
Parallel Programming with MPI

**Science & Technology Support
High Performance Computing**

**Ohio Supercomputer Center
1224 Kinnear Road
Columbus, OH 43212-1163**

Table of Contents

- [Setting the Stage](#)
- [Brief History of MPI](#)
- [MPI Program Structure](#)
- [Message Passing](#)
- [Point-to-Point Communications](#)
- [Non-Blocking Communications](#)
- [Collective Communication](#)
- [Derived Data Types](#)
- [Virtual Topologies](#)
- [Problem Set](#)

Setting the Stage

- [Overview of parallel computing](#)
- [Parallel architectures](#)
- [Parallel programming models](#)
- [Hardware](#)
- [Software](#)

Overview of Parallel Computing

- **Parallel computing is when a program uses concurrency to either**
 - decrease the runtime needed to solve a problem
 - increase the size of problem that can be solved
- **The direction in which high-performance computing is headed!**
- **Mainly this is a price/performance issue**
 - Vector machines (e.g., Cray T94) very expensive to engineer and run
 - Commodity hardware/software - Clusters!

Writing a Parallel Application

- **Decompose the problem into tasks**
 - Ideally, these tasks can be worked on independently of the others
- **Map tasks onto “threads of execution” (processors)**
- **Threads have shared and local data**
 - Shared: used by more than one thread
 - Local: Private to each thread
- **Write source code using some parallel programming environment**
- **Choices may depend on (among many things)**
 - the hardware platform to be run on
 - the level performance needed
 - the nature of the problem

Parallel Architectures

- **Distributed memory (OSC Cray T3E)**
 - Each processor has local memory
 - Cannot directly access the memory of other processors
- **Shared memory (OSC Origin 2000, Beowulf Cluster)**
 - Processors can directly reference memory attached to other processors
 - Shared memory may be physically distributed
 - **The cost to access remote memory may be high!**
 - Several processors may sit on one memory bus (SMP)
- **Combinations are very common, e.g. Beowulf Cluster:**
 - 32 compute nodes, each with 4 CPUs sharing 2GB of memory
 - High-speed interconnect between nodes

Parallel Programming Models

- **Distributed memory systems**
 - For processors to share data, the programmer must explicitly arrange for communication - “Message Passing”
 - Message passing libraries:
 - **MPI (“Message Passing Interface”)**
 - **PVM (“Parallel Virtual Machine”)**
 - **Shmem (Cray/SGI only)**
- **Shared memory systems**
 - “Thread” based programming
 - Compiler directives (OpenMP; various proprietary systems)
 - Can also do explicit message passing, of course

Parallel Computing: Hardware

- **In very good shape!**
- **Processors are cheap and powerful**
 - Intel, Alpha, MIPS, ...
 - Theoretical performance approaching 1 GFLOP/sec
- **SMP nodes with 1-8 CPUs are common**
- **Clusters with 32 nodes are common**
- **Affordable, high-performance interconnect technology is becoming available - Beowulf clusters!**
- **Systems with a few hundreds of processors and good inter-processor communication are not hard to build**

Parallel Computing: Software

- **Not as mature as the hardware**
- **The main obstacle to making use of all this power**
 - Perceived difficulties with writing parallel codes outweigh the benefits
- **Emergence of standards is helping enormously**
 - MPI
 - OpenMP
- **Programming in a shared memory environment generally easier**
- **Often better performance using message passing**
 - Much like assembly language vs. C/Fortran

Brief History of MPI

- [What is MPI](#)
- [MPI Forum](#)
- [Goals and Scope of MPI](#)
- [MPI on OSC Parallel Platforms](#)

What Is MPI

- **M**essage **P**assing **I**nterface
- What is the message?

DATA

- Allows data to be passed between processes in a distributed memory environment

MPI Forum

- **First message-passing interface standard**
 - Successor to PVM
- **Sixty people from forty different organizations**
- **International representation**
- **MPI 1.1 Standard developed from 92-94**
- **MPI 2.0 Standard developed from 95-97**
- **Standards documents**
 - <http://www.mcs.anl.gov/mpi/index.html>
 - <http://www.mpi-forum.org/docs/docs.html> (postscript versions)

Goals and Scope of MPI

- **MPI's prime goals are:**
 - To provide source-code portability
 - To allow efficient implementation
- **It also offers:**
 - A great deal of functionality
 - Support for heterogeneous parallel architectures

Acknowledgments

- **Edinburgh Parallel Computing Centre/University of Edinburgh** for material on which this course is based
- **Dr. David Ennis** of the **Ohio Supercomputer Center** who initially developed this course

MPI on OSC Parallel Platforms

- **SGI Origin 2000**
 - Vendor implementation of MPI
 - 32 MIPS R12000 processors @ 300 MHz
- **Intel PIII-550MHz Beowulf cluster**
 - MPICH implementation
 - 128 Intel Pentium III/Xeon @ 550MHz processors
- **SGI O2 workstation cluster**
 - Vendor implementation of MPI
 - 12 SGI O2 nodes (R10000 MIPS processor)
 - 6 SGI Indy nodes (R4000 MIPS processor)

MPI Program Structure

- [Handles](#)
- [MPI Communicator](#)
- [MPI Comm world](#)
- [Header files](#)
- [MPI function format](#)
- [Initializing MPI](#)
- [Communicator Size](#)
- [Process Rank](#)
- [Exiting MPI](#)

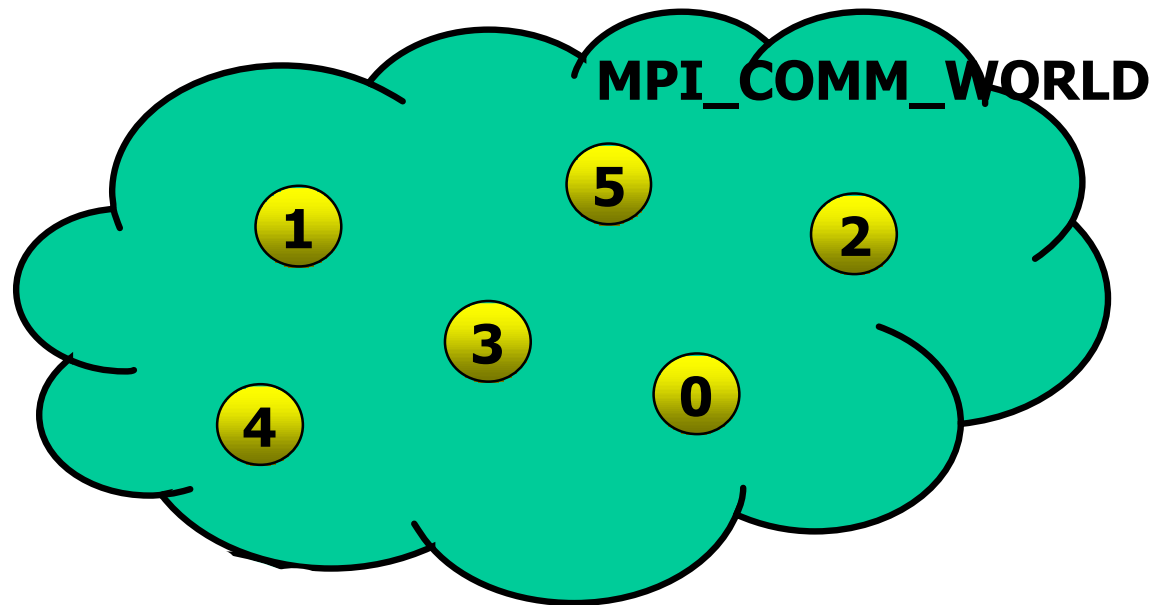
Handles

- MPI controls its own internal data structures
- MPI releases “handles” to allow programmers to refer to these
- C handles are of defined `typedefs`
- In Fortran, all handles have type `INTEGER`

MPI Communicator

- Programmer view: **group of processes** that are allowed to communicate with each other
- All MPI communication calls have a communicator argument
- Most often you will use `MPI_COMM_WORLD`
 - Defined when you call `MPI_Init`
 - It is all of your processors...

MPI_COMM_WORLD Communicator



Header Files

- MPI constants and handles are defined here

C:

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

Fortran:

```
include 'mpif.h'
```

MPI Function Format

C:

```
error = MPI_Xxxxx(parameter,...);
```

```
MPI_Xxxxx(parameter,...);
```

Fortran:

```
CALL MPI_XXXXX(parameter,...,IERROR)
```

Initializing MPI

- Must be the **first** routine called (only once)

C:

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

Fortran:

```
INTEGER IERROR
```

```
CALL MPI_INIT(IERROR)
```

Communicator Size

- How many processes are contained within a communicator?

C:

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

Fortran:

```
INTEGER COMM, SIZE, IERROR
```

```
CALL MPI_COMM_SIZE(COMM, SIZE, IERROR)
```

Process Rank

- **Process ID number within the communicator**
 - Starts with zero and goes to (n-1) where n is the number of processes requested
- **Used to identify the source and destination of messages**

C:

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

Fortran:

```
INTEGER COMM, RANK, IERROR
```

```
CALL MPI_COMM_RANK(COMM, RANK, IERROR)
```


Exiting MPI

- Must be called **last** by “all” processes

C:

```
MPI_Finalize()
```

Fortran:

```
CALL MPI_FINALIZE(IERROR)
```

Bones.c

```
#include <mpi.h>

void 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);

    /* ... your code here ... */

    MPI_Finalize ();
}
```

Bones.f

```
PROGRAM skeleton
INCLUDE 'mpif.h'
INTEGER ierror, rank, size
CALL MPI_INIT(ierror)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)
```

C ... your code here ...

```
CALL MPI_FINALIZE(ierror)
END
```

What's in a Message

- [Messages](#)
- [MPI Basic Datatypes - C](#)
- [MPI Basic Datatypes - Fortran](#)
- [Rules and Rationale](#)

Messages

- **A message contains an array of elements of some particular MPI datatype**
- **MPI Datatypes:**
 - Basic types
 - Derived types
- **Derived types can be build up from basic types**
- **C types are different from Fortran types**

MPI Basic Datatypes - 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	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

MPI Basic Datatypes - Fortran

MPI Datatype	Fortran Datatype
<code>MPI_INTEGER</code>	<code>INTEGER</code>
<code>MPI_REAL</code>	<code>REAL</code>
<code>MPI_DOUBLE_PRECISION</code>	<code>DOUBLE PRECISION</code>
<code>MPI_COMPLEX</code>	<code>COMPLEX</code>
<code>MPI_LOGICAL</code>	<code>LOGICAL</code>
<code>MPI_CHARACTER</code>	<code>CHARACTER(1)</code>
<code>MPI_BYTE</code>	
<code>MPI_PACKED</code>	

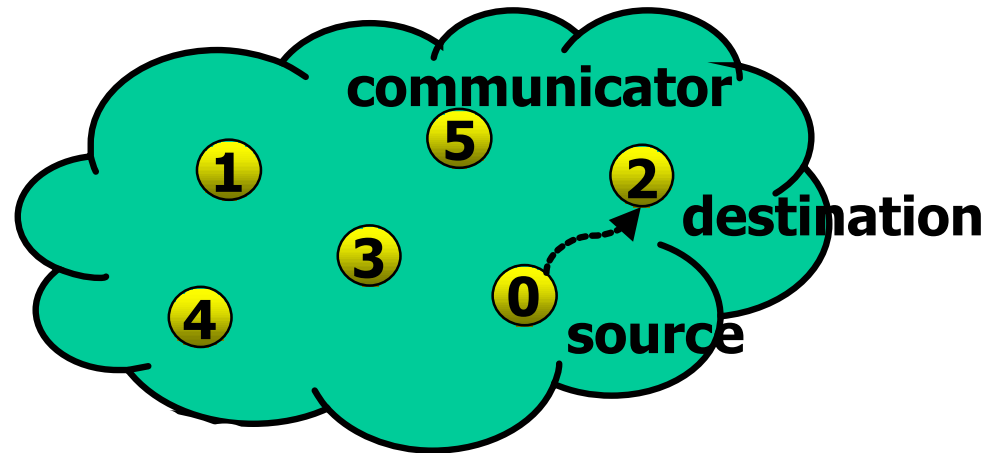
Rules and Rationale

- Programmer declares variables to have “normal” C/Fortran type, but uses matching MPI datatypes as arguments in MPI routines
- Mechanism to handle type conversion in a heterogeneous collection of machines
- **General rule: MPI datatype specified in a receive must match the MPI datatype specified in the send**

Point-to-Point Communications

- [Definitions](#)
- [Communication Modes](#)
- [Routine Names \(blocking\)](#)
- [Sending a Message](#)
- [Memory Mapping](#)
- [Synchronous Send](#)
- [Buffered Send](#)
- [Standard Send](#)
- [Ready Send](#)
- [Receiving a Message](#)
- [Wildcarding](#)
- [Getting Wildcarded Information](#)
- [Received Message Count](#)
- [Message Order Preservation](#)
- [Sample Programs](#)
- [Timers](#)

Point-to-Point Communication



- Communication between two processes
- **Source** process *sends* message to destination process
- Destination process *receives* the message
- Communication takes place within a communicator
- Destination process is identified by its rank in the communicator

Definitions

- **“Completion” of the communication means that memory locations used in the message transfer can be safely accessed**
 - Send: variable sent can be reused after completion
 - Receive: variable received can now be used
- **MPI communication modes differ in what conditions are needed for completion**
- **Communication modes can be blocking or non-blocking**
 - Blocking: return from routine implies completion
 - Non-blocking: routine returns immediately, user must test for completion

Communication Modes

Mode	Completion Condition
Synchronous send	Only completes when the receive has completed
Buffered send	Always completes (unless an error occurs), irrespective of receiver
Standard send	Message sent (receive state unknown)
Ready send	Always completes (unless an error occurs), irrespective of whether the receive has completed
Receive	Completes when a message has arrived

Routine Names (blocking)

MODE	MPI CALL
Standard send	MPI_SEND
Synchronous send	MPI_SSEND
Buffered send	MPI_BSEND
Ready send	MPI_RSEND
Receive	MPI_RECV

Sending a Message

C:

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

Fortran:

```
<type>      BUF(*)  
INTEGER     COUNT, DATATYPE, DEST, TAG  
INTEGER     COMM, IERROR
```

```
CALL MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
```

Arguments

<code>buf</code>	starting <i>address</i> of the data to be sent
<code>count</code>	number of elements to be sent
<code>datatype</code>	MPI datatype of each element
<code>dest</code>	rank of destination process
<code>tag</code>	message marker (set by user)
<code>comm</code>	MPI communicator of processors involved

```
MPI_SEND(data, 500, MPI_REAL, 6, 33, MPI_COMM_WORLD, IERROR)
```

Memory Mapping

The Fortran 2-D array

1,1	1,2	1,3
2,1	2,2	2,3
3,1	3,2	3,3



Is stored in memory

1,1
2,1
3,1
1,2
2,2
3,2
1,3
2,3
3,3

Synchronous Send

- **Completion criteria:**
Completes when message has been received
- **Use if need to know that message has been received**
- **Sending & receiving processes synchronize**
 - regardless of who is faster
 - processor idle time is probable
- **Safest communication method**

Buffered Send

- **Completion criteria:**
Completes when message copied to buffer
- **Advantage: Completes immediately**
- **Disadvantage: User cannot assume there is a pre-allocated buffer**
- **Control your own buffer space using MPI routines**
MPI_Buffer_attach
MPI_Buffer_detach

Standard Send

- **Completion criteria:**
Unknown!
- **May or may not imply that message has arrived at destination**
- **Don't make any assumptions (implementation dependent)**

Ready Send

- **Completion criteria:**
Completes immediately, but only successful if matching receive already posted
- **Advantage: Completes immediately**
- **Disadvantage: User must synchronize processors so that receiver is ready**
- **Potential for good performance**

Receiving a Message

C:

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

Fortran:

```
<type>          BUF(*)
INTEGER         COUNT, DATATYPE, DEST, TAG
INTEGER         COMM, STATUS(MPI_STATUS_SIZE), IERROR
```

```
CALL MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)
```

For a communication to succeed

- Sender must specify a valid destination rank
- Receiver must specify a valid source rank
- The communicator must be the same
- Tags must match
- Receiver's buffer must be large enough

Wildcarding

- Receiver can wildcard
- To receive from any source

`MPI_ANY_SOURCE`

To receive with any tag

`MPI_ANY_TAG`

- Actual source and tag are returned in the receiver's status parameter

Using the Status Handle

- Information from a wildcarded receive is returned from `MPI_RECV` in `status handle`

Information	C	Fortran
source	<code>status.MPI_SOURCE</code>	<code>status(MPI_SOURCE)</code>
tag	<code>status.MPI_TAG</code>	<code>status(MPI_TAG)</code>
count	<code>MPI_Get_count</code>	<code>MPI_GET_COUNT</code>

Received Message Count

- Message received may not fill receive buffer
- `count` is number of elements actually received

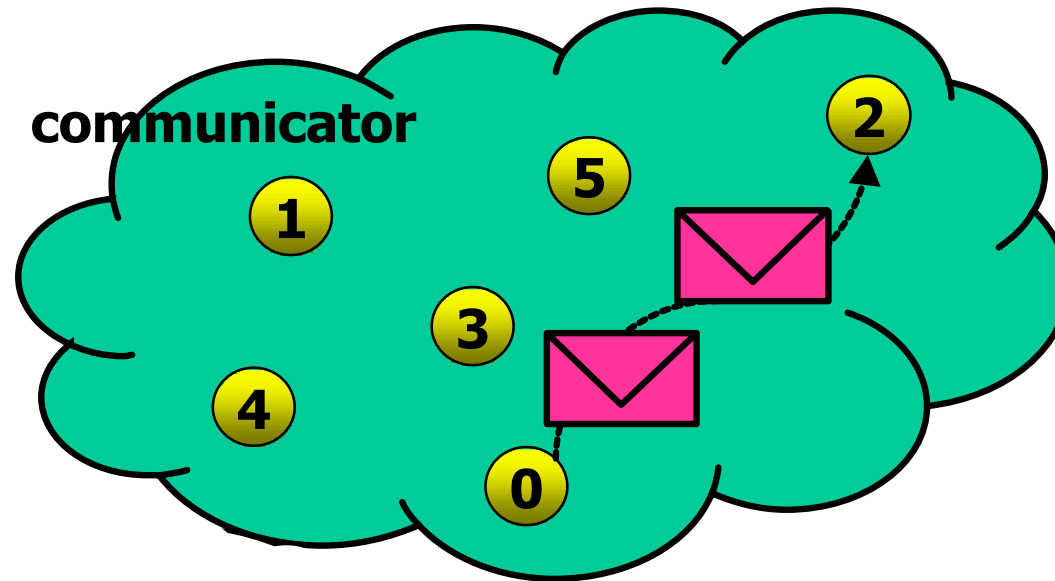
C:

```
int MPI_Get_count (MPI_Status *status,  
                  MPI_Datatype datatype, int *count)
```

Fortran:

```
INTEGER          STATUS(MPI_STATUS_SIZE), DATATYPE  
INTEGER          COUNT, IERROR  
  
CALL MPI_GET_COUNT(STATUS, DATATYPE, COUNT, IERROR)
```

Message Order Preservation



- Messages do not overtake each other
- Example: Process 0 sends two messages
Process 2 posts two receives that match either message
Order preserved

Sample Program #1 - C

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
/* Run with two processes */
void main(int argc, char *argv[]) {
    int rank, i, count;
    float data[100],value[200];
    MPI_Status status;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    if(rank==1) {
        for(i=0;i<100;++i) data[i]=i;
        MPI_Send(data,100,MPI_FLOAT,0,55,MPI_COMM_WORLD);
    } else {
        MPI_Recv(value,200,MPI_FLOAT,MPI_ANY_SOURCE,55,MPI_COMM_WORLD,&status);
        printf("P:%d Got data from processor %d \n",rank, status.MPI_SOURCE);
        MPI_Get_count(&status,MPI_FLOAT,&count);
        printf("P:%d Got %d elements \n",rank,count);
        printf("P:%d value[5]=%f \n",rank,value[5]);
    }
    MPI_Finalize();
}
```

```
P: 0 Got data from processor 1
P: 0 Got 100 elements
P: 0 value[5]=5.000000
```

Sample Program #1 - Fortran

```
PROGRAM p2p
C Run with two processes
  INCLUDE 'mpif.h'
  INTEGER err, rank, size
  real data(100), value(200)
  integer status(MPI_STATUS_SIZE)
  integer count
  CALL MPI_INIT(err)
  CALL MPI_COMM_RANK(MPI_COMM_WORLD,rank,err)
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD,size,err)
  if (rank.eq.1) then
    data=3.0
    call MPI_SEND(data,100,MPI_REAL,0,55,MPI_COMM_WORLD,err)
  else
    call MPI_RECV(value,200,MPI_REAL,MPI_ANY_SOURCE,55,
&                MPI_COMM_WORLD,status,err)
    print *, "P:",rank," got data from processor ",
&           status(MPI_SOURCE)
    call MPI_GET_COUNT(status,MPI_REAL,count,err)
    print *, "P:",rank," got ",count," elements"
    print *, "P:",rank," value(5)=",value(5)
  end if
  CALL MPI_FINALIZE(err)
END
```

```
P: 0 Got data from processor 1
P: 0 Got 100 elements
P: 0 value[5]=3.
```

Timers

- Time is measured in seconds
- Time to perform a task is measured by consulting the timer before and after

C:

```
double MPI_Wtime(void);
```

Fortran:

```
DOUBLE PRECISION MPI_WTIME()
```

Non-Blocking Communication

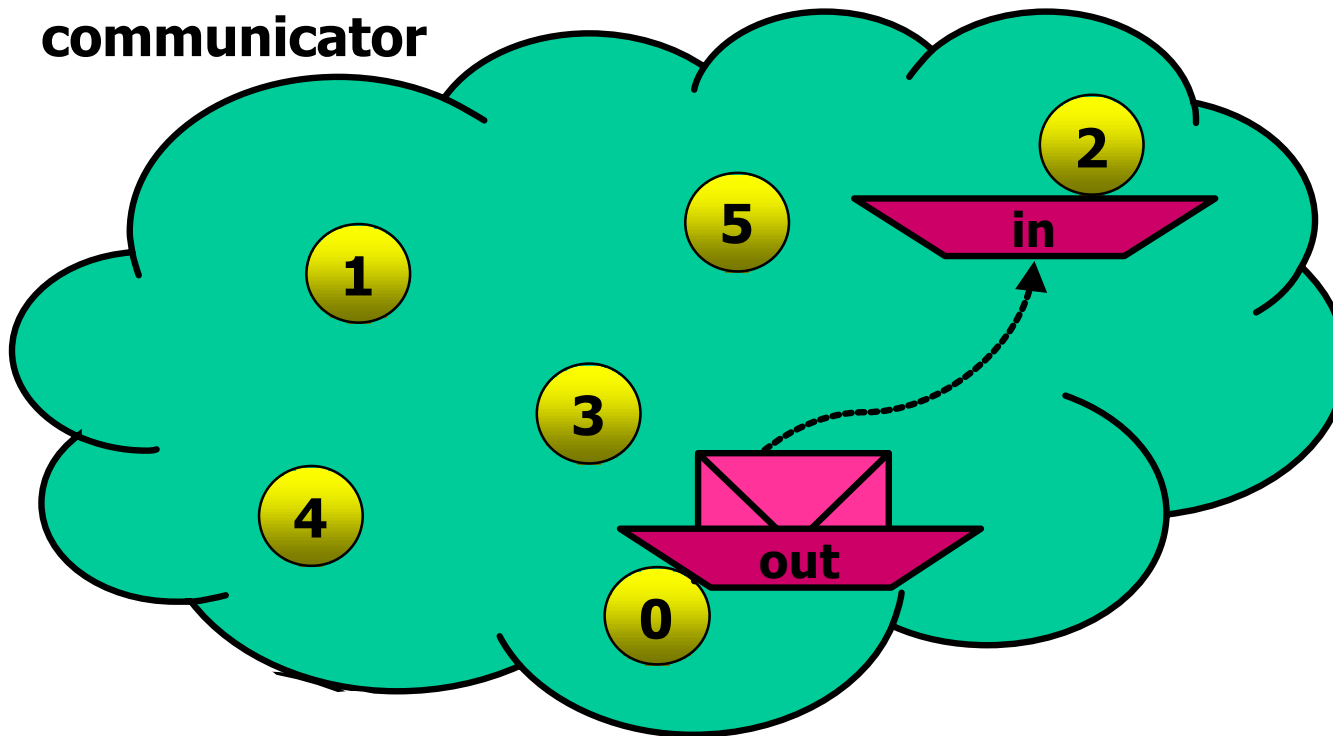
- [Non-Blocking Communications](#)
- [Non-Blocking Send](#)
- [Non-Blocking Receive](#)
- [Handles](#)
- [Non-Blocking Synchronous Send](#)
- [Non-Blocking Receive](#)
- [Blocking and Non-Blocking](#)
- [Routine Names](#)
- [Completion Tests](#)
- [Wait/Test Routines](#)
- [Multiple Communications](#)
- [Testing Multiple Non-Blocking Communications](#)

Non-Blocking Communications

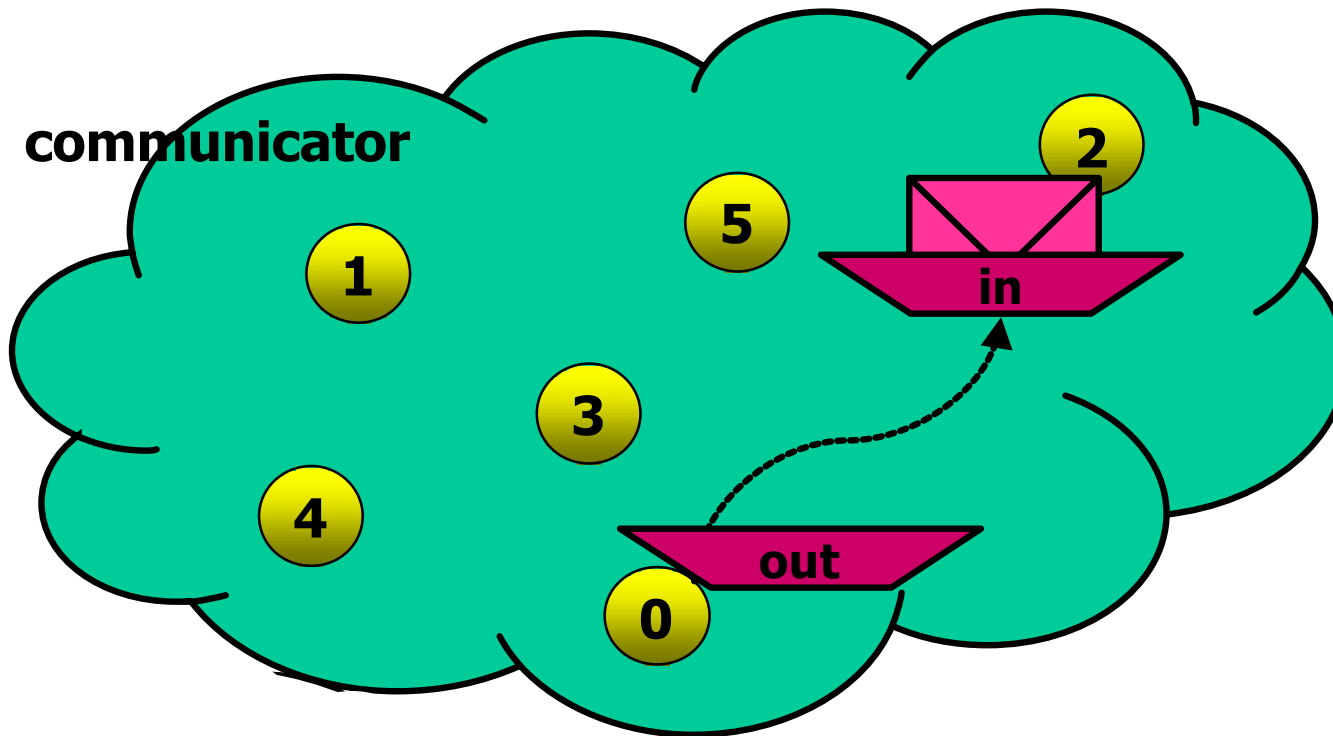
Separate communication into three phases:

1. **Initiate non-blocking communication (“post” a send or receive)**
2. **Do some other work not involving the data in transfer**
 - Overlap calculation and communication
 - Latency hiding
3. **Wait for non-blocking communication to complete**

Non-Blocking Send



Non-Blocking Receive



Handles Used For Non-Blocking Communication

Datatype	Same as for blocking (MPI_Datatype or INTEGER)
Communicator	Same as for blocking (MPI_Comm or INTEGER)
Request	MPI_Request or INTEGER

- **A request handle is allocated when a non-blocking communication is initiated**
- **The request handle is used for testing if a specific communication has completed**

Non-Blocking Synchronous Send

C:

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

Fortran:

```
<type>          BUF(*)  
INTEGER         COUNT,DATATYPE,DEST,TAG,COMM  
INTEGER         REQUEST,IERROR  
  
CALL MPI_ISEND(BUF,COUNT,DATATYPE,DEST,TAG,COMM,REQUEST,IERROR)
```

Non-Blocking Receive

C:

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

Fortran:

```
<type>          BUF(*)  
INTEGER         COUNT, DATATYPE, SOURCE, TAG, COMM  
INTEGER         REQUEST, IERROR
```

```
CALL MPI_Irecv(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
```

Note: There is no **STATUS** argument.

Blocking and Non-Blocking

- Send and receive can be blocking or non-blocking
- A blocking send can be used with a non-blocking receive, and vice-versa
- Non-blocking sends can use any mode -- synchronous, buffered, standard, or ready

Note: There is no advantage for buffered or ready modes.

Routine Names

Non-Blocking Operation	MPI Call
Standard send	<code>MPI_ISEND</code>
Synchronous send	<code>MPI_ISSEND</code>
Buffered send	<code>MPI_IBSEND</code>
Ready send	<code>MPI_IRSEND</code>
Receive	<code>MPI_IRECV</code>

Completion Tests

- **Waiting vs. Testing**

wait routine does not return until completion finished

test routine returns a TRUE or FALSE value
depending on whether or not the communication
has completed

Wait/Test Routines

C:

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

Fortran:

```
INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERR
LOGICAL FLAG

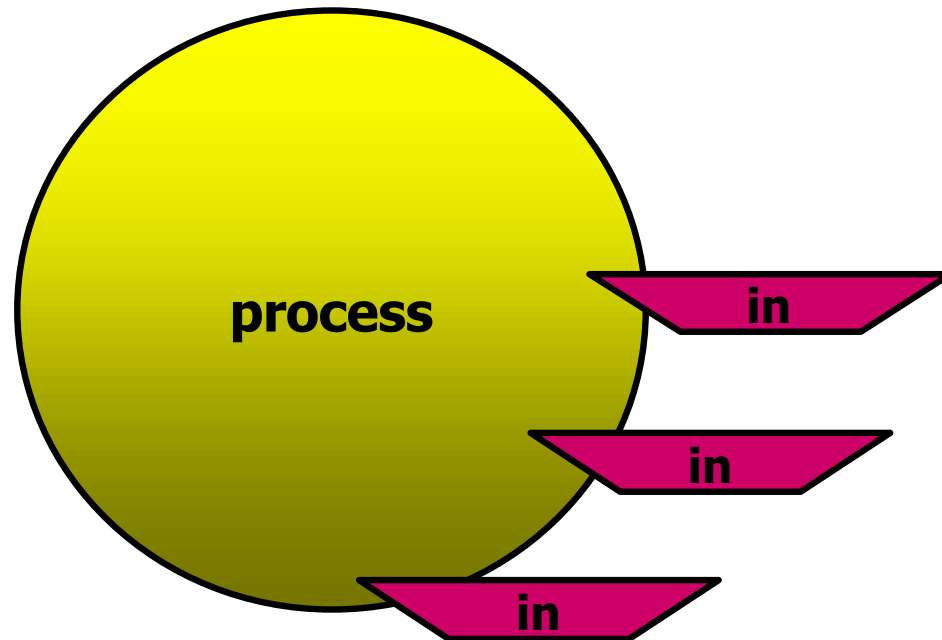
CALL MPI_WAIT(REQUEST, STATUS, IERR)
CALL MPI_TEST(REQUEST, FLAG, STATUS, IERR)
```

Note: Here is where **STATUS** appears.

Multiple Communications

- Test or wait for completion of **one** (and only one) message
MPI_Waitany
MPI_Testany
- Test or wait for completion of **all** messages
MPI_Waitall
MPI_Testall
- Test or wait for completion of **as many messages as possible**
MPI_Waitsome
MPI_Testsome

Testing Multiple Non-Blocking Communications



Collective Communication

- [Collective Communication](#)
 - [Barrier Synchronization](#)
 - [Broadcast*](#)
 - [Scatter*](#)
 - [Gather](#)
 - [Gather/Scatter Variations](#)
 - [Summary Illustration](#)
 - [Global Reduction Operations](#)
 - [Predefined Reduction Operations](#)
 - [MPI Reduce](#)
 - [Minloc and Maxloc*](#)
 - [User-defined Reduction Operators](#)
 - [Reduction Operator Functions](#)
 - [Registering a User-defined Reduction Operator*](#)
 - [Variants of MPI Reduce](#)
- *includes sample C and Fortran programs

Collective Communication

- Communications involving a group of processes
- Called by *all* processes in a communicator
- Examples:
 - Broadcast, scatter, gather (Data Distribution)
 - Global sum, global maximum, etc. (Collective Operations)
 - Barrier synchronization

Characteristics of Collective Communication

- **Collective communication will not interfere with point-to-point communication and vice-versa**
- **All processes must call the collective routine**
- **Synchronization not guaranteed (except for barrier)**
- **No non-blocking collective communication**
- **No tags**
- **Receive buffers must be exactly the right size**

Barrier Synchronization

- **Red** light for each processor: turns **green** when all processors have arrived
- Slower than hardware barriers (example: Cray T3E)

C:

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

Fortran:

```
INTEGER COMM, IERROR
```

```
CALL MPI_BARRIER (COMM, IERROR)
```

Broadcast

- **One-to-all communication: same data sent from root process to all the others in the communicator**

- **C:**

```
int MPI_Bcast (void *buffer, int, count,  
              MPI_Datatype datatype, int root, MPI_Comm comm)
```

- **Fortran:**

```
<type> BUFFER (*)  
INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR  
  
MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM IERROR)
```

- **All processes must specify same root rank and communicator**

Sample Program #5 - C

```
#include<mpi.h>
void main (int argc, char *argv[]) {
    int rank;
    double param;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    if(rank==5) param=23.0;
    MPI_Bcast(&param,1,MPI_DOUBLE,5,MPI_COMM_WORLD);
    printf("P:%d after broadcast parameter is %f\n",rank,param);
    MPI_Finalize();
}
```

```
P:0 after broadcast parameter is 23.000000
P:6 after broadcast parameter is 23.000000
P:5 after broadcast parameter is 23.000000
P:2 after broadcast parameter is 23.000000
P:3 after broadcast parameter is 23.000000
P:7 after broadcast parameter is 23.000000
P:1 after broadcast parameter is 23.000000
P:4 after broadcast parameter is 23.000000
```


Sample Program #5 - Fortran

```
PROGRAM broadcast
INCLUDE 'mpif.h'
INTEGER err, rank, size
real param
CALL MPI_INIT(err)
CALL MPI_COMM_RANK(MPI_WORLD_COMM,rank,err)
CALL MPI_COMM_SIZE(MPI_WORLD_COMM,size,err)
if(rank.eq.5) param=23.0
call MPI_BCAST(param,1,MPI_REAL,5,MPI_COMM_WORLD,err)
print *, "P:",rank, " after broadcast param is ",param
CALL MPI_FINALIZE(err)
END
```

```
P:1 after broadcast parameter is 23.
P:3 after broadcast parameter is 23.
P:4 after broadcast parameter is 23
P:0 after broadcast parameter is 23
P:5 after broadcast parameter is 23.
P:6 after broadcast parameter is 23.
P:7 after broadcast parameter is 23.
P:2 after broadcast parameter is 23.
```

Scatter

- **One-to-all communication: different data sent to each process in the communicator (in rank order)**

C:

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

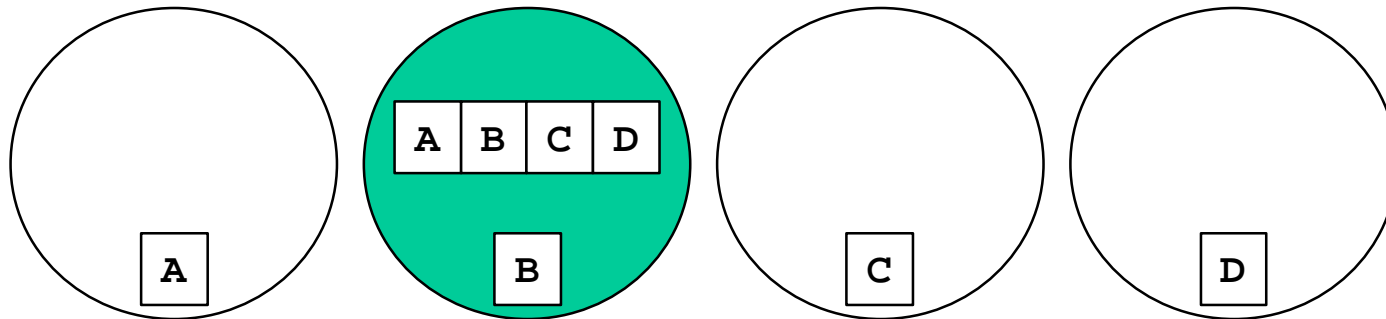
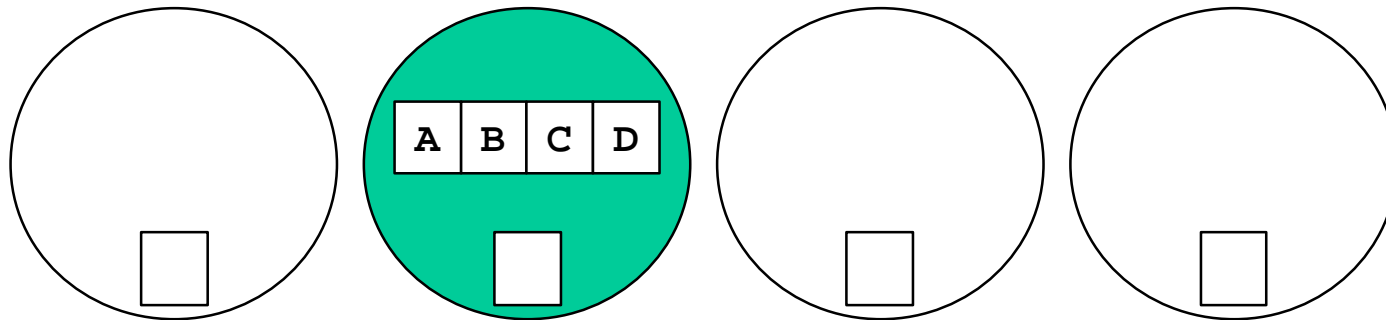
Fortran:

```
<type> SENDBUF(*), RECVBUF(*)
```

```
CALL MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
                RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR)
```

- **sendcount is the number of elements sent to each process, not the “total” number sent**
 - send arguments are significant only at the root process
-

Scatter Example



rank

0

1

2

3

Sample Program #6 - C

```
#include <mpi.h>
void main (int argc, char *argv[]) {
    int rank,size,i,j;
    double param[4],mine;
    int sndcnt,revcnt;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    revcnt=1;
    if(rank==3){
        for(i=0;i<4;i++) param[i]=23.0+i;
        sndcnt=1;
    }

    MPI_Scatter(param,sndcnt,MPI_DOUBLE,&mine,revcnt,MPI_DOUBLE,3,MPI_COMM_
WORLD);
    printf("P:%d mine is %f\n",rank,mine);
    MPI_Finalize();
}
```

```
P:0 mine is 23.000000
P:1 mine is 24.000000
P:2 mine is 25.000000
P:3 mine is 26.000000
```

Sample Program #6 - Fortran

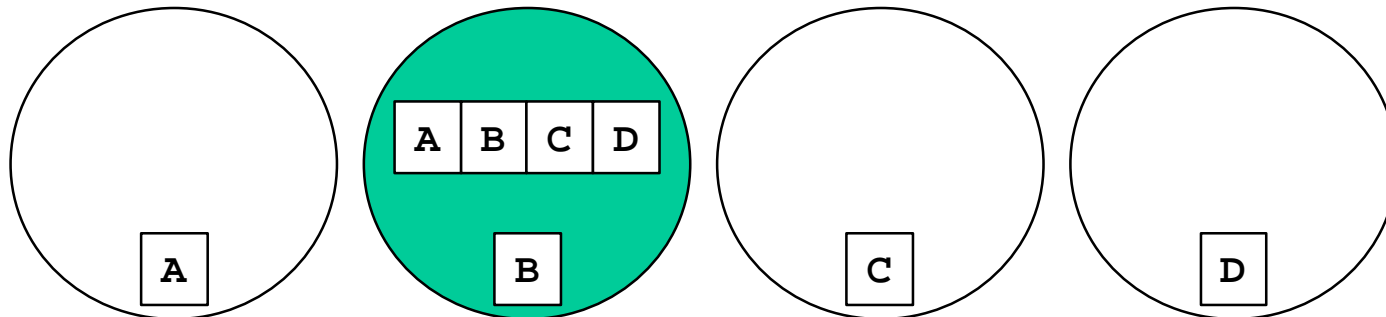
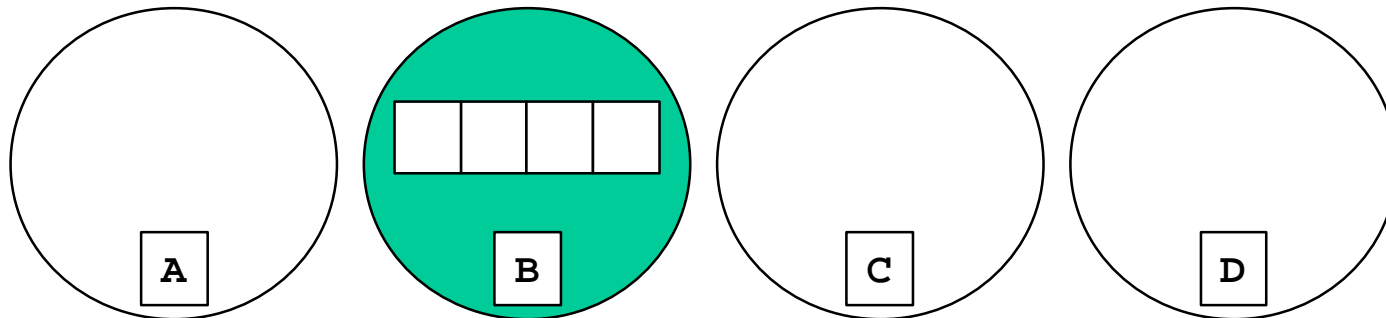
```
PROGRAM scatter
INCLUDE 'mpif.h'
INTEGER err, rank, size
real param(4), mine
integer sndcnt,rcvcnt
CALL MPI_INIT(err)
CALL MPI_COMM_RANK(MPI_WORLD_COMM,rank,err)
CALL MPI_COMM_SIZE(MPI_WORLD_COMM,size,err)
rcvcnt=1
if(rank.eq.3) then
  do i=1,4
    param(i)=23.0+i
  end do
  sndcnt=1
end if
call MPI_SCATTER(param,sndcnt,MPI_REAL,mine,rcvcnt,MPI_REAL,
&                3,MPI_COMM_WORLD,err)
print *,"P:",rank," mine is ",mine
CALL MPI_FINALIZE(err)
END
```

```
P:1 mine is 25.
P:3 mine is 27.
P:0 mine is 24.
P:2 mine is 26.
```

Gather

- **All-to-one communication: different data collected by root process**
 - Collection done in rank order
- **MPI_GATHER & MPI_Gather have same arguments as matching scatter routines**
- **Receive arguments only meaningful at the root process**

Gather Example



rank

0

1

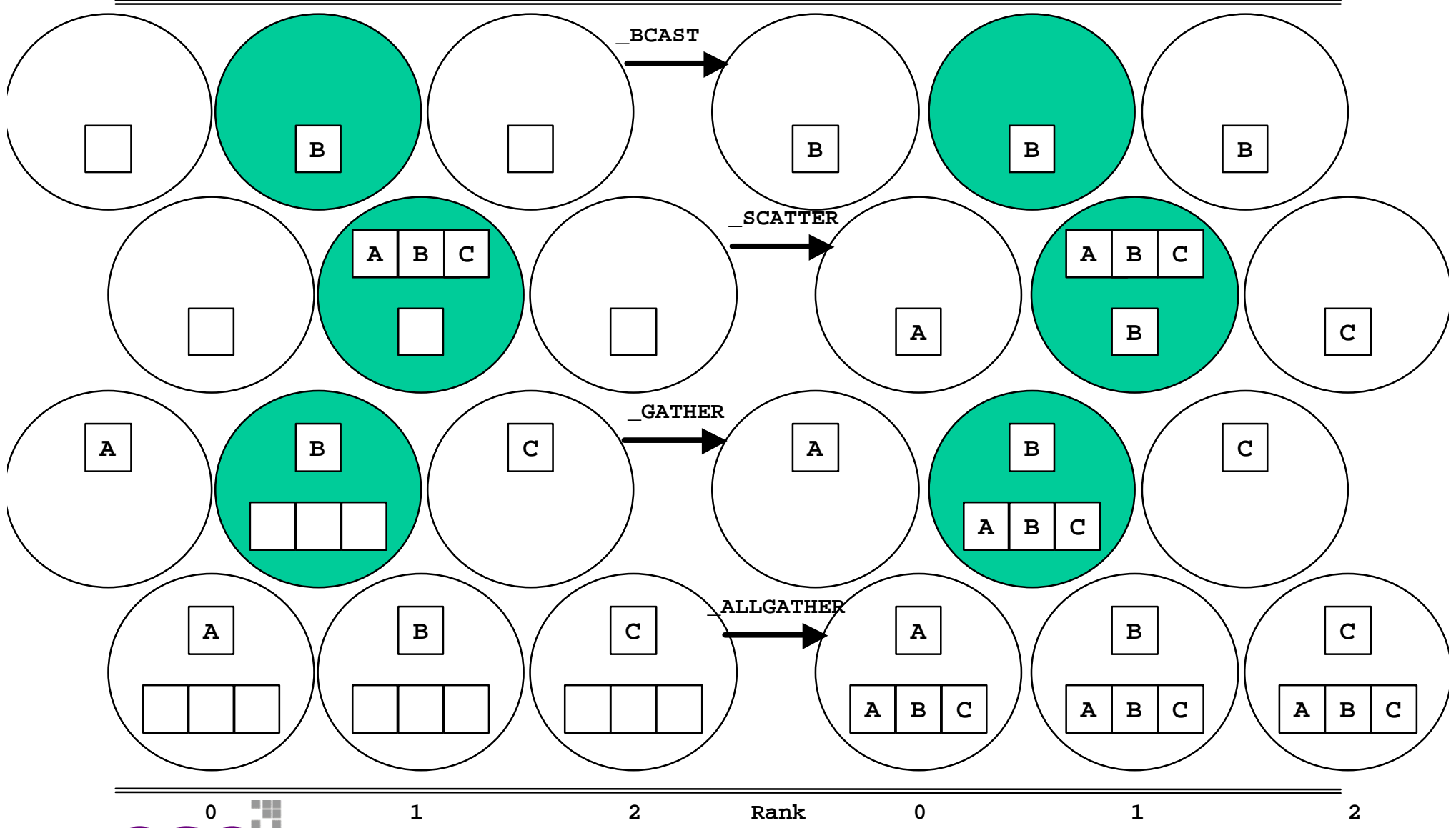
2

3

Gather/Scatter Variations

- `MPI_Allgather`
- `MPI_Alltoall`
- **No root process specified: all processes get gathered or scattered data**
- **Send and receive arguments significant for all processes**

Summary



Global Reduction Operations

- **Used to compute a result involving data distributed over a group of processes**
- **Examples:**
 - Global sum or product
 - Global maximum or minimum
 - Global user-defined operation

Example of a Global Sum

- Sum of all the `x` values is placed in `result` only on processor 0

C:

```
MPI_Reduce(&x,&result,1, MPI_INTEGER,MPI_SUM,0,  
          MPI_COMM_WORLD)
```

Fortran:

```
CALL MPI_REDUCE(x,result,1,MPI_INTEGER,MPI_SUM,0,  
              MPI_COMM_WORLD,IERROR)
```

Predefined Reduction Operations

MPI Name	Function
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bitwise AND
MPI_LOR	Logical OR
MPI_BOR	Bitwise OR
MPI_LXOR	Logical exclusive OR
MPI_BXOR	Bitwise exclusive OR
MPI_MAXLOC	Maximum and location
MPI_MINLOC	Minimum and location

General Form

- **count** is the number of “ops” done on consecutive elements of **sendbuf** (it is also size of **recvbuf**)
- **op** is an associative operator that takes two operands of type **datatype** and returns a result of the same type

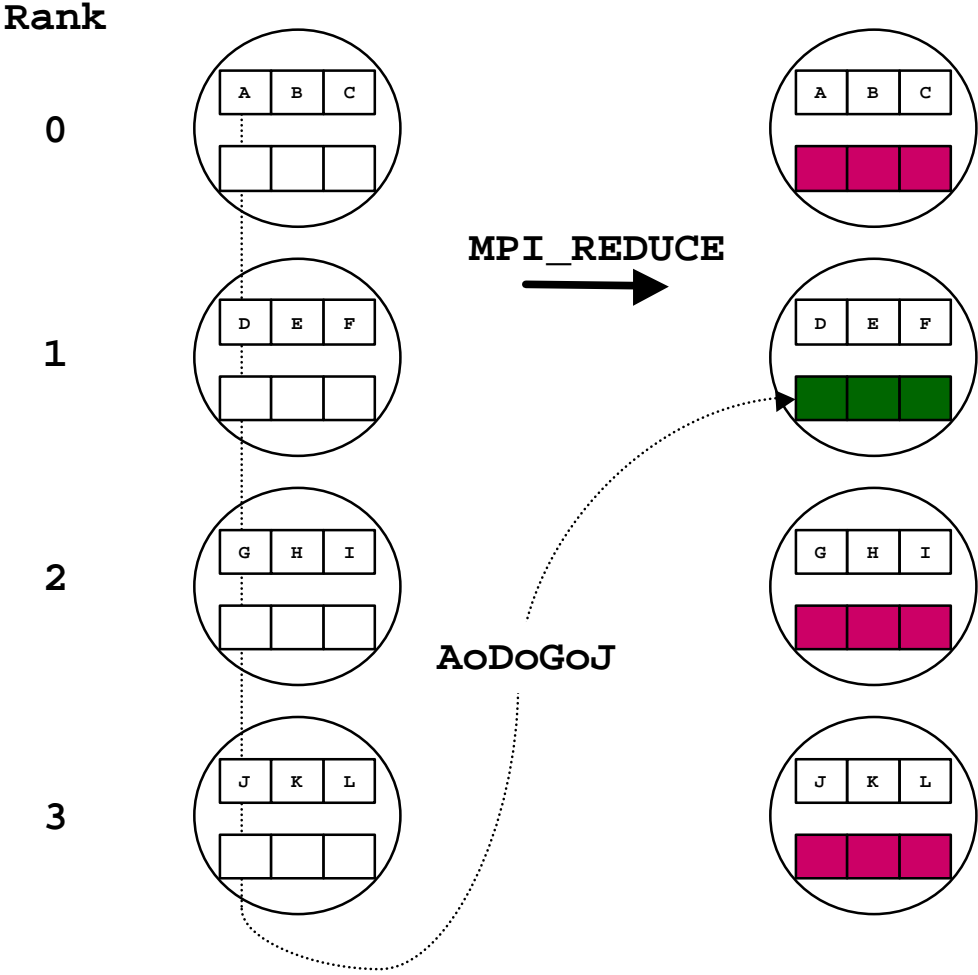
C:

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

Fortran:

```
<type> SENDBUF(*), RECVBUF(*)  
CALL MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR)
```

MPI_Reduce



Minloc and Maxloc

- **Designed to compute a global minimum/maximum and index associated with the extreme value**
 - Common application: index is the processor rank (see sample program)
- **If more than one extreme, get the first**
- **Designed to work on operands that consist of a value and index pair**
- **MPI_Datatypes include:**

C:

```
MPI_FLOAT_INT, MPI_DOUBLE_INT, MPI_LONG_INT, MPI_2INT, MPI_SHORT_INT,  
MPI_LONG_DOUBLE_INT
```

Fortran:

```
MPI_2REAL, MPI_2DOUBLEPRECISION, MPI_2INTEGER
```

Sample Program #7 - C

```
#include <mpi.h>
/* Run with 16 processes */
void main (int argc, char *argv[]) {
    int rank;
    struct {
        double value;
        int rank;
    } in, out;
    int root;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    in.value=rank+1;
    in.rank=rank;
    root=7;
    MPI_Reduce(&in,&out,1,MPI_DOUBLE_INT,MPI_MAXLOC,root,MPI_COMM_WORLD);
    if(rank==root) printf("PE:%d max=%lf at rank
%d\n",rank,out.value,out.rank);
    MPI_Reduce(&in,&out,1,MPI_DOUBLE_INT,MPI_MINLOC,root,MPI_COMM_WORLD);
    if(rank==root) printf("PE:%d min=%lf at rank
%d\n",rank,out.value,out.rank);
    MPI_Finalize();
}
```

```
P:7 max=16.000000 at rank 15
P:7 min=1.000000 at rank 0
```


Sample Program #7 - Fortran

```
PROGRAM MaxMin
C
C Run with 8 processes
C
  INCLUDE 'mpif.h'
  INTEGER err, rank, size
  integer in(2),out(2)
  CALL MPI_INIT(err)
  CALL MPI_COMM_RANK(MPI_WORLD_COMM,rank,err)
  CALL MPI_COMM_SIZE(MPI_WORLD_COMM,size,err)
  in(1)=rank+1
  in(2)=rank
  call MPI_REDUCE(in,out,1,MPI_2INTEGER,MPI_MAXLOC,
&                7,MPI_COMM_WORLD,err)
  if(rank.eq.7) print *, "P:",rank," max=",out(1)," at rank ",out(2)

  call MPI_REDUCE(in,out,1,MPI_2INTEGER,MPI_MINLOC,
&                2,MPI_COMM_WORLD,err)
  if(rank.eq.2) print *, "P:",rank," min=",out(1)," at rank ",out(2)
  CALL MPI_FINALIZE(err)
END
```

```
P:2 min=1 at rank 0
P:7 max=8 at rank 7
```

User-Defined Reduction Operators

- Reducing using an arbitrary operator `c`

C -- function of type `MPI_User_function`:

```
void my_operator (void *invec, void *inoutvec, int *len,  
                 MPI_Datatype *datatype)
```

Fortran -- function of type:

```
<type> INVEC(LEN), INOUTVEC(LEN)  
INTEGER LEN, DATATYPE
```

```
FUNCTION MY_OPERATOR (INVEC(*), INOUTVEC(*), LEN, DATATYPE)
```

Reduction Operator Functions

- Operator function for c must have syntax:

```
for (i=1 to len)
    inoutvec(i) = inoutvec(i) C invec(i)
```

- Operator c need not commute
- **inoutvec** argument acts as both a second input operand as well as the output of the function

Registering a User-Defined Reduction Operator

- Operator handles have type `MPI_Op` or `INTEGER`
- If `commute` is `TRUE`, reduction may be performed faster

C:

```
int MPI_Op_create (MPI_User_function *function,  
                  int commute, MPI_Op *op)
```

Fortran:

```
EXTERNAL FUNC  
INTEGER OP, IERROR  
LOGICAL COMMUTE
```

```
MPI_OP_CREATE (FUNC, COMMUTE, OP, IERROR)
```

Sample Program #8 - C

```
#include <mpi.h>
typedef struct {
    double real,imag;
} complex;

void cprod(complex *in, complex *inout, int *len, MPI_Datatype *dptr) {
    int i;
    complex c;
    for (i=0; i<*len; ++i) {
        c.real=(*in).real * (*inout).real - (*in).imag * (*inout).imag;
        c.imag=(*in).real * (*inout).imag + (*in).imag * (*inout).real;
        *inout=c;
        in++;
        inout++;
    }
}

void main (int argc, char *argv[]) {
    int rank;
    int root;
    complex source,result;
```

Sample Program #8 - C (cont.)

```
MPI_Op myop;
MPI_Datatype ctype;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

MPI_Type_contiguous(2, MPI_DOUBLE, &ctype);
MPI_Type_commit(&ctype);
MPI_Op_create(cprod, TRUE, &myop);
root=2;
source.real=rank+1;
source.imag=rank+2;
MPI_Reduce(&source, &result, 1, ctype, myop, root, MPI_COMM_WORLD);
if(rank==root) printf ("PE:%d result is %lf + %lfi\n", rank,
result.real, result.imag);
MPI_Finalize();
}
```

```
P:2 result is -185.000000 + -180.000000i
```

Sample Program #8 - Fortran

```
PROGRAM UserOP
  INCLUDE 'mpif.h'
  INTEGER err, rank, size
  integer source, reslt
  external digit
  logical commute
  integer myop
  CALL MPI_INIT(err)
  CALL MPI_COMM_RANK(MPI_WORLD_COMM,rank,err)
  CALL MPI_COMM_SIZE(MPI_WORLD_COMM,size,err)
  commute=.true.
  call MPI_OP_CREATE(digit,commute,myop,err)
  source=(rank+1)**2
  call MPI_BARRIER(MPI_COM_WORLD,err)
  call MPI_SCAN(source,reslt,1,MPI_INTEGER,myop,MPI_COMM_WORLD,err)
  print *,"P:",rank," my result is ",reslt
  CALL MPI_FINALIZE(err)
END

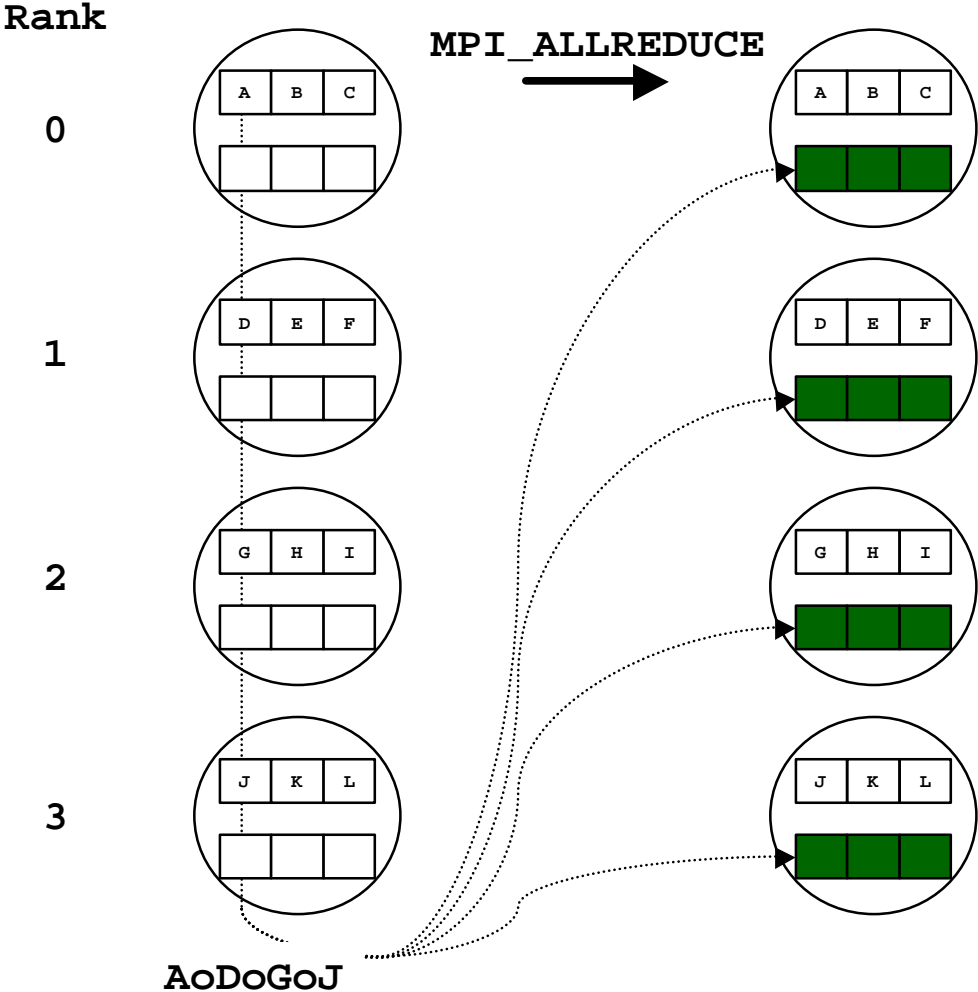
integer function digit(in,inout,len,type)
  integer in(len),inout(len)
  integer len,type
  do i=1,len
    inout(i)=mod((in(i)+inout(i)),10)
  end do
  digit = 5
end
```

```
P:6 my result is 0
P:5 my result is 1
P:7 my result is 4
P:1 my result is 5
P:3 my result is 0
P:2 my result is 4
P:4 my result is 5
P:0 my result is 1
```

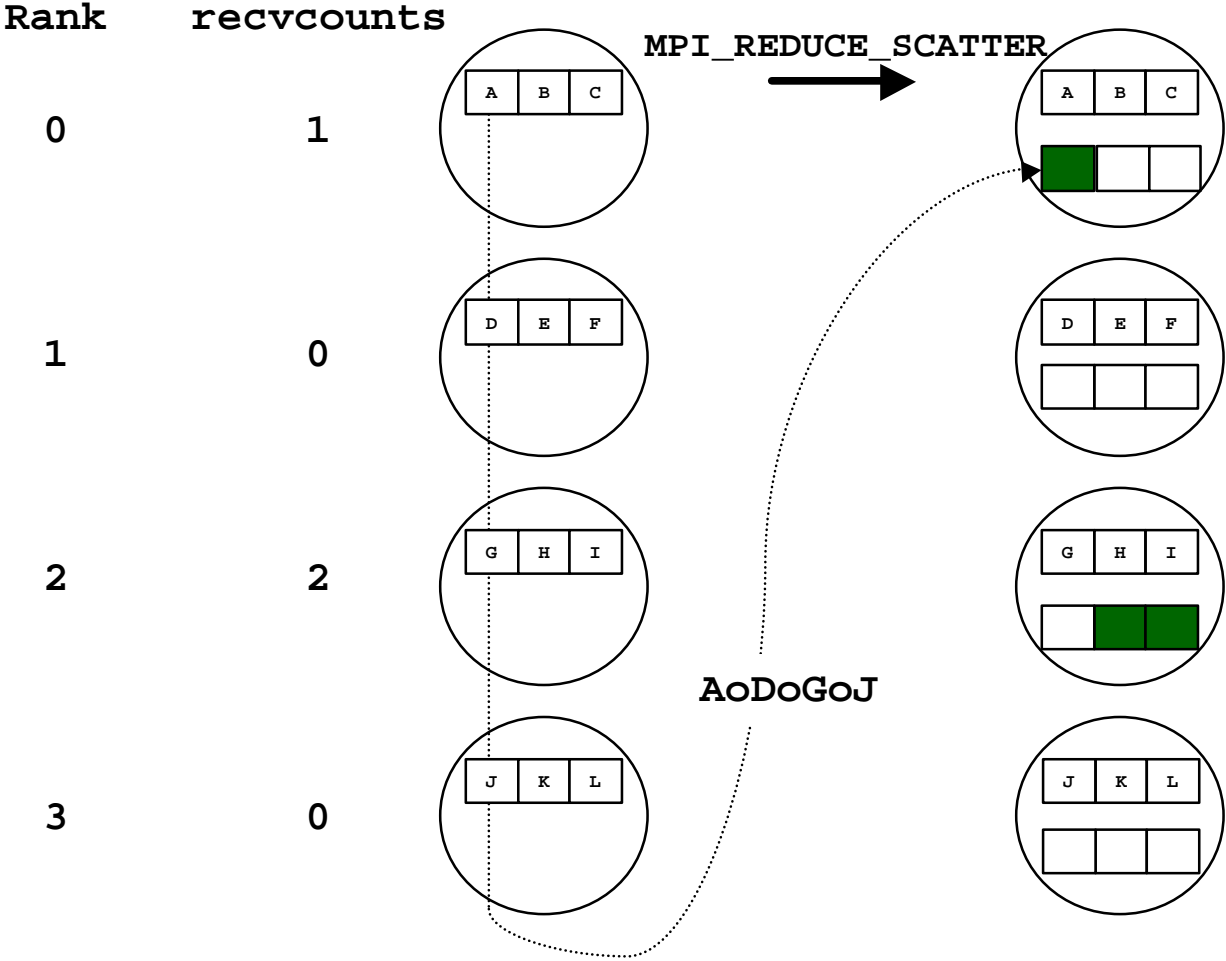
Variants of MPI_REDUCE

- **MPI_ALLREDUCE** -- no root process (all get results)
- **MPI_REDUCE_SCATTER** -- multiple results are scattered
- **MPI_SCAN** -- “parallel prefix”

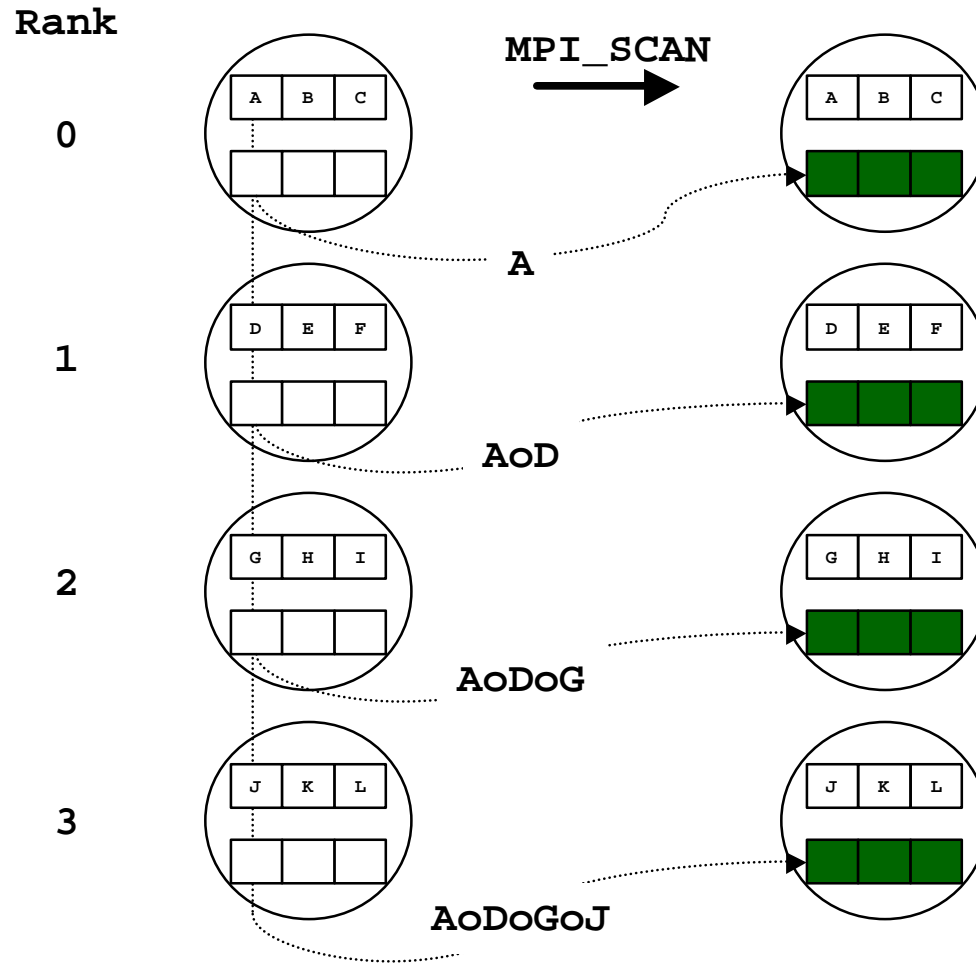
MPI_ALLREDUCE



MPI_REDUCE_SCATTER



MPI_SCAN



Derived Datatypes

- [MPI Datatypes](#)
- [Procedure](#)
- [Datatype Construction](#)
- [Type Maps](#)
- [Contiguous Datatype*](#)
- [Vector Datatype*](#)
- [Extent of a Datatype](#)
- [Structure Datatype*](#)
- [Committing a Datatype](#)

*includes sample C and Fortran programs

MPI Datatypes

- **Basic types**
- **Derived types**
 - Constructed from existing types (basic and derived)
 - Used in MPI communication routines to transfer high-level, extensive data entities
- **Examples:**
 - Sub-arrays or “unnatural” array memory striding
 - C structures and Fortran common blocks
 - Large set of general variables
- **Alternative to repeated sends of varied basic types**
 - Slow, clumsy, and error prone

Procedure

- **Construct** the new datatype using appropriate MPI routines
MPI_Type_contiguous, MPI_Type_vector,
MPI_Type_struct, MPI_Type_indexed,
MPI_Type_hvector, MPI_Type_hindexed
- **Commit** the new datatype
MPI_Type_Commit
- **Use** the new datatype in sends/receives, etc.

Datatype Construction

- **Datatype specified by its *type map***
 - Stencil laid over memory
- **Displacements are offsets (in bytes) from the starting memory address of the desired data**
 - `MPI_Type_extent` function can be used to get size (in bytes) of datatypes

Type Maps

Basic datatype 0	Displacement of datatype 0
Basic datatype 1	Displacement of datatype 1
...	...
Basic datatype n-1	Displacement of datatype n-1

Contiguous Datatype

- The **simplest** derived datatype consists of a number of contiguous items of the same datatype

C:

```
int MPI_Type_contiguous (int count,  
                        MPI_Datatype oldtype, MPI_Datatype *newtype)
```

Fortran:

```
INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR
```

```
CALL MPI_TYPE_CONTIGUOUS (COUNT, OLDTYPE, NEWTYPE, IERROR)
```

Sample Program #2 - C

```
#include <stdio.h>
#include <mpi.h>
/* Run with four processes */
void main(int argc, char *argv[]) {
    int rank;
    MPI_Status status;
    struct {
        int x;    int y;    int z;
    } point;
    MPI_Datatype ptype;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Type_contiguous(3,MPI_INT,&ptype);
    MPI_Type_commit(&ptype);
    if(rank==3){
        point.x=15; point.y=23; point.z=6;
        MPI_Send(&point,1,ptype,1,52,MPI_COMM_WORLD);
    } else if(rank==1) {
        MPI_Recv(&point,1,ptype,3,52,MPI_COMM_WORLD,&status);
        printf("P:%d received coords are (%d,%d,%d)
\n",rank,point.x,point.y,point.z);
    }
    MPI_Finalize();
}
```

```
P:1 received coords are (15,23,6)
```

Sample Program #2 - Fortran

```
PROGRAM contiguous
C Run with four processes
  INCLUDE 'mpif.h'
  INTEGER err, rank, size
  integer status(MPI_STATUS_SIZE)
  integer x,y,z
  common/point/x,y,z
  integer ptype
  CALL MPI_INIT(err)
  CALL MPI_COMM_RANK(MPI_COMM_WORLD,rank,err)
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD,size,err)
  call MPI_TYPE_CONTIGUOUS(3,MPI_INTEGER,ptype,err)
  call MPI_TYPE_COMMIT(ptype,err)
  print *,rank,size
  if(rank.eq.3) then
    x=15
    y=23
    z=6
    call MPI_SEND(x,1,ptype,1,30,MPI_COMM_WORLD,err)
  else if(rank.eq.1)then
    call MPI_RECV(x,1,ptype,3,30,MPI_COMM_WORLD,status,err)
    print *,'P:',rank,' coords are ',x,y,z
  end if
  CALL MPI_FINALIZE(err)
END
```

```
P:1 coords are 15, 23, 6
```

Vector Datatype

- User completely specifies memory locations defining the vector

C:

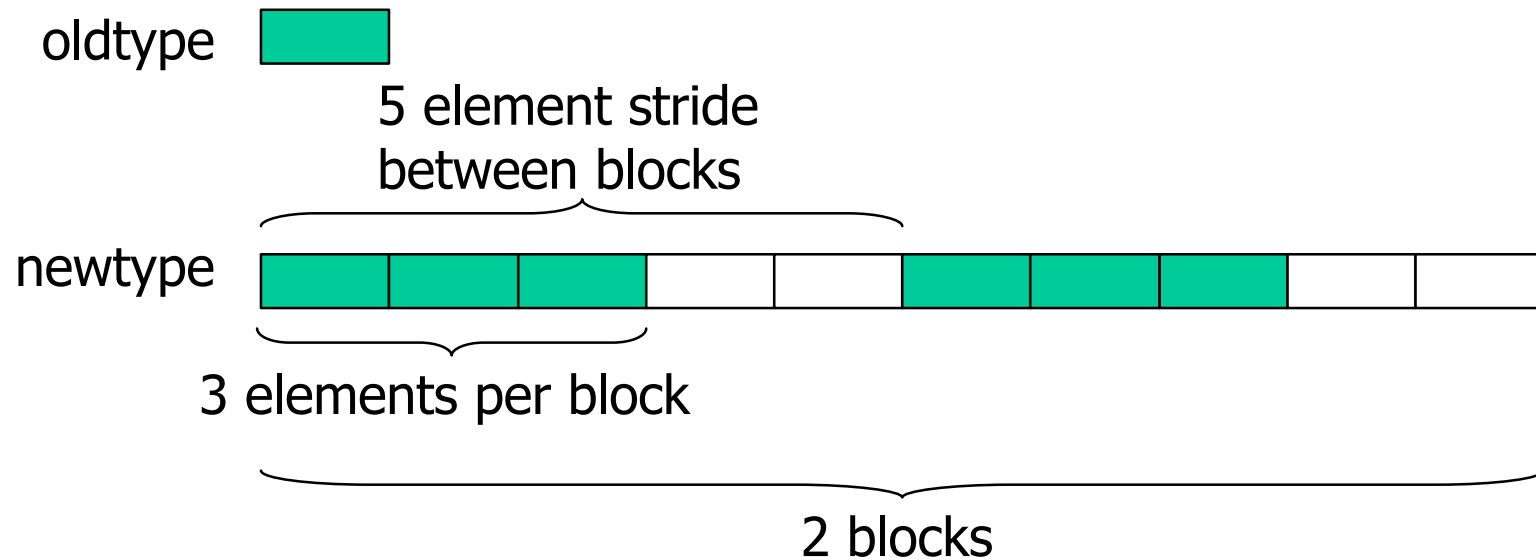
```
int MPI_Type_vector(int count, int blocklength, int stride,  
MPI_Datatype oldtype, MPI_Datatype *newtype)
```

Fortran:

```
CALL MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE,  
OLDTYPE, NEWTYPE, IERROR)
```

- *newtype* has *count* blocks each consisting of *blocklength* copies of *oldtype*
- Displacement between blocks is set by *stride*

Vector Datatype Example



- **count = 2**
- **stride = 5**
- **blocklength = 3**

Sample Program #3 - C

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

void main(int argc, char *argv[]) {
    int rank,i,j;
    MPI_Status status;
    double x[4][8];
    MPI_Datatype coltype;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Type_vector(4,1,8,MPI_DOUBLE,&coltype);
    MPI_Type_commit(&coltype);
    if(rank==3){
        for(i=0;i<4;++i)
            for(j=0;j<8;++j) x[i][j]=pow(10.0,i+1)+j;
        MPI_Send(&x[0][7],1,coltype,1,52,MPI_COMM_WORLD);
    } else if(rank==1) {
        MPI_Recv(&x[0][2],1,coltype,3,52,MPI_COMM_WORLD,&status);
        for(i=0;i<4;++i)printf("P:%d my x[%d][2]=%1f\n",rank,i,x[i][2]);
    }
    MPI_Finalize();
}
```

```
P:1 my x[0][2]=17.000000
P:1 my x[1][2]=107.000000
P:1 my x[2][2]=1007.000000
P:1 my x[3][2]=10007.000000
```

Sample Program #3 - Fortran

```
PROGRAM vector
C Run with four processes
INCLUDE 'mpif.h'
INTEGER err, rank, size
integer status(MPI_STATUS_SIZE)
real x(4,8)
integer rowtype
CALL MPI_INIT(err)
CALL MPI_COMM_RANK(MPI_COMM_WORLD,rank,err)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD,size,err)
call MPI_TYPE_VECTOR(8,1,4,MPI_REAL,rowtype,err)
call MPI_TYPE_COMMIT(rowtype,err)
if(rank.eq.3) then
  do i=1,4
    do j=1,8
      x(i,j)=10.0**i+j
    end do
  enddo
  call MPI_SEND(x(2,1),1,rowtype,1,30,MPI_COMM_WORLD,err)
else if(rank.eq.1)then
  call MPI_RECV(x(4,1),1,rowtype,3,30,MPI_COMM_WORLD,status,err)
  print *, 'P:',rank,' the 4th row of x is'
  do i=1,8
    print*,x(4,i)
  end do
end if
CALL MPI_FINALIZE(err)
END
```

```
P:1 the 4th row of x is
101.
102.
103.
104.
105.
106.
107.
108.
```

Extent of a Datatype

- Handy **utility** function for datatype construction
- Extent defined to be the memory span (in bytes) of a datatype

C:

```
MPI_Type_extent (MPI_Datatype datatype, MPI_Aint* extent)
```

Fortran:

```
INTEGER          DATATYPE, EXTENT, IERROR
```

```
CALL MPI_TYPE_EXTENT (DATATYPE, EXTENT, IERROR)
```


Structure Datatype

- Use for variables comprised of heterogeneous datatypes
 - C structures
 - Fortran common blocks
- This is the most general derived data type

C:

```
int MPI_Type_struct (int count, int *array_of_blocklengths,  
                    MPI_Aint *array_of_displacements,  
                    MPI_Datatype *array_of_types,  
                    MPI_Datatype *newtype)
```

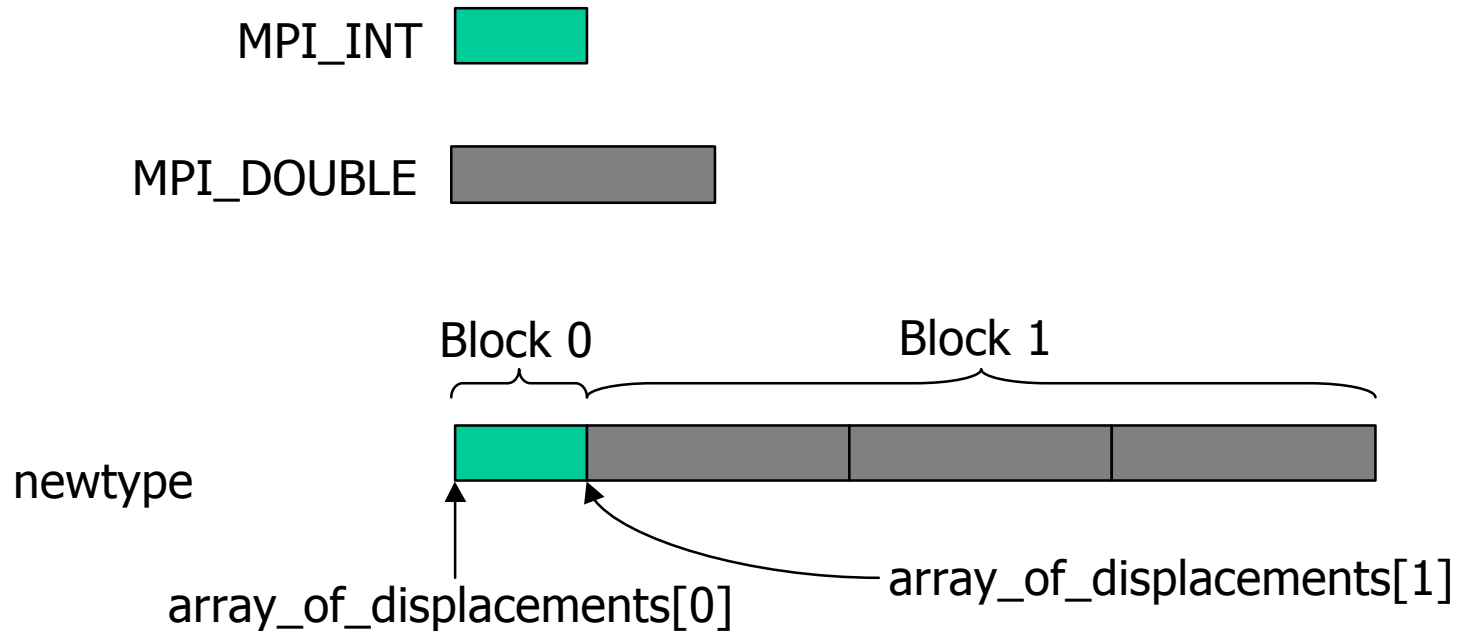
Fortran:

```
CALL MPI_TYPE_STRUCT(COUNT,ARRAY_OF_BLOCKLENGTHS,  
                    ARRAY_OF_DISPLACEMENTS,ARRAY_OF_TYPES,  
                    NEWTYPE,IERROR)
```

Structure Datatype (cont)

newtype consists of `count` blocks where the i^{th} block is `array_of_blocklengths[i]` copies of the type `array_of_types[i]`. The displacement of the i^{th} block (in bytes) is given by `array_of_displacements[i]`.

Struct Datatype Example



- `count = 2`
- `array_of_blocklengths = {1,3}`
- `array_of_types = {MPI_INT, MPI_DOUBLE}`
- `array_of_displacements = {0, extent(MPI_INT)}`

Sample Program #4 - C

```
#include <stdio.h>
#include<mpi.h>
void main(int argc, char *argv[]) {
    int rank,i;
    MPI_Status status;
    struct {
        int num;
        float x;
        double data[4];
    } a;
    int blocklengths[3]={1,1,4};
    MPI_Datatype types[3]={MPI_INT,MPI_FLOAT,MPI_DOUBLE};
    MPI_Aint displacements[3];
    MPI_Datatype restype;
    MPI_Aint intex,floatex;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Type_extent(MPI_INT,&intex);MPI_Type_extent(MPI_FLOAT,&floatex);
    displacements[0]= (MPI_Aint) 0; displacements[1]=intex;
    displacements[2]=intex+floatex;
    MPI_Type_struct(3,blocklengths,displacements,types,&restype);
```

Sample Program #4 - C (cont.)

```
...
MPI_Type_commit(&restype);
if (rank==3){
    a.num=6; a.x=3.14; for(i=0;i<4;++i) a.data[i]=(double) i;
    MPI_Send(&a,1,restype,1,52,MPI_COMM_WORLD);
} else if(rank==1) {
    MPI_Recv(&a,1,restype,3,52,MPI_COMM_WORLD,&status);
    printf("P:%d my a is %d %f %lf %lf %lf %lf\n",
           rank,a.num,a.x,a.data[0],a.data[1],a.data[2],a.data[3]);
}
MPI_Finalize();
}
```

```
P:1 my a is 6 3.140000 0.000000 1.000000 2.000000 3.000002
```

Sample Program #4 - Fortran

```
PROGRAM structure
INCLUDE 'mpif.h'
INTEGER err, rank, size
integer status(MPI_STATUS_SIZE)
integer num
real x
complex data(4)
common /result/num,x,data
integer blocklengths(3)
data blocklengths/1,1,4/
integer displacements(3)
integer types(3),restype
data types/MPI_INTEGER,MPI_REAL,MPI_COMPLEX/
integer intex,realex

CALL MPI_INIT(err)
CALL MPI_COMM_RANK(MPI_COMM_WORLD,rank,err)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD,size,err)
call MPI_TYPE_EXTENT(MPI_INTEGER,intex,err)
call MPI_TYPE_EXTENT(MPI_REAL,realex,err)
displacements(1)=0
displacements(2)=intex
displacements(3)=intex+realex
```

Sample Program #4 - Fortran (cont.)

```
call MPI_TYPE_STRUCT(3,blocklengths, displacements,types,
&                    restype,err)
call MPI_TYPE_COMMIT(restype,err)
if(rank.eq.3) then
  num=6
  x=3.14
  do i=1,4
    data(i)=cmplx(i,i)
  end do
  call MPI_SEND(num,1,restype,1,30,MPI_COMM_WORLD,err)
else if(rank.eq.1)then
  call MPI_RECV(num,1,restype,3,30,MPI_COMM_WORLD,status,err)
  print*,'P:',rank,' I got'
  print*,num
  print*,x
  print*,data
end if
CALL MPI_FINALIZE(err)
END
```

```
P:1 I got
6
3.1400001
(1.,1.), (2.,2.), (3.,3.), (4.,4.)
```

Committing a Datatype

- Once a datatype has been constructed, it needs to be committed before it is used.
- This is done using `MPI_TYPE_COMMIT`

C:

```
int MPI_Type_commit (MPI_Datatype *datatype)
```

Fortran:

```
CALL MPI_TYPE_COMMIT (DATATYPE, IERROR)
```


Virtual Topologies

- [Virtual Topologies](#)
- [Topology Types](#)
- [Creating a Cartesian Virtual Topology](#)
- [Cartesian Example](#)
- [Cartesian Mapping Functions](#)
 - MPI_CART_RANK*
 - MPI_CART_COORDS*
 - MPI_CART_SHIFT*
- [Cartesian Partitioning](#)

*includes sample C and Fortran programs

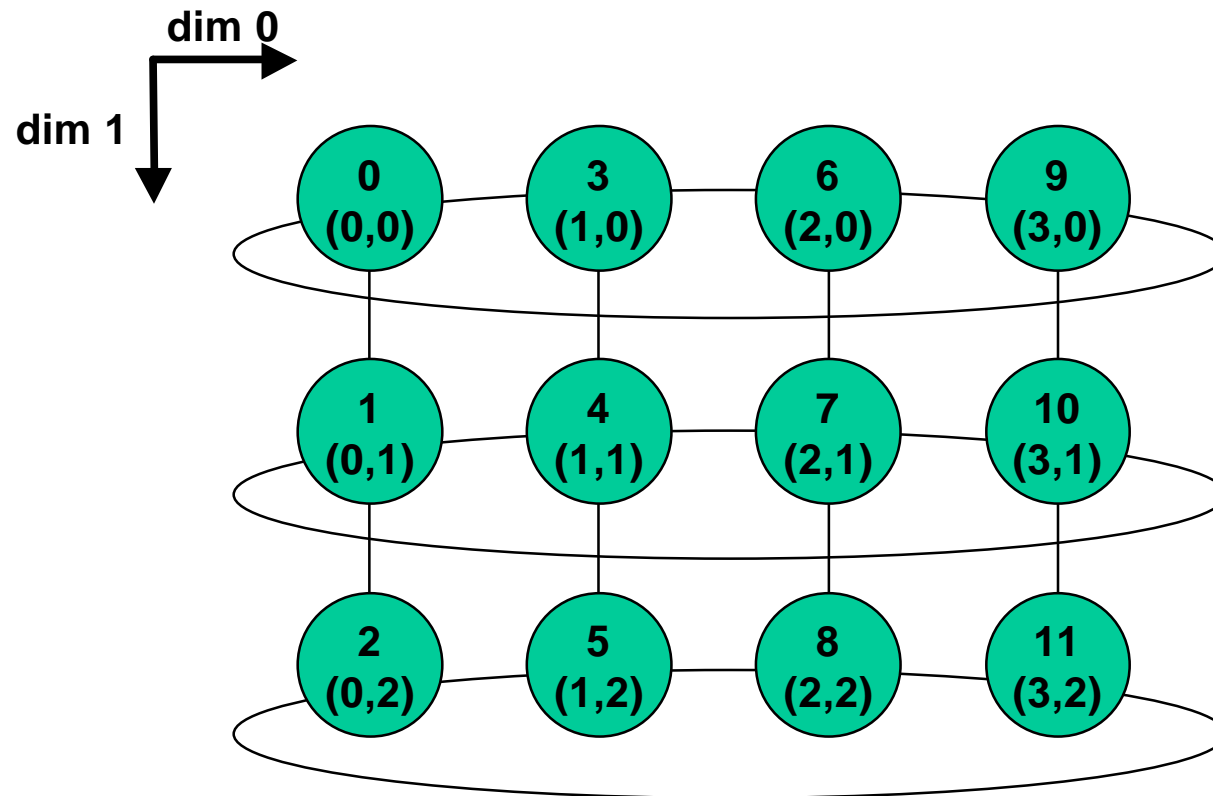
Virtual Topologies

- Convenient process naming
- Naming scheme to fit the communication pattern
- Simplifies writing of code
- Can allow MPI to optimize communications
- Rationale: access to useful topology routines

How to use a Virtual Topology

- Creating a topology produces a new communicator
- MPI provides “mapping functions”
- Mapping functions compute processor ranks, based on the topology naming scheme

Example - 2D Torus



Topology types

- **Cartesian topologies**
 - Each process is connected to its neighbors in a virtual grid
 - Boundaries can be cyclic
 - Processes can be identified by Cartesian coordinates

- **Graph topologies**
 - General graphs
 - Will not be covered here

Creating a Cartesian Virtual Topology

C:

```
int MPI_Cart_create (MPI_Comm comm_old, int ndims,  
                    int *dims, int *periods, int reorder,  
                    MPI_Comm *comm_cart)
```

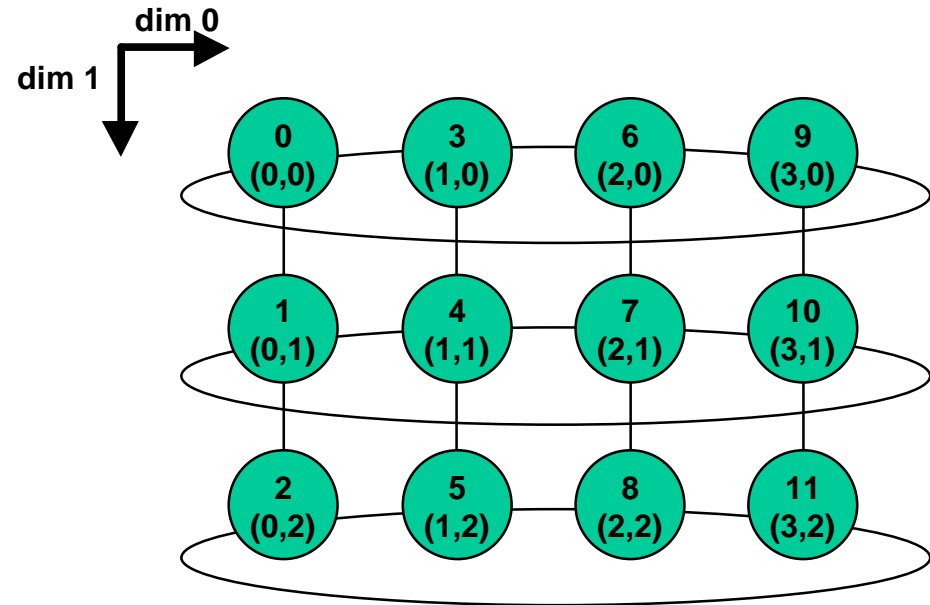
Fortran:

```
INTEGER      COMM_OLD,NDIMS,DIMS(*),COMM_CART,IERROR  
LOGICAL      PERIODS(*),REORDER  
  
CALL MPI_CART_CREATE(COMM_OLD,NDIMS,DIMS,PERIODS,REORDER,  
                    COMM_CART,IERROR)
```

Arguments

<code>comm_old</code>	existing communicator
<code>ndims</code>	number of dimensions
<code>periods</code>	logical array indicating whether a dimension is cyclic (If TRUE, cyclic boundary conditions)
<code>reorder</code>	logical (If FALSE, rank preserved) (If TRUE, possible rank reordering)
<code>comm_cart</code>	new cartesian communicator

Cartesian Example



```
MPI_Comm vu;  
int dim[2], period[2], reorder;  
  
dim[0]=4; dim[1]=3;  
period[0]=TRUE; period[1]=FALSE;  
reorder=TRUE;  
  
MPI_Cart_create(MPI_COMM_WORLD, 2, dim, period, reorder, &vu);
```


Cartesian Mapping Functions

Mapping process grid coordinates to ranks

C:

```
int MPI_Cart_rank (MPI_Comm comm, int *coords, int *rank)
```

Fortran:

```
INTEGER      COMM, COORDS (*), RANK, IERROR
```

```
CALL MPI_CART_RANK (COMM, COORDS, RANK, IERROR)
```

Cartesian Mapping Functions

Mapping ranks to process grid coordinates

C:

```
int MPI_Cart_coords (MPI_Comm comm, int rank, int maxdims,  
                    int *coords)
```

Fortran:

```
INTEGER          COMM,RANK,MAXDIMS,COORDS(*),IERROR
```

```
CALL MPI_CART_COORDS(COMM,RANK,MAXDIMS,COORDS,IERROR)
```

Sample Program #9 - C

```
#include<mpi.h>
/* Run with 12 processes */
void main(int argc, char *argv[]) {
    int rank;
    MPI_Comm vu;
    int dim[2],period[2],reorder;
    int coord[2],id;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    dim[0]=4; dim[1]=3;
    period[0]=TRUE; period[1]=FALSE;
    reorder=TRUE;
    MPI_Cart_create(MPI_COMM_WORLD,2,dim,period,reorder,&vu);
    if(rank==5){
        MPI_Cart_coords(vu,rank,2,coord);
        printf("P:%d My coordinates are %d %d\n",rank,coord[0],coord[1]);
    }
    if(rank==0) {
        coord[0]=3; coord[1]=1;
        MPI_Cart_rank(vu,coord,&id);
        printf("The processor at position (%d, %d) has rank %d\n",coord[0],coord[1],id);
    }
    MPI_Finalize();
}
```

```
The processor at position (3,1) has rank 10
P:5 My coordinates are 1 2
```

Sample Program #9 - Fortran

```
PROGRAM Cartesian
C Run with 12 processes
  INCLUDE 'mpif.h'
  INTEGER err, rank, size
  integer vu,dim(2),coord(2),id
  logical period(2),reorder
  CALL MPI_INIT(err)
  CALL MPI_COMM_RANK(MPI_COMM_WORLD,rank,err)
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD,size,err)
  dim(1)=4
  dim(2)=3
  period(1)=.true.
  period(2)=.false.
  reorder=.true.
  call MPI_CART_CREATE(MPI_COMM_WORLD,2,dim,period,reorder,vu,err)
  if(rank.eq.5) then
    call MPI_CART_COORDS(vu,rank,2,coord,err)
    print*,'P:',rank,' my coordinates are',coord
  end if
  if(rank.eq.0) then
    coord(1)=3
    coord(2)=1
    call MPI_CART_RANK(vu,coord,id,err)
    print*,'P:',rank,' processor at position',coord,' is',id
  end if
  CALL MPI_FINALIZE(err)
END
```

```
P:5 my coordinates are 1, 2
P:0 processor at position 3, 1 is 10
```

Cartesian Mapping Functions

Computing ranks of neighboring processes

C:

```
int MPI_Cart_shift (MPI_Comm comm, int direction, int disp,  
                  int *rank_source, int *rank_dest)
```

Fortran:

```
INTEGER          COMM,DIRECTION,DISP,RANK_SOURCE,RANK_DEST,IERROR  
  
CALL MPI_CART_SHIFT(COMM,DIRECTION,DISP,RANK_SOURCE,  
                   RANK_DEST,IERROR)
```

MPI_Cart_shift

- Does not actually shift data: returns the correct ranks for a shift which can be used in subsequent communication calls
- Arguments:
 - `direction` dimension in which the shift should be made
 - `disp` length of the shift in processor coordinates (+ or -)
 - `rank_source` where calling process should receive a message **from** during the shift
 - `rank_dest` where calling process should send a message **to** during the shift
- If shift off of the topology, `MPI_Proc_null` is returned

Sample Program #10 - C

```
#include<mpi.h>
#define TRUE 1
#define FALSE 0
void main(int argc, char *argv[]) {
    int rank;
    MPI_Comm vu;
    int dim[2],period[2],reorder;
    int up,down,right,left;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    dim[0]=4; dim[1]=3;
    period[0]=TRUE; period[1]=FALSE;
    reorder=TRUE;
    MPI_Cart_create(MPI_COMM_WORLD,2,dim,period,reorder,&vu);
    if(rank==9){
        MPI_Cart_shift(vu,0,1,&left,&right);
        MPI_Cart_shift(vu,1,1,&up,&down);
        printf("P:%d My neighbors are r: %d d:%d l:%d
u:%d\n",rank,right,down,left,up);
    }
    MPI_Finalize();
}
```

```
P:9 my neighbors are r:0 d:10 l:6 u:-1
```

Sample Program #10- Fortran

```
PROGRAM neighbors
C
C Run with 12 processes
C
  INCLUDE 'mpif.h'
  INTEGER err, rank, size
  integer vu
  integer dim(2)
  logical period(2),reorder
  integer up,down,right,left
  CALL MPI_INIT(err)
  CALL MPI_COMM_RANK(MPI_COMM_WORLD,rank,err)
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD,size,err)
  dim(1)=4
  dim(2)=3
  period(1)=.true.
  period(2)=.false.
  reorder=.true.
  call MPI_CART_CREATE(MPI_COMM_WORLD,2,dim,period,reorder,vu,err)
  if(rank.eq.9) then
    call MPI_CART_SHIFT(vu,0,1,left,right,err)
    call MPI_CART_SHIFT(vu,1,1,up,down,err)
    print*,'P:',rank,' neighbors (rdlu)are',right,down,left,up
  end if
  CALL MPI_FINALIZE(err)
END
```

```
P:9 neighbors (rdlu) are 0, 10, 6, -1
```


Cartesian Partitioning

- Often we want to do an operation on only part of an existing Cartesian topology
- Cut a grid up into 'slices'
- A new communicator is produced for each slice
- Each slice can then perform its own collective communications
- `MPI_Cart_sub` and `MPI_CART_SUB` generate new communicators for the slice

MPI_Cart_sub

C:

```
int MPI_Cart_sub (MPI_Comm comm, int *remain_dims,  
                 MPI_Comm *newcomm)
```

Fortran:

```
INTEGER      COMM,NEWCOMM,IERROR  
LOGICAL      REMAIN_DIMS( *)
```

```
CALL MPI_CART_SUB(COMM,REMAIN_DIMS,NEWCOMM,IERROR)
```

- If `comm` is a 2x3x4 grid and `remain_dims={TRUE,FALSE,TRUE}`, then three new communicators are created each being a 2x4 grid
- Calling processor receives back only the new communicator it is in

Problem Set

- ① a) Write an MPI program in which each processors prints out the message “Parallel Processing”. Pick the number of processors you wish to use as long as it is four or more.

b) Revise your program written for part a) so that each processor prints out its rank as well as the message. Run this program six times. What do you notice?

c) Revise your program written for part b) so that only the processor whose rank equals half the total number of processors performs the output. (When you run your code, use an even number of processors.)
- ② Rewrite Sample Program #1 (in the handouts) so that a message is sent from P1 (P1= processor with rank 1) to P2. The message should be an array of 50 integers: each element equal to 6. P2 should receive the message with both the source and tag wildcarded. P2 should then print out who was the source, what was the tag, and one of the array elements received.
- ③ Write a program in which data are exchanged between P1 and P3. P1 should calculate the squares of the first 200 integers. The resulting array should then be transferred to P3.

P3 should divide the integers between 20 and 119 by 53.0. The resulting (real) array should then be transferred to P1.

Output whatever you feel is necessary to confirm that the data transfers occurred correctly. Use **NO** wildcarding in your sends and receives.

Problem Set

- ④ Write a program which will be run on four processors. P3 will receive messages from all the other processors with the source rank being wildcarded. Each processor should send a different integer (you pick them). When P3 receives a message from P0, it should print out 2 raised to the integer power contained in the message. When P3 receives a message from P1, it should print out the factorial of the integer received. And, finally, when P3 receives a message from P2 it should print out the sum of all integers up to and including the integer in the message.
- ⑤ Write a program in which four processors search an array in parallel (each gets a fourth of the elements to search). All the processors are searching the integer array for the element whose value is 11. There is only one 11 in the entire array of 400 integers.

By using the **non-blocking** MPI commands you have learned, have each processor continue searching until one of them has found the 11. Then they all should stop and print out the index they stopped their own search at.

You have been given a file called data which contains the integer array (ASCII, one element per line). Before the searching begins have **ONLY** P0 read in the array elements from the data file and distribute one fourth to each of the other processors and keep one fourth for its own search.

- ⑥ If you did not use the MPI broadcast command in your program for problem 5, rewrite it so it does. There is a natural place in the program where a broadcast is useful

Problem Set

- ⑦ If you did not use the MPI scatter command in your program for problem 5, rewrite it so it does. There is a natural place in the program where a scatter is useful
- ⑧ In this problem each of eight processors used will contain an integer value in its memory that will be the operand in a **collective reduction operation**. The operand value for each processor is -27, -4, 31, 16, 20, 13, 49, and 1 respectively.

Write a program in which the maximum value of the integer operands is determined. The result should be stored on P5. P5 should then transfer the maximum value to all the other processors. All eight processors will then normalize their operands by dividing by the maximum value. (**EXTRA CREDIT**: Consider using MPI_ALL_REDUCE)

Finally, the program should calculate the sum of all the normalized values and put the result on P2. P2 should then output the normalized global sum.

- ⑨ Write a program for three processors in the boss-worker programming style. Specifically, P0 will be the boss. It will first create a 40-element real 1-D array. The value for each array element will be the its index raised to the 2.5 power.

Processors 1 and 2 will act as workers. P1 will calculate the average of all the array elements which have an odd index. P2 will calculate the average of all the array elements which have an even index. When the workers are done they will output their results.

P0 will transfer **ONLY** the odd-indexed elements of the array to P1. Similarly for P2 and the even-indexed elements. Your program should do these transfers by making a new **derived vector data type** that consists of every other element in the 1-D array. The boss and each worker should use only one send/receive pair, using the new data type.

Problem Set

- ⑩ Write a program that will make a **virtual topology** using 8 processors. The topology should consist of 4 processor rows and 2 processors columns with no wrap-around (cyclic) in either dimension. Each processor in the topology should have an integer variable with the same values used in Problem 8.

After your program has created the topology, it should use virtual topology utility functions to have each processor calculate the average value of its integer and the integers contained in it's neighbors. Each processor should then output its calculated average. (**NOTE:** do not use "diagonal" neighbors in the averaging. Only use "up", "down", "left", and "right" neighbors, if they exist).