



Parallel Programming Using MPI

David Porter & Mark Nelson

(612) 626-0802
help@msi.umn.edu

September 27, 2011

Supercomputing Institute
for Advanced Computational Research



UNIVERSITY OF MINNESOTA
Driven to DiscoverSM

Agenda

10:00-10:15 Introduction to MSI Resources

10:15-10:30 Introduction to MPI

10:30-11:30 Blocking Communication

11:30-12:00 Hands-on

12:00- 1:00 Lunch

1:00- 1:45 Non-Blocking Communication

1:45- 2:20 Collective Communication

2:20- 2:45 Hands-on

2:45- 2:50 Break

2:50- 3:30 Collective Computation and Synchronization

3:30- 4:00 Hands-on

Introduction



Supercomputing Institute
for Advanced Computational Research



UNIVERSITY OF MINNESOTA
Driven to DiscoverSM

Itasca



<http://www.msi.umn.edu/Itasca>

HP Linux Cluster

- 1091 compute nodes
- Each node has 2 quad-core 2.8 GHz Intel Nehalem processors
- Total of 8,728 cores
- 24 GB of memory per node
- Aggregate of 26 TB of RAM
- QDR Infiniband interconnect
- Scratch space: Lustre shared file system
- Currently 550 TB

Calhoun



<http://www.msi.umn.edu/calhoun>

SGI Altix XE 1300

- 256 compute nodes
- Each node has 2 quad-core 2.66 GHz Intel Clovertown processors
- Total of 2048 cores
- 16 GB of memory per node
- Aggregate of 4.1 TB of RAM
- Three Altix 240 server nodes
- Two Altix 240 interactive nodes
- Diskless Boot Nodes
 - Two Altix 240 node
- Infiniband 4x DDR HCA
- Scratch Space - 36 TB (/scratch1 ... /scratch4)

Koronis

- ***NIH***

- ***uv1000***

Production system: 1152 cores, 3 TiB memory

- ***Two uv100s***

Development systems: 72 cores, 48 GB, TESLA

- ***One uv10 and three SGI C1103 sysems***

Interactive Graphics nodes

static.msi.umn.edu/tutorial/hardwareprogramming/Koronis_2011june16_final.pdf

www.msi.umn.edu/hardware/koronis

UV1000: ccNUMA Architecture

ccNUMA:

- Cache coherent non-uniform memory access
- Memory local to processor but available to all
- Copies of memory cached locally

NUMALink 5 (NL5)

- SGI's 5th generation NUMA interconnect
- 4 NUMALink 5 lines per processor board
- 7.5 GB/s (unidirectional) *peak* per NL5 line
- 2-D torus of NL5 lines between boardpairs

	Itasca		Calhoun	
Primary Queue	24 hours	1086 nodes (8688 cores)	24 hours	254 nodes (2,032 cores)
Development Queue	2 hours	32 nodes (256 cores)	1 hour	8 nodes (64 cores)
Medium Queue	N/A	N/A	48 hours	16 nodes (128 cores)
Long Queue	48 hours	28 nodes (224 cores)	96 hours	16 nodes (128 cores)
Max Queue	N/A	N/A	300 hours	2 nodes (16 cores)
Restrictions	Up to 2 running and 3 queued jobs per user.		Up to 5 running and 5 queued jobs per user.	

Intro to MPI



Introduction to parallel programming

Serial vs. Parallel

- **Serial: => One processor**
 - Execution of a program sequentially, one statement at a time
- **Parallel: => Multiple processors**
 - Breaking tasks into smaller tasks–Coordinating the workers
 - Assigning smaller tasks to workers to work simultaneously
- **In parallel computing, a program uses concurrency to either**
 - decrease the runtime needed to solve a problem
 - increase the size of problem that can be solved

Introduction to parallel programming

What kind of applications can benefit?

- Materials / Superconductivity, Fluid Flow, Weather/Climate, Structural Deformation , Genetics / Protein interactions, Seismic Modeling, and others...

How to solve these problems?

- Take advantage of parallelism
 - Large problems generally have many operations which can be performed concurrently
- Parallelism can be exploited at many levels by the Computer hardware
 - Within the CPU core, multiple functional units, pipelining
 - Within the Chip, many cores
 - On a node, multiple chips
 - In a system, many nodes

Introduction to parallel programming

Parallel Programming

– Involves:

- Decomposing an algorithm or data into parts
- Distributing the parts as tasks to multiple processors
- Processors to work simultaneously
- Coordinating work and communications of those processors

– Considerations

- Type of parallel architecture being used
- Type of processor communications used

Introduction to parallel programming

Parallel programming

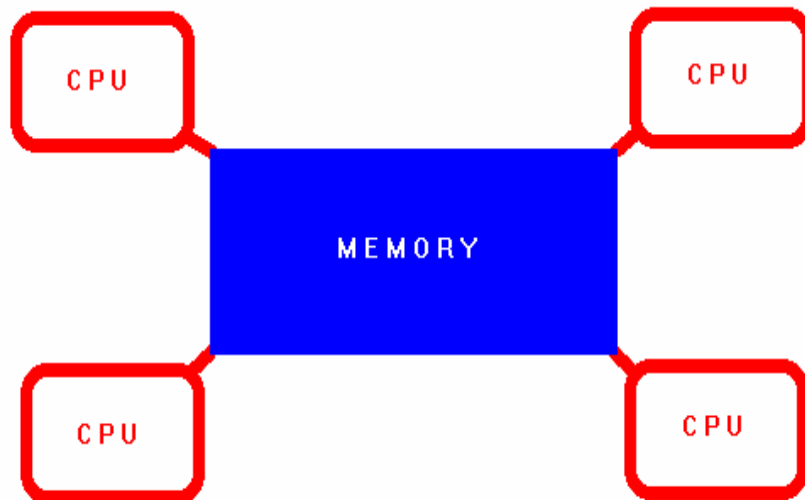
– **Requires:**

- **Multiple processors**
- **Network (distributed memory machines, cluster, etc.)**
- **Environment to create and manage parallel processing**
 - **Operating System**
 - **Parallel Programming Paradigm**
 - » **Message Passing: MPI**
 - » **OpenMP, pThreads**
 - » **CUDA, OpenCL**
- **A parallel algorithm → parallel program**

- **Processor Communications and Memory architectures**
 - **Inter-processor communication is required to:**
 - Convey information and data between processors
 - Synchronize processor activities
 - **Inter-processor communication depends on memory architecture, which impacts on how the program is written**
 - **Memory architectures**
 - Shared Memory
 - Distributed Memory
 - Distributed Shared Memory

Shared Memory

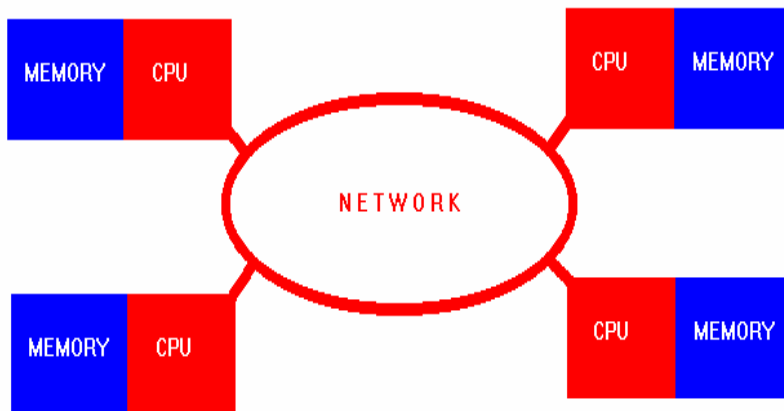
Shared Memory



- Only one processor can access the shared memory location at a time
- Synchronization achieved by controlling tasks reading from and writing to the shared memory
- Advantages: Easy for user to use efficiently, data sharing among tasks is fast, ...
- Disadvantages: Memory is bandwidth limited, user responsible for specifying synchronization, ...

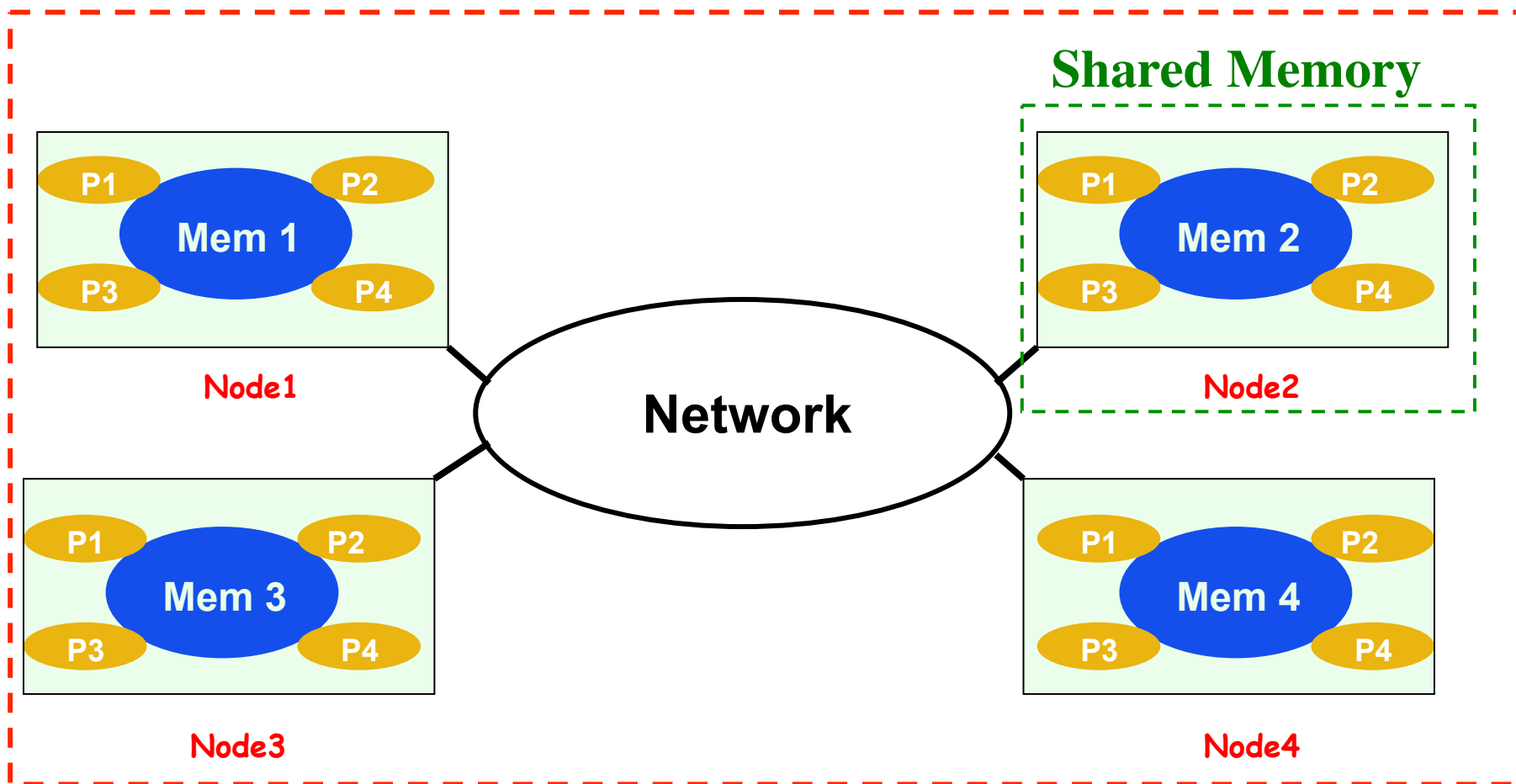
Distributed Memory

Distributed Memory



- Data is shared across a communication network using message passing
- User responsible for synchronization using message passing
- Advantages: Scalability, Each processor can rapidly access its own memory without interference, ...
- Disadvantages: Difficult to map existing data structures to this memory organization, User responsible for send/receive data among processors, ...

Distributed Shared Memory



MPI

- **MPI** Stands for: **M**essage **P**assing **I**nterface
- A message passing library **specification**
 - Model for distributed memory platforms
 - Not a compiler specification
- For parallel computers, clusters and heterogeneous networks
- Designed for
 - End users
 - Library writers
 - Tool developers
- Interface specification have been defined for C/C++ and Fortran Programs

MPI-Forum

- The MPI standards body
 - 60 people from forty different organizations (industry, academia, gov. labs)
 - International representation
- MPI 1.X Standard developed from 1992-1994
 - Base standard
 - Fortran and C language APIs
 - Current revision 1.3
- MPI 2.X Standard developed from 1995-1997
 - MPI I/O
 - One-sided communication
 - Current revision 2.2
- MPI 3.0 Standard under development 2008-?
 - Non-blocking collectives
 - Revisions/additions to one-sided communication
 - Fault tolerance
 - Hybrid programming – threads, PGAS, GPU programming
- Standards documents
 - <http://www.mcs.anl.gov/mpi>
 - <http://www.mpi-forum.org/>

Reasons for using MPI

- **Standardization** – supported on virtually all HPC platforms. Practically replaced all previous message passing libraries.
- **Portability** - There is no need to modify your source code when you port your application to a different platform that supports (and is compliant with) the MPI standard.
- **Performance Opportunities** - Vendor implementations should be able to exploit native hardware features to optimize performance.
- **Functionality** - Over 115 routines are defined in MPI-1 alone. There are a lot more routines defined in MPI-2.
- **Availability** - A variety of implementations are available, both vendor and public domain.

Parallel programming paradigms

- **SPMD** (Single Program Multiple Data)
 - All processes follow essentially the same execution path
 - Data-driven execution
 - Same program, different data
- **MPMD** (Multiple Program Multiple Data)
 - Master and slave processes follow distinctly different execution paths

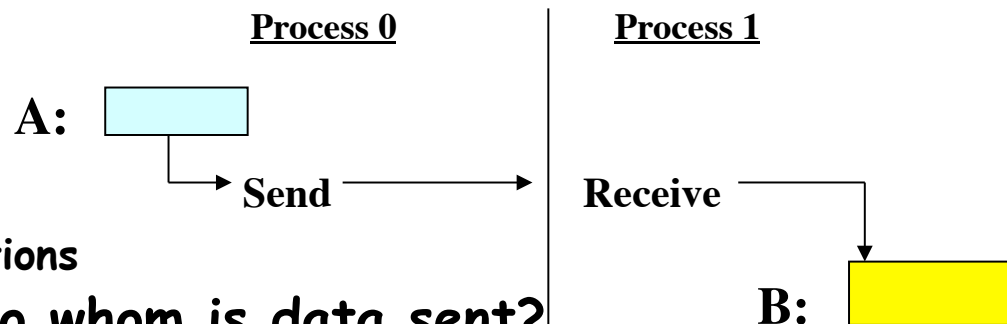
Point to Point Communication:

MPI Blocking Communication



Sending and Receiving Messages

- Basic message passing procedure: One process sends a message and a second process receives the message.



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



Message is divided into **data** and **envelope**

- **data**
 - buffer
 - count
 - datatype
- **envelope**
 - process identifier (source/destination rank)
 - message tag
 - communicator

MPI Calling Conventions

Fortran Bindings:

Call **MPI_XXXX** (... , ierror)

- Case insensitive
- Almost all MPI calls are subroutines
- ierror is always the last parameter
- Program must include 'mpif.h'

C Bindings:

int ierror = **MPI_Xxxxxx** (...)

- Case sensitive (as it always is in C)
- All MPI calls are functions: most return integer error code
- Program must include "mpi.h"
- Most parameters are passed by reference (i.e as pointers)

MPI Basic Send/Receive

Blocking send:

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

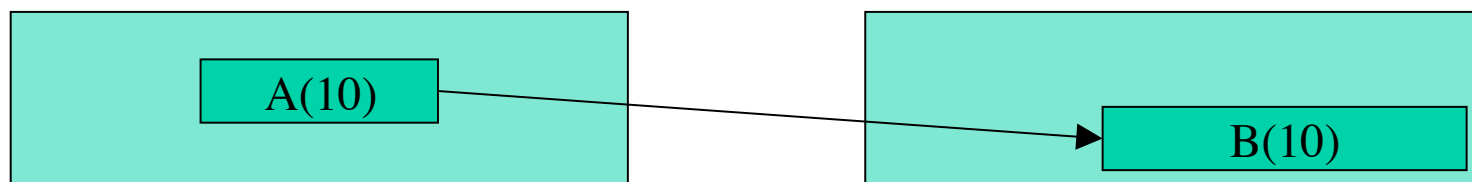
Blocking receive:

MPI_Recv (buffer, count, datatype, source, tag, comm, status)

Example: sending an array **A** of 10 integers

MPI_Send (**A**, 10, MPI_INT, dest, tag, MPI_COMM_WORLD)

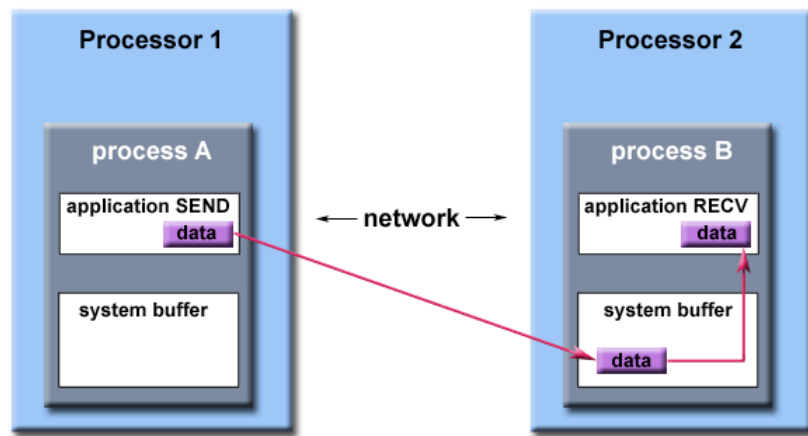
MPI_Recv (**B**, 10, MPI_INT, source, tag, MPI_COMM_WORLD, status)



MPI_Send(A, 10, MPI_INT, 1, ...)

MPI_Recv(B, 10, MPI_INT, 0, ...)

Buffering



Path of a message buffered at the receiving process

- A **system buffer** area is reserved to hold data in transit
- System buffer space is:
 - Opaque to the programmer and managed entirely by the MPI library
 - A finite resource that can be easy to exhaust
 - Often mysterious and not well documented
 - Able to exist on the sending side, the receiving side, or both
 - Something that may improve program performance because it allows send - receive operations to be asynchronous.
- User managed address space (i.e. your program variables) is called the **application buffer**.

MPI C Datatypes

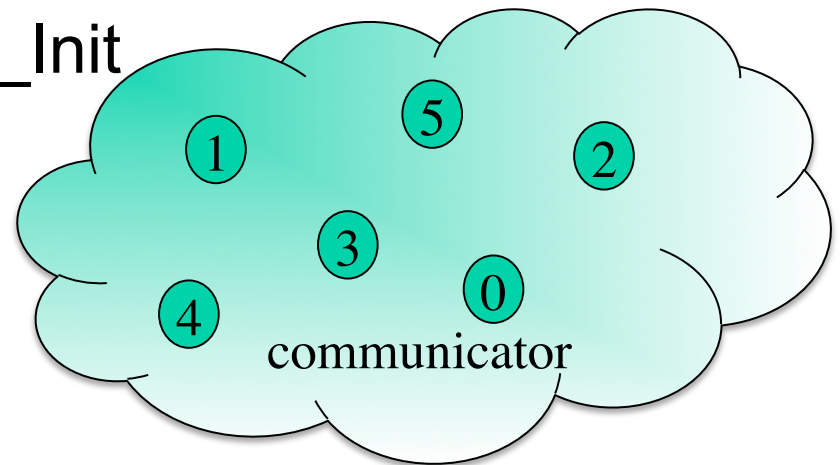
MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED	unsigned int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long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

MPI Fortran Datatypes

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

MPI Communicators

- An MPI object that defines a group of processes that are permitted to communicate with one another
- All MPI communication calls have a communicator argument
- Most often you will use MPI_COMM_WORLD
 - Default communicator
 - Defined when you call MPI_Init
 - It is all of your processes...



MPI Process Identifier: Rank

- A **rank** is an integer identifier assigned by the system to every process when the process initializes. Each process has its own unique rank.
- A rank is sometimes also called a "**task ID**". Ranks are contiguous and begin at zero.
- A rank is used by the programmer to specify the source and destination of a message
- A rank is often used conditionally by the application to control program execution (if rank=0 do this / if rank=1 do that).

MPI Message Tag

- Tags allow programmers to deal with the arrival of messages in an orderly manner
- The MPI standard guarantees that integers can be used as tags, but most implementations allow a much larger range of message tags
- `MPI_ANY_TAG` can be used as a wild card

Types of Point-to-Point Operations:

- Communication Modes
 - Define the procedure used to transmit the message and set of criteria for determining when the communication event (send or receive) is complete
 - Four communication modes available for sends:
 - Standard
 - Synchronous
 - Buffered
 - Ready
- Blocking vs non-blocking send/receive calls

MPI Blocking Communication

- MPI_SEND does not complete until buffer is empty (available for reuse)
- MPI_RECV does not complete until buffer is full (available for use)
- A process sending data *may or may not* be blocked until the receive buffer is filled – depending on many factors.
- Completion of communication generally depends on the message size and the system buffer size
- Blocking communication is simple to use but can be prone to deadlocks
- A blocking or nonblocking send can be paired to a blocking or nonblocking receive

Deadlocks

- **Two or more processes are in contention for the same set of resources**
- **Cause**
 - All tasks are waiting for events that haven't been initiated yet
- **Avoiding**
 - Different ordering of calls between tasks
 - Non-blocking calls
 - Use of MPI_SendRecv
 - Buffered mode

MPI Deadlock Examples

- Below is an example that may lead to a deadlock

Process 0	Process 1
Send (1)	Send (0)
Recv (1)	Recv (0)

- An example that definitely will deadlock

Process 0	Process 1
Recv (1)	Recv (0)
Send (1)	Send (0)

Note: the RECV call is blocking. The send call never execute, and both processes are blocked at the RECV resulting in deadlock.

- The following scenario is always safe

Process 0	Process 1
Send (1)	Recv (0)
Recv (1)	Send (0)

Fortran Example

```
program MPI_small
include 'mpif.h'
integer rank, size, ierror, tag, status(MPI_STATUS_SIZE)
character(12) message

call MPI_INIT(ierror)
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
tag = 100
if(rank .eq. 0) then

message = 'Hello, world'
do i=1, size-1
    call MPI_SEND(message, 12, MPI_CHARACTER, i, tag, MPI_COMM_WORLD,
    ierror)
enddo
else
    call MPI_RECV(message, 12, MPI_CHARACTER, 0, tag, MPI_COMM_WORLD,
    status, ierror)
endif
print*, 'node' rank, ': ', message
call MPI_FINALIZE(ierror)
end
```

C Example

```
#include<stdio.h>
#include "mpi.h"
main(int argc, char **argv)
{
    int rank, size, tag, rc, i;
    MPI_Status status;
    char message[20];
    rc = MPI_Init(&argc,&argv);
    rc = MPI_Comm_size(MPI_COMM_WORLD,&size);
    rc = MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    tag = 100;
    if(rank == 0) {
        strcpy(message, "Hello, world");
        for (i=1; i<size; i++)
            rc = MPI_Send(message, 13, MPI_CHAR, i, tag, MPI_COMM_WORLD);
    }
    else
        rc = MPI_Recv(message, 13, MPI_CHAR, 0, tag, MPI_COMM_WORLD,
            &status);
    print("node %d : %.13s\n", rank,message);
    rc = MPI_Finalize();
}
```

Compiling and Running MPI codes

Calhoun: SGI Altix XE 1300

- To compile: **module load intel ompi/intel**

- Fortran: **mpif90 -O3 program.f**

- C: **mpicc -O3 program.c**

- C++: **mpicxx -O3 program.c**

- To run:

- Interactively:

- mpirun -np 2 ./a.out**

- Batch jobs to the queue : Use PBS

<http://www.msi.umn.edu/hardware/calhoun/quickstart.html>

Hands-on

http://www.msi.umn.edu/tutorial/scicomp/general/MPI/workshop_MPI

Get example and build

```
ssh calhoun.msi.umn.edu  
cp -r /soft/examples/hello_mpi_intel_ompi .  
cd hello_mpi_intel_ompi  
module load intel_ompi/intel  
make
```

Run interactively on login node

```
mpirun -np 4 ./hello
```

Run in PBS queue

```
qsub run.pbs
```


Point to Point Communication:

MPI non-blocking Communication



Blocking vs Non-blocking

Blocking:

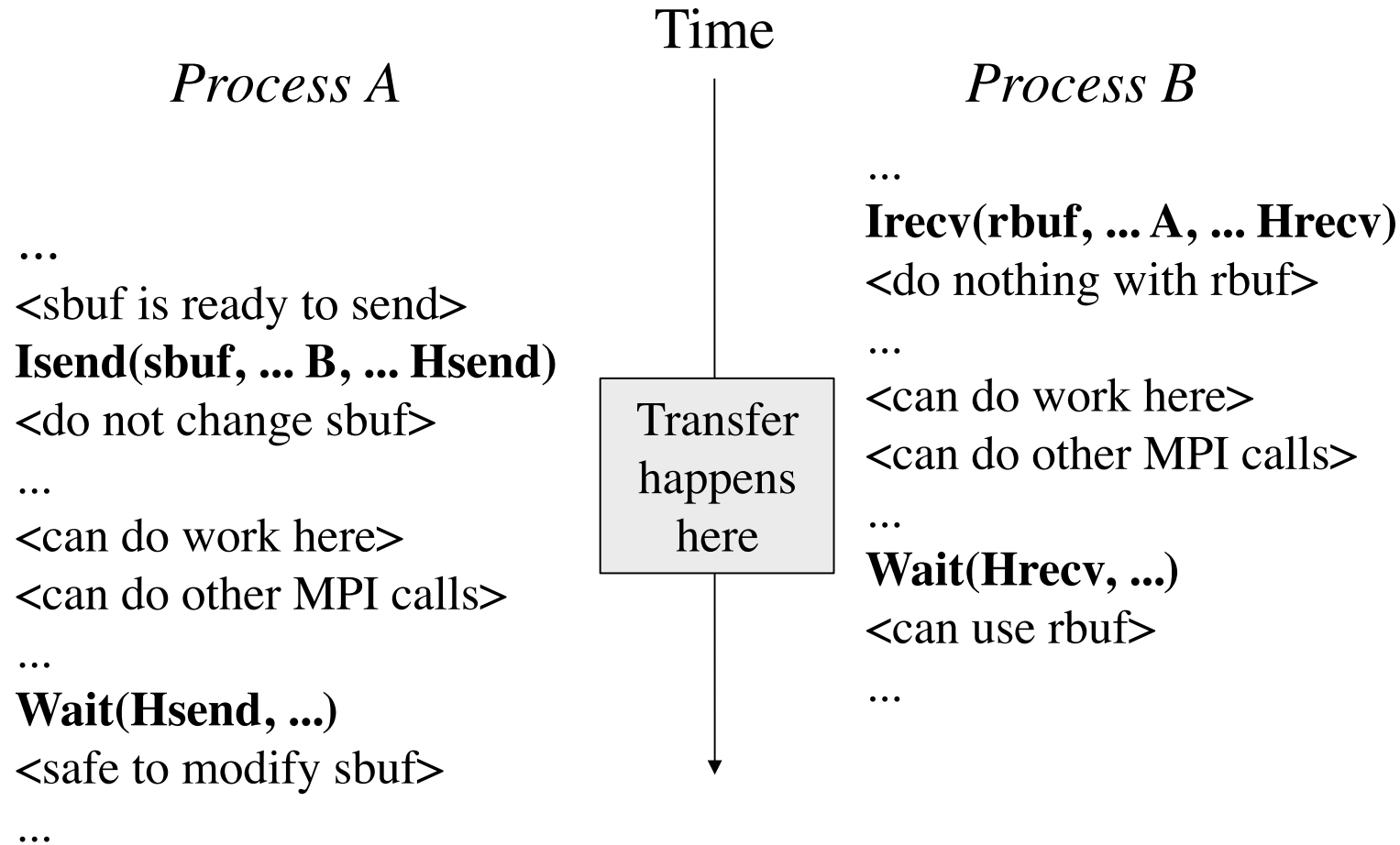
- A blocking send routine will only "return" after it is safe to modify the application buffer (your send data) for reuse. Safe means that modifications will not affect the data intended for the receive task. Safe does not imply that the data was actually received - it may very well be sitting in a system buffer.
- A blocking send can be synchronous which means there is handshaking occurring with the receive task to confirm a safe send.
- A blocking send can be asynchronous if a system buffer is used to hold the data for eventual delivery to the receive.
- A blocking receive only "returns" after the data has arrived and is ready for use by the program.



Non-blocking:

- Non-blocking send and receive routines behave similarly - they will return almost immediately. They do not wait for any communication events to complete, such as message copying from user memory to system buffer space or the actual arrival of message.
- Non-blocking operations simply "request" the MPI library to perform the operation when it is able. The user can not predict when that will happen.
- It is unsafe to modify the application buffer (your variable space) until you know for a fact the requested non-blocking operation was actually performed by the library. There are "wait" routines used to do this.
- Non-blocking communications are primarily used to overlap computation with communication and exploit possible performance gains

Non Blocking Communication Example



Illustrates conservative use of standard Isend & Irecv calls

Non-blocking Send: Syntax

C:

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

FORTRAN:

```
subroutine MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG,  
                   COMM, REQUEST, IERROR)
```

- [IN buf] initial address of send buffer (choice)
- [IN count] number of elements in send buffer (integer)
- [IN datatype] datatype of each send buffer element (handle)
- [IN dest] rank of destination (integer)
- [IN tag] message tag (integer)
- [IN comm] communicator (handle)
- [OUT request] communication request (handle)

Non-blocking Recv: Syntax

C:

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

FORTRAN:

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

[OUT buf] initial address of receive buffer (choice)

[IN count] number of elements in receive buffer (integer)

[IN datatype] datatype of each receive buffer element (handle)

[IN dest] rank of source (integer)

[IN tag] message tag (integer)

[IN comm] communicator (handle)

[OUT request] communication request (handle)

Non-blocking Communication: completion calls

- Wait: MPI_WAIT or MPI_WAITALL
 - Used for non-blocking Sends and Receives
 - Suspends until an operation completes
 - MPI_WAIT syntax
 - Fortran MPI_WAIT (request, status, ierror)
 - C: MPI_Wait (request, status)
- Test: MPI_TEST
 - Returns immediately with information about the status of an operation.
 - MPI_TEST Syntax
 - Fortran: MPI_TEST (request, flag, status, ierror)
 - C: MPI_Test (request, flag, status)

Non-blocking Communication: completion calls

A request object can be deallocated without waiting for the associated communication to complete, by using the following operation:

`MPI_REQUEST_FREE(request)`

[INOUT request] communication request (handle)

C: `int MPI_Request_free(MPI_Request *request)`

FORTTRAN: `MPI_REQUEST_FREE(REQUEST, IERROR)`

Non-blocking Communication – Good Example

- Example: Simple usage of nonblocking operations and MPI_WAIT

```
IF(rank .EQ. 0) THEN
```

```
    CALL MPI_ISEND(a(1), 10, MPI_REAL, 1, tag, comm, request, ierr)
```

```
    ****do computation while communication is happening****
```

```
    CALL MPI_WAIT(request, status, ierr)
```

```
ELSE
```

```
    CALL MPI_IRECV(a(1), 10, MPI_REAL, 0, tag, comm, Request, ierr)
```

```
    ****do computation while communication is happening****
```

```
    CALL MPI_WAIT(Request, status, ierr)
```

```
END IF
```

Non-blocking Communication - Bad Example

```
IF(rank .EQ. 0) THEN
    CALL MPI_ISEND(outval, 1, MPI_REAL, 1, Tag1, comm, req, ierr)
    CALL MPI_REQUEST_FREE(req, ierr)
    CALL MPI_Irecv(inval, 1, MPI_REAL, 1, Tag2, comm, req, ierr)
    CALL MPI_WAIT(req, status, ierr)
ELSE ! rank.EQ.1
    CALL MPI_Irecv(inval, 1, MPI_REAL, 0, Tag1, comm, req, ierr)
    CALL MPI_WAIT(req, status, ierr)
    CALL MPI_ISEND(outval, 1, MPI_REAL, 0, Tag2, comm, req, ierr)
    CALL MPI_REQUEST_FREE(req, ierr)
END IF
```

- Problem: no distinction among the requests, rank 1 may wedge

Non-blocking Communication – Complete Example

```
INCLUDE "mpif.h"
INTEGER ierror, rank, size, status(MPI_STATUS_SIZE), requests (2)
LOGICAL FLAG

CALL MPI_INIT(ierror)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, np, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
IF(rank .eq. 0) THEN
    c = 9.2
    a = 4.2
    b = 8.4
    CALL MPI_ISEND(a, 1, MPI_REAL, 1, 101, MPI_COMM_WORLD, requests(1), ierror)
    ! Here one can do some computation, which will not store/write a
    b=b*a
    CALL MPI_WAIT(requests(1), status, ierror)
    a=b*c
ELSE
    a = 14.2
    b = 18.4
    CALL MPI_IRECV(c, 1, MPI_REAL, 0, 101, MPI_COMM_WORLD, requests(2), ierror)
    ! Here one can do some computation which will not read or write c
    b = b + a
    CALL MPI_WAIT(requests(2), status, ierror)
    c = a + c
END IF
CALL MPI_FINALIZE(ierror)
STOP
END
```

Non-blocking Communication

Benefits:

- Overlap communication with computation
- Post non-blocking sends/receives early and do waits late
- Avoid Deadlocks
- Decrease Synchronization Overhead
- Can Reduce Systems Overhead

Considerations

- It is recommended to set the MPI_Irecv before the MPI_Isend is called.
- Be careful with reads and writes
 - Avoid writing to send buffer between MPI_Isend and MPI_Wait
 - Avoid reading and writing in receive buffer between MPI_Irecv and MPI_Wait



MPI

Collective Communication

MPI Collective Communication

Three Classes of Collective Operations

Data movement: broadcast, gather, all-gather, scatter, and all-to-all

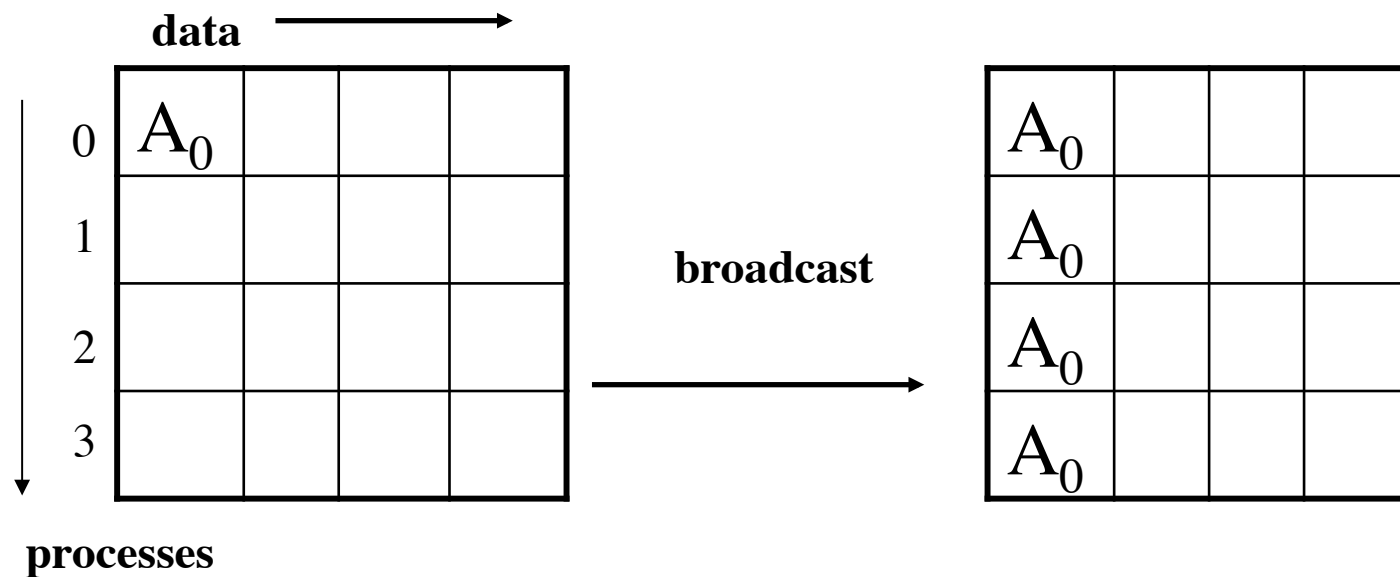
Collective computations (reductions): one member of the group collects data from the others and performs an operation (e.g., max, min) on the data.

Synchronization: process wait until all members of the group have reached the synchronization point

Every process must call the same collective communication function.

MPI Collective Communication: Broadcast

One processor needs to send (broadcast) some data (either a scalar or vector) to all the processes in a group.



MPI Collective Communication: Broadcast

Syntax

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

Fortran: `MPI_BCAST (buffer, count, datatype, root, comm, ierr)`

where:

buffer: is the starting address of a buffer

count: is an integer indicating the number of data elements in the buffer

datatype: is MPI defined constant indicating the data type of the elements in the buffer

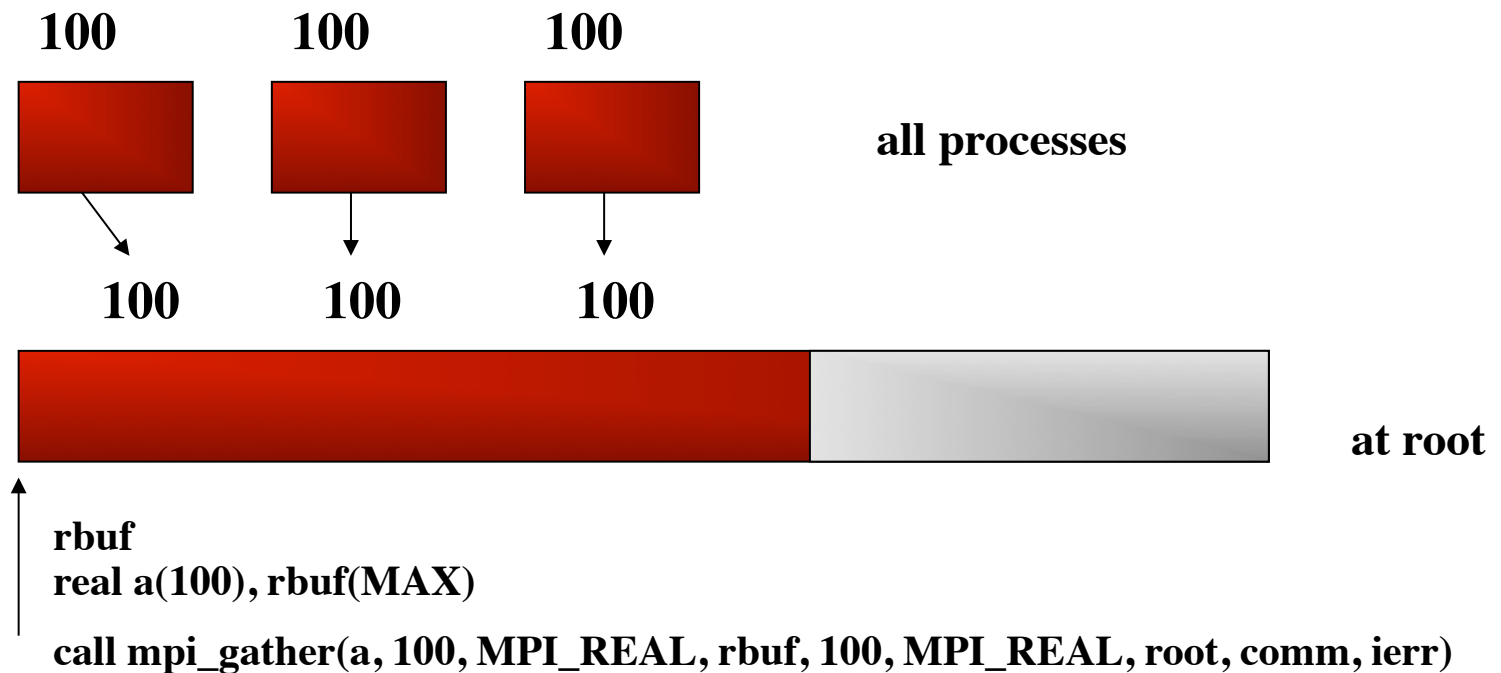
root: is an integer indicating the rank of broadcast root process

comm: is the communicator

The `MPI_BCAST` must be called by each process in the group, specifying the same `comm` and `root`. The message is sent from the root process to all processes in the group, including the root process.

MPI Collective Communication: Gather and Scatter

Often, an array is distributed throughout all processors in the group, and one wants to collect each piece of the array into a specified process.



MPI Collective Communication: Gather

Syntax

C:

```
int MPI_Gather(void* sbuf, int scount, MPI_Datatype stype, void* rbuf, int rcount,  
              MPI_Datatype rtype, int root, MPI_Comm comm)
```

FORTTRAN:

```
MPI_GATHER (sbuf, scount, stype, rbuf, rcount, rtype, root, comm, ierr)
```

where:

sbuf:	is the starting address of a buffer,
scount:	is the number of elements in the send buffer,
stype:	is the data type of send buffer elements,
rbuf:	is the address of the receive buffer
rcount:	is the number of elements for any single receive
rtype:	is the data type of the receive buffer elements
root:	is the rank of receiving process, and
comm:	is the communicator
ierr:	is error message

MPI Collective Communication: Gather Example

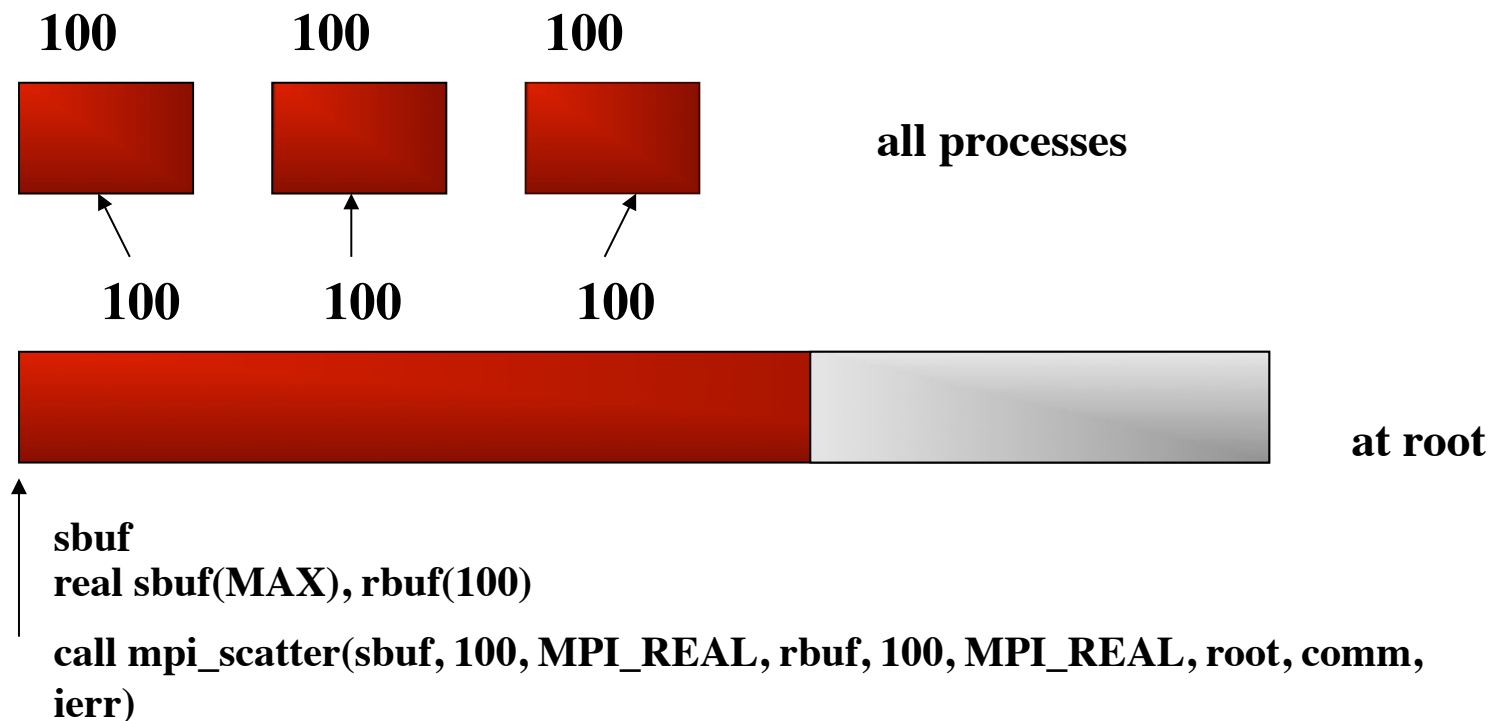
```
INCLUDE 'mpif.h'
DIMENSION A(25, 100), b(100), cpart(25), ctotat(100)
INTEGER root, rank
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, np, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
root=1
A=10*rank+0.5
B=0.5
DO I=1,25
  cpart(I)=0.
  DO K=1,100
    cpart(I)=cpart(I)+A(I,K)*b(K)
  END DO
END DO
CALL MPI_GATHER (cpart, 25, MPI_REAL, ctotat, 25,
&MPI_REAL, root, MPI_COMM_WORLD, ierr)
If(rank.eq.root) print*, (ctotat(I),I=15,100,25)
CALL MPI_FINALIZE(ierr)
END
```

$$\begin{array}{c} \mathbf{A} \\ \hline \text{Processor 1} \\ \hline \text{Processor 2} \\ \hline \text{Processor 3} \\ \hline \text{Processor 4} \end{array} * \mathbf{b} = \mathbf{c} \begin{array}{c} \hline 1 \\ \hline 2 \\ \hline 3 \\ \hline 4 \end{array}$$

- A: Matrix distributed by rows
- B: Vector shared by all process
- C: results to get by the root process

MPI Collective Communication: Scatter

One process wants to distribute the data of equal segment to all the processes in a group.



MPI Collective Communication: Scatter Syntax

C:

```
int MPI_Scatter(void* sbuf, int scount, MPI_Datatype stype, void* rbuf,  
               int rcount, MPI_Datatype rtype, int root, MPI_Comm comm)
```

FORTTRAN:

```
MPI_SCATTER(sbuf, scount, stype, rbuf, rcount, rtype, root, comm, ierr)
```

where:

sbuf:	is the address of the send buffer,
scount:	is the number of elements sent to each process,
stype:	is the data type of the send buffer elements,
rbuf:	is the address of the receive buffer,
rcount:	is the number of elements in the receive buffer,
rtype:	is the data type of the receive buffer elements,
root:	is the rank of the sending process, and
comm:	is the communicator

Note: sbuf is significant for root process only

```

PROGRAM scatter
INCLUDE 'mpif.h'
INTEGER isend(3)
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)

CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
IF (myrank == 0) THEN
    DO i=1, nprocs
        isend(i) = i
    ENDDO
end if
CALL MPI_SCATTER (isend, 1, MPI_INTEGER,
&                irecv, 1, MPI_INTEGER, 0,
&                MPI_COMM_WORLD, ierr)
PRINT *, 'irecv = ', irecv
CALL MPI_FINALIZE(ierr)
END

```

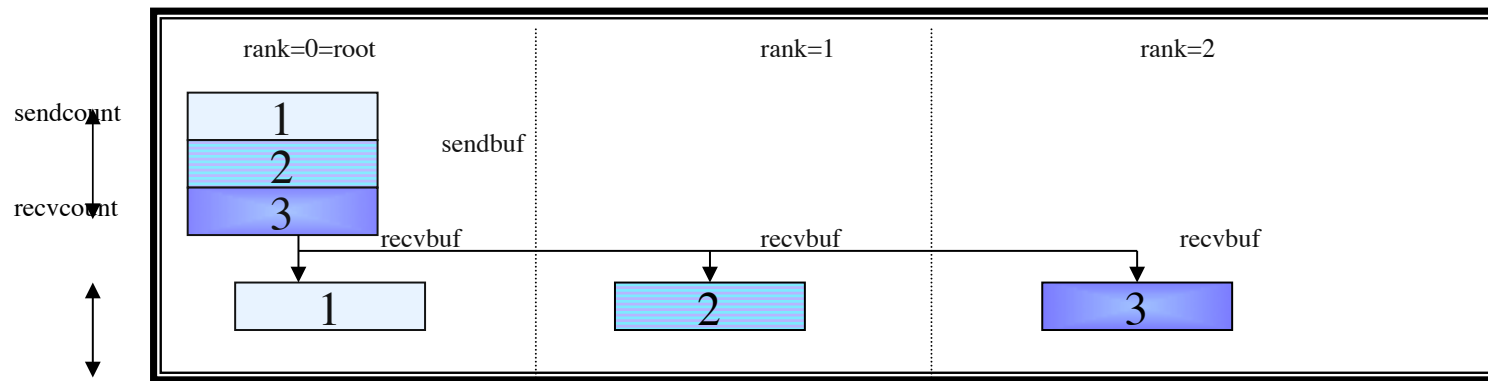
Sample execution

\$ a.out -procs 3

0: irecv = 1

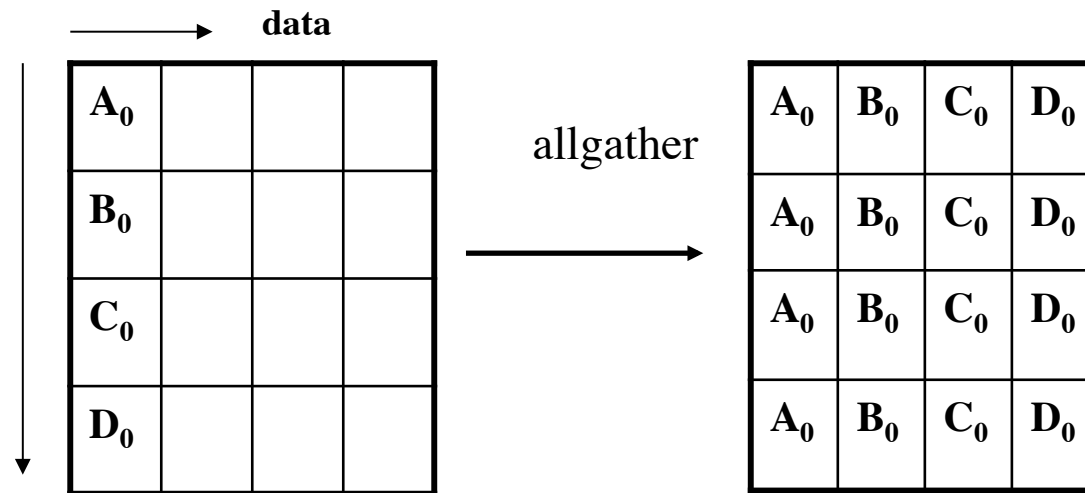
1: irecv = 2

2: irecv = 3



MPI Collective Communication: All Gather

- MPI_ALLGATHER can be thought of as MPI GATHER where all processes, not only one, receive the result.



processes

- The syntax of MPI_ALLGATHER is similar to MPI_GATHER. However, the argument root is dropped.

MPI Collective Communication: All Gather Syntax

C:

```
int MPI_Allgather (void* sbuf, int scout, MPI_Datatype stype, void* rbuf, int rcount,  
                  MPI_Datatype rtype, MPI_Comm comm
```

FORTTRAN

```
MPI_ALLGATHER (sbuf, scout, stype, rbuf, rcount, rtype, comm, ierr)
```

Example: back to the previous “gather” example, what should we do if every process needs the results of array Ctotal for next computation?

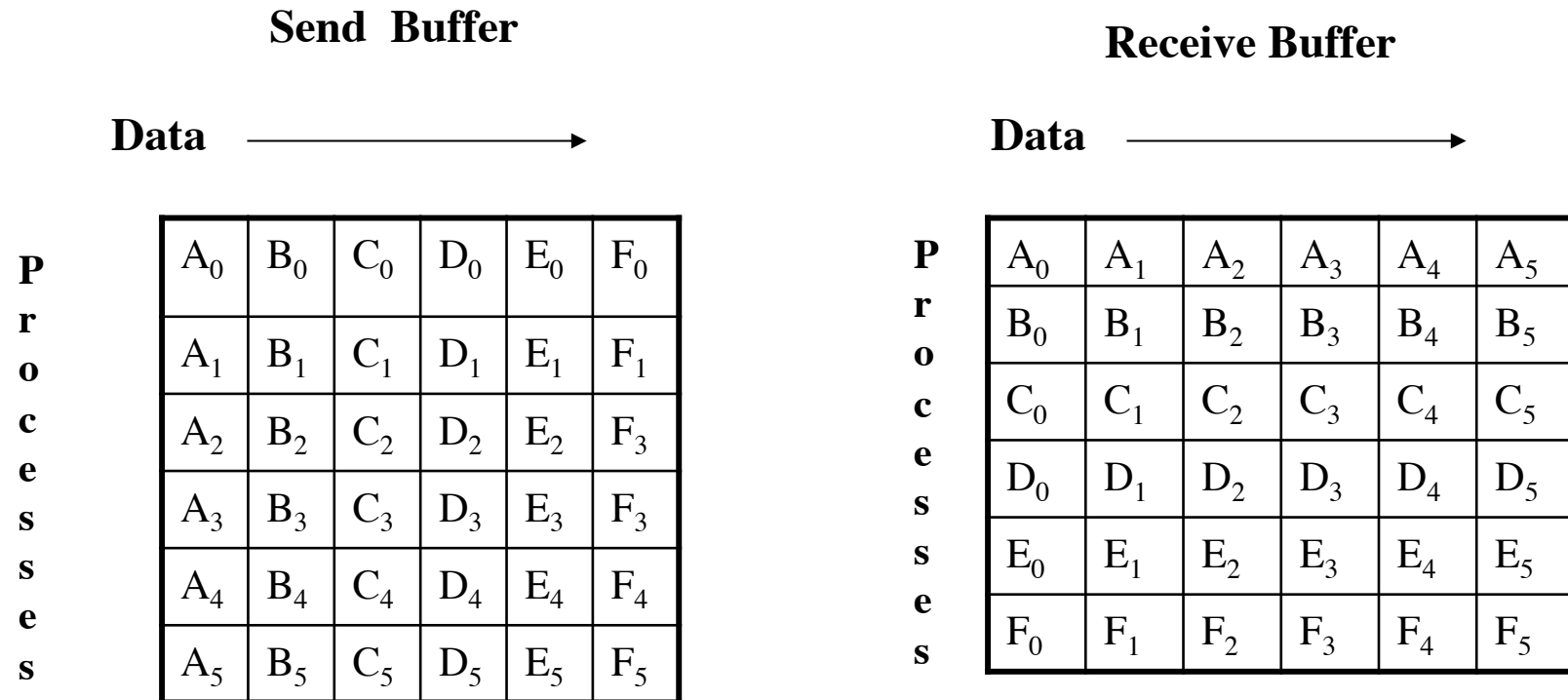
Replcace

```
CALL MPI_GATHER (cpart, 25, MPI_REAL, ctotal, 25, \  
MPI_REAL, root, MPI_COMM_WORLD, ierr)  
print*, (ctotal(I),I=15,100,25)
```

With

```
CALL MPI_ALLGATHER (cpart, 25, MPI_REAL, ctotal, 25, \  
MPI_REAL, MPI_COMM_WORLD, ierr)  
print*, (ctotal(I),I=15,100,25)
```


MPI Collective Communication: Alltoall



MPI Collective Communication: Alltoall Syntax

C:

```
int MPI_Alltoall(void* sbuf, int scout, MPI_Datatype stype, void* rbuf,  
                int rcount, MPI_Datatype rtype, MPI_Comm comm )
```

FORTTRAN:

```
MPI_ALLTOALL (sbuf, scout, stype, rbuf, rcount, rtype, comm, ierr)
```

where:

sbuf:	is the starting address of send buffer,
scount:	is the number of elements sent to each process,
stype:	is the data type of send buffer elements,
rbuf:	is the address of receive buffer,
rcount:	is the number of elements received from any process,
rtype:	is the data type of receive buffer elements, and
comm:	is the group communicator.

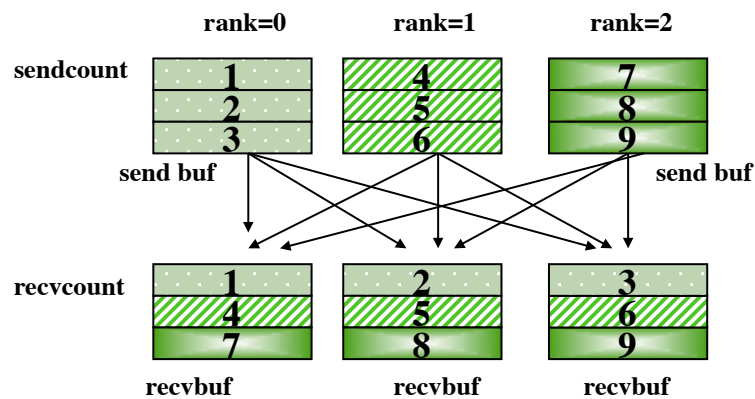


Figure of MPI_ALLTOALL

\$ a.out -procs 3

```
0: isend 1 2 3
1: isend 4 5 6
2: isend 7 8 9
0: irecv 1 4 7
1: irecv 2 5 8
2: irecv 3 6 9
```

```
PROGRAM alltoall
INCLUDE 'mpif.h'
INTEGER isend (3), irecv (3)
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs,
&ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD,
&myrank, ierr)
DO i=1, nprocs
    isend (i) = i + nprocs * myrank
ENDDO
PRINT *, 'isend =', isend
CALL MPI_ALLTOALL(isend, 1, MPI_INTEGER,
&
    irecv, 1, MPI_INTEGER,
&
    MPI_COMM_WORLD, ierr)
PRINT *, 'irecv =', irecv
CALL MPI_FINALIZE(ierr)
END
```

Hands-on

http://www.msi.umn.edu/tutorial/scicomp/general/MPI/workshop_MPI

Get example and build


```
ssh calhoun.msi.umn.edu  
cp -r /soft/examples/hello_mpi_intel_ompi .  
cd hello_mpi_intel_ompi  
module load intel ompi/intel  
make
```

Run interactively on login node

```
mpirun -np 4 ./hello
```

Run in PBS queue

```
qsub run.pbs
```



MPI

Collective Computations and Synchronization



UNIVERSITY OF MINNESOTA
Driven to DiscoverSM

MPI_Reduce

- Collective computation routines perform a global operation across all members of a group
- The partial result in each process in the group is combined in one specified process or all the processes using some desired function.
- Three reduces routines:
 - **MPI_REDUCE** returns results to a single process;
 - **MPI_ALLREDUCE** returns results to all processes in the group;
 - **MPI_REDUCE_SCATTER** scatters a vector, which results in a reduce operation, across all processes.



MPI Predefined Reduce Operations

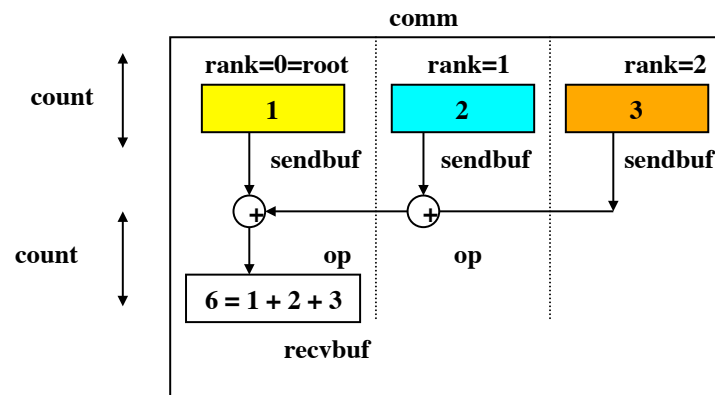
	Name	Meaning	C type	FORTTRAN type
MPI_MAX		maximum	integer, float	integer, real, complex
MPI_MIN		minimum	integer, float	integer, real, complex
MPI_SUM		sum	integer, float	integer, real, complex
MPI_PROD		product	integer, float	integer, real, complex
MPI LAND		logical and	integer	logical
MPI_BAND		bit-wise and	integer, MPI_BYTE	integer, MPI_BYTE
MPI_LOR		logical or	integer	logical
MPI BOR		bit-wise or	integer, MPI_BYTE	integer, MPI_BYTE
MPI_LXOR		logical xor	integer	logical
MPI_BXOR		bit-wise xor	integer MPI_BYTE	integer, MPI_BYTE
MPI_MAXLOC		max value and location	combination of int, float, double, and long double	combination of integer, real, complex, double precision
MPI_MINLOC		min value and location	combination of int, float, double, and long double	combination of integer, real complex, double precision

MPI_REDUCE

Usage: CALL **MPI_REDUCE** (sendbuf, recvbuf, count, datatype, op, root, comm, ierror)

Parameters

(CHOICE) **sendbuf** The address of the send buffer (IN)
(CHOICE) **recvbuf** The address of the receive buffer, sendbuf and recvbuf cannot overlap in memory. (significant only at root) (OUT)
INTEGER **count** The number of elements in the send buffer (IN)
INTEGER **datatype** The data type of elements of the send buffer (handle) (IN)
INTEGER **op** The reduction operation (handle) (IN)
INTEGER **root** The rank of the root process (IN)
INTEGER **comm** The communicator (handle) (IN)
INTEGER **ierror** The Fortran return code



Sample Program

```
PROGRAM reduce
  include 'mpif.h'
  integer sendbuf, recvbuf, nprocs, myrank
  CALL MPI_INIT (ierr)
  CALL MPI_COMM_SIZE (MPI_COMM_WORLD, nprocs, ierr)
  CALL MPI_COMM_RANK (MPI_COMM_WORLD, myrank, ierr)
  sendbuf= myrank+1
  CALL MPI_REDUCE (sendbuf, recvbuf, 1, MPI_INTEGER,
    &                MPI_SUM, 0, MPI_COMM_WORLD, ierr)
  IF (myrank==0) THEN
    PRINT *, 'recvbuf =', recvbuf
  endif
  CALL MPI_FINALIZE (ierr)
END
```

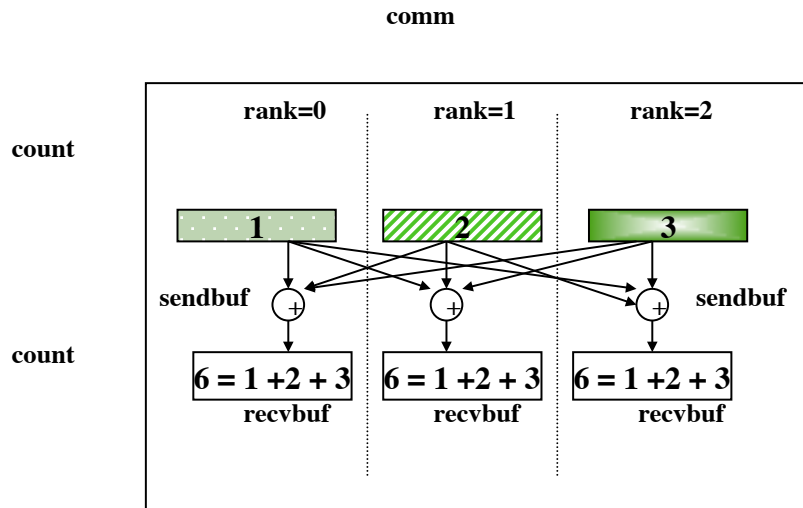
Sample execution

```
% mpirun -np 3 ./a.out
```

```
% recvbuf = 6
```


MPI_ALLREDUCE

Usage: CALL **MPI_ALLREDUCE** (sendbuf, recvbuf, count, datatype, op, comm, ierror)



Parameters

(CHOICE) sendbuf	The starting address of the send buffer (IN)
(CHOICE) recvbuf	The starting address of the receive buffer,, sendbuf and recvbuf cannot overlap in memory (OUT)
INTEGER count	The number of elements in the send buffer (IN)
INTEGER datatype	The data type of elements of the send buffer (handle) (IN)
INTEGER op	The reduction operation (handle)(IN)
INTEGER comm	The communicator (handle) (IN)
INTEGER ierror	The Fortran return code

Sample program

```

PROGRAM allreduce
  INCLUDE 'mpif.h'
  integer sendbuf, recvbuf, nprocs, myrank
  CALL MPI_INIT (ierr)
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
  CALL MPI_COMM_RANK (MPI_COMM_WORLD, myrank, ierr)
      sendbuf = myrank + 1
  CALL MPI_ALLREDUCE (sendbuf, recvbuf, 1, MPI_INTEGER,
    MPI_SUM,
    &      MPI_COMM_WORLD, ierr)
  PRINT *, 'recvbuf =', recvbuf
  CALL MPI_FINALIZE (ierr)
END

```

Sample execution

```

$ mpirun -np 3 ./a.out
      recvbuf = 6
      recvbuf = 6
      recvbuf = 6

```

MPI_REDUCE_SCATTER

Usage: CALL **MPI_REDUCE_SCATTER** (sendbuf, recvbuf, recvcnt, datatype, op, comm, ierror)

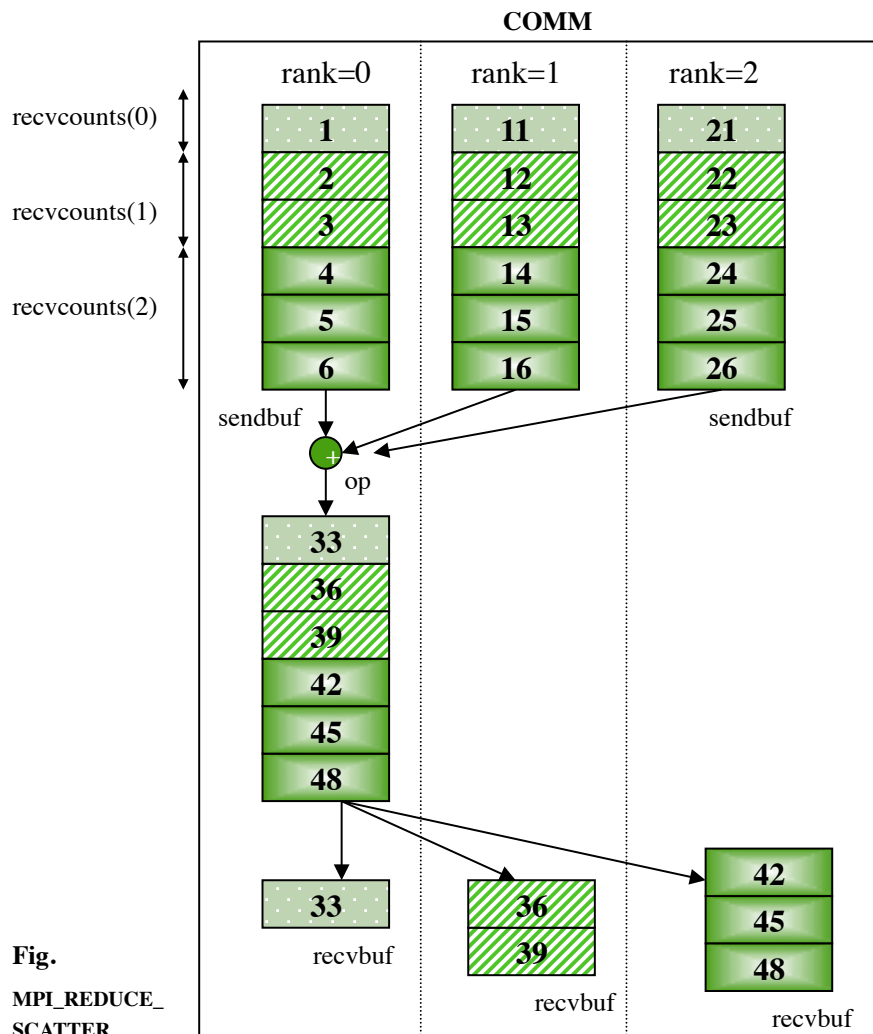


Fig.
MPI_REDUCE_
SCATTER

Sample Program

```

PROGRAM reduce_scatter
INCLUDE 'mpif.h'
INTEGER sendbuf (6), recvbuf (3)
INTEGER recvcnt(0:2)
DATA recvcnt/1,2,3/
CALL MPI_INIT (ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
DO i=1,6
    sendbuf (i) = i + myrank * 10
ENDDO
CALL MPI_REDUCE_SCATTER(sendbuf, recvbuf, recvcnt,
& MPI_INTEGER, MPI_SUM, MPI_COMM_WORLD, ierr)
PRINT *, 'recvbuf =', recvbuf
CALL MPI_FINALIZE(ierr)
END
    
```

\$ mpirun -np 3 ./a.out

```

recvbuf = 33  0  0
recvbuf = 36 39  0
recvbuf = 42 45 48
    
```

MPI_REDUCE_SCATTER

Usage: CALL MPI_REDUCE_SCATTER(sendbuf, recvbuf, recvcnts, datatype, op, comm, ierror)

Parameters:

(CHOICE) sendbuf	The starting address of the send buffer (IN)
(CHOICE) recvbuf	The starting address of the receive buffer, sendbuf and recvbuf cannot overlap in memory. (OUT)
INTEGER recvcnts (*)	Integer array specifying the number of elements in result distributed to each process. Must be identical on all calling processes. (IN)
INTEGER datatype	The data type of elements of the input buffer (handle) (IN)
INTEGER op	The reduction operation (handle) (IN)
INTEGER comm	The communicator (handle) (IN)
INTEGER ierror	The Fortran return code

Description:

MPI_REDUCE_SCATTER first performs an element-wise reduction on vector of count = \sum , recvcnts(i) elements in the send buffer defined by sendbuf, count and datatype. Next, the resulting vector is split into n disjoint segments, where n is the number of members in the group. Segment i contains recvcnts(i) elements. the ith segment is sent to process I and stored in the receive buffer defined by recvbuf, recvcnts(i) and datatype. MPI_REDUCE_SCATTER is functionally equivalent to MPI_REDUCE with count equal to the sum of recvcnts(i) followed by MPI_SCATTERV with sendcounts equal to recvcnts. All processes in comm need to call this routine.

MPI_SCAN

Scan

A scan or prefix-reduction operation performs partial reductions on distributed data.

C: int MPI_Scan (void* sbuf, void* rbuf, int count,
 MPI_Datatype datatype, MPI_OP op, MPI_Comm comm)

FORTTRAN: MPI_SCAN (sbuf, rbuf, count, datatype, op, comm, ierr)

Where:

sbuf: is the starting address of the send buffer,
rbuf: is the starting address of receive buffer,
count: is the number of elements in input buffer,
datatype: is the data type of elements of input buffer
op: is the operation, and
comm: is the group communicator.

MPI_SCAN

Usage: CALL **MPI_SCAN** (sendbuf, recvbuf, count, datatype, op, comm, ierror)

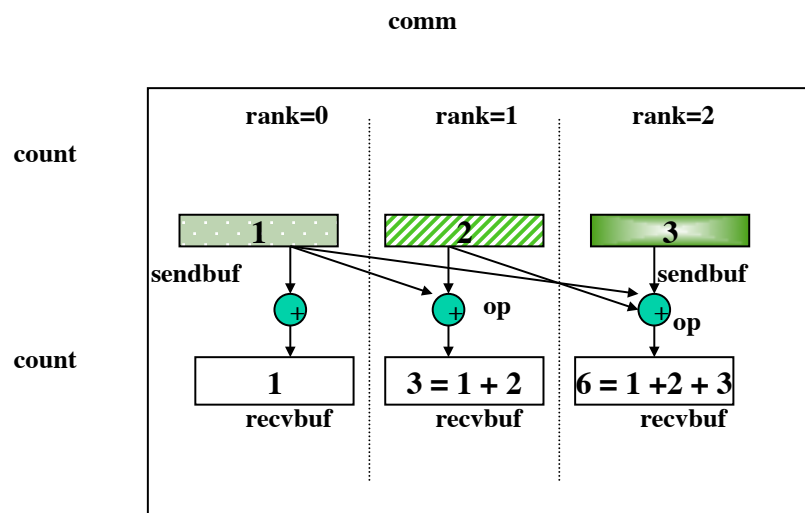


Figure. MPI_SCAN

Parameters

(CHOICE) sendbuf	The starting address of the send buffer (IN)
(CHOICE) recvbuf	The starting address of the receive buffer, sendbuf and recvbuf cannot overlap in memory (OUT)
INTEGER count	The number of elements in sendbuf (IN)
INTEGER datatype	The data type of elements of sendbuf (handle) (IN)
INTEGER op	The reduction operation (handle) (IN)
INTEGER comm	The communicator (handle) (IN)
INTEGER ierror	The Fortran return code

Sample program

```

PROGRAM scan
  INCLUDE 'mpif.h'
  integer sendbuf, recvbuf
  CALL MPI_INIT (ierr)
  CALL MPI_COMM_SIZE (MPI_COMM_WORLD, nprocs, ierr)
  CALL MPI_COMM_RANK (MPI_COMM_WORLD, myrank, ierr)
  sendbuf = myrank + 1
  CALL MPI_SCAN (sendbuf, recvbuf, 1, MPI_INTEGER,
    &
    MPI_SUM, MPI_COMM_WORLD, ierr)
  PRINT *, 'recvbuf = ' recvbuf
  CALL MPI_FINALIZE(ierr)
END
  
```

Sample execution

```

$ mpirun -np 3 ./a.out
recvbuf = 1
recv buf = 3
recvbuf = 6
  
```

Barrier Synchronization

- Two types of synchronization:
 - Implicit synchronization
 - Explicit synchronization: `MPI_BARRIER`
- MPI provides a function call, `MPI_BARRIER`, to synchronize all processes within a communicator.
- A barrier is simply a synchronization primitive. A node calling it will be blocked until all the nodes within the group have called it.



Barrier Synchronization

The syntax of MPI_BARRIER for both C and Fortran program is:

- C:

MPI_Barrier (MPI_Comm MPI_Comm_world)

- FORTRAN

MPI_BARRIER (MPI_Comm_world, ierror)

where:

MPI_Comm: is an MPI predefined structure of communicators,

MPI_Comm_world: is an integer denoting a communicator

ierror: is an integer return error code.

Other MPI Topics

- **MPI Group management and Communicators**
- **Derived data types with MPI**
 - Contiguous
 - Vector and Hvector
 - Indexed and Hindexed
 - Struct
- **MPI2**
 - Dynamic process management
 - One sided communication
 - Cooperative I/O
 - C++ Bindings, Fortran 90 additions
 - Extended collective operations
 - Miscellaneous other functionality

Hands-on

http://www.msi.umn.edu/tutorial/scicomp/general/MPI/workshop_MPI

Get example and build

```
ssh calhoun.msi.umn.edu
cp -r /soft/examples/hello_mpi_intel_ompi .
cd hello_mpi_intel_ompi
module load intel_ompi/intel
make
```

Run interactively on login node

```
mpirun -np 4 ./hello
```

Run in PBS queue

```
qsub run.pbs
```

THANK YOU

More info at
www.msi.umn.edu
612-626-0802

The University of Minnesota is an equal opportunity educator and employer. This PowerPoint is available in alternative formats upon request. Direct requests to Minnesota Supercomputing Institute, 599 Walter library, 117 Pleasant St. SE, Minneapolis, Minnesota, 55455, 612-624-0528.

© 2009 Regents of the University of Minnesota. All rights reserved.

Supercomputing Institute
for Advanced Computational Research



UNIVERSITY OF MINNESOTA
Driven to DiscoverSM