An advantage is that communication and synchronization are decoupled
- One-sided operations are part of MPI-2.