Parallel Programming Using MPI

David Porter & Drew Gustafson

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

October 20, 2016

acroread /home/dhp/public/mpi.pdf

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

Mesabi

HP Linux Cluster

750+ compute nodes
Each node has 2 x 12-core 2.5 GHz Intel Haswell processors
18,750+ cores
711+ Tflop aggregate performance

From 64 GB to 1 TB of memory per node
Aggregate memory: 67+ TB of RAM

40 GPU nodes:
2 Nvidia Tesla K40 GPUs

FDR/EDR Infiniband interconnect
➔ 5+ GB/s node-to-node communication

IB connect to Panasuas global file system

- https://www.msi.umn.edu/content/mesabi

Itasca

- https://www.msi.umn.edu/content/itasca

HP Linux Cluster

1091 compute nodes
  2 quad-core 2.8 GHz Intel Nehalem processors
  24 GB of memory per node

Total of 8,728 cores
Aggregate of 26 TB of RAM

QDR Infiniband interconnect
➔3+ GB/s none-to-node communication

Introduction to parallel programming

Serial ➔ one statement at a time
one thread of execution, and
one process

Parallel ➔ multiple concurrent statements
multiple threads of execution, and/or
one or more processes

Parallel Programming

Involves:
Decomposing work into many tasks
Distributing tasks to multiple threads or processes
Threads/processes work simultaneously
Coordinating work and communication of threads

Considerations
Type of parallel architecture being used
Type of communication needed between tasks

Parallel Programming

Uses

Multiple processors & threads
Multiple cores
Network (distributed memory machines, cluster, etc.)
Environment to create and manage parallel processing
Operating System

Parallel Programming Paradigms

Distributed memory:	multiple processes	MPI
Shared Memory:	multiple threads	OpenMP

Hardware Considerations

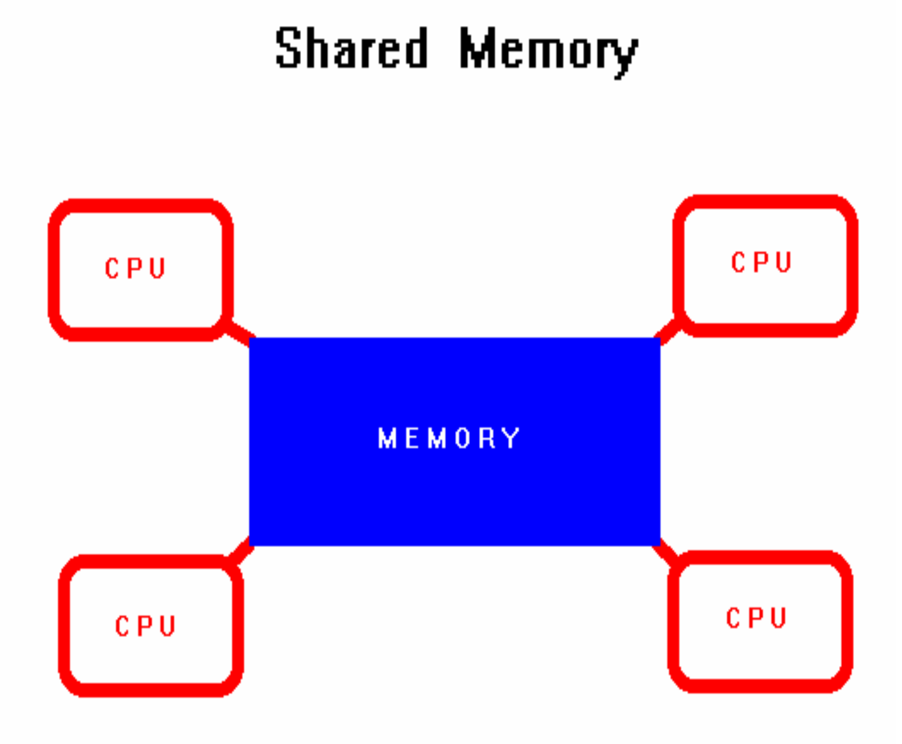
Memory architectures

Shared Memory (NUMA)
Distributed Memory
Cluster of Shared Memory "nodes"

Inter-node communication is required to:

Convey information and data between nodes
Start, stop, & synchronize processes across nodes

Shared Memory



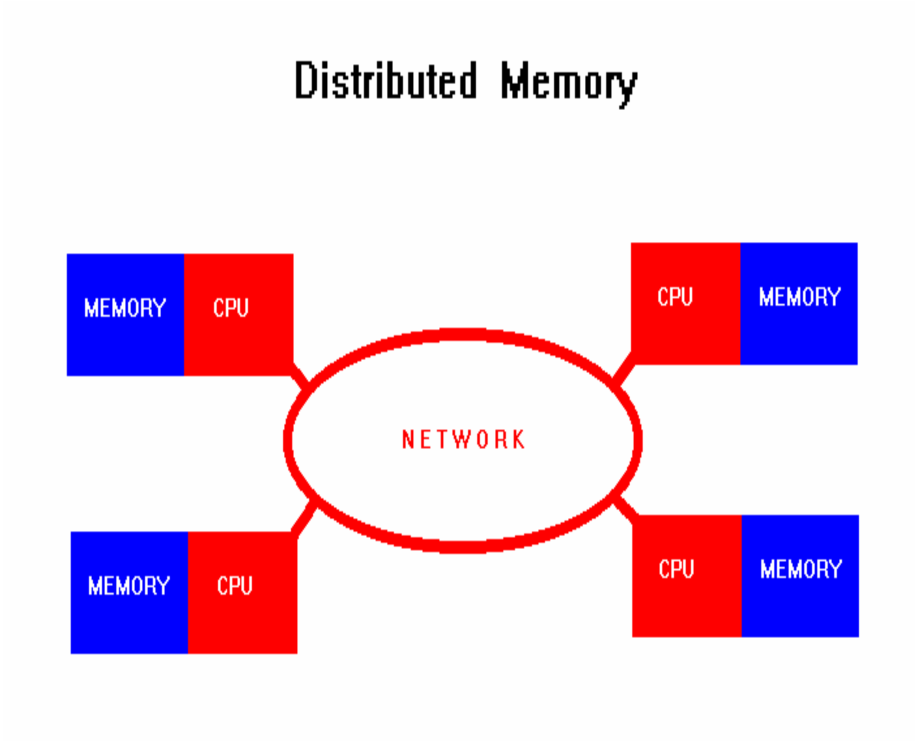
Only one processor can access the shared memory location at a time.

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, Total memory limited to one node

Distributed Memory



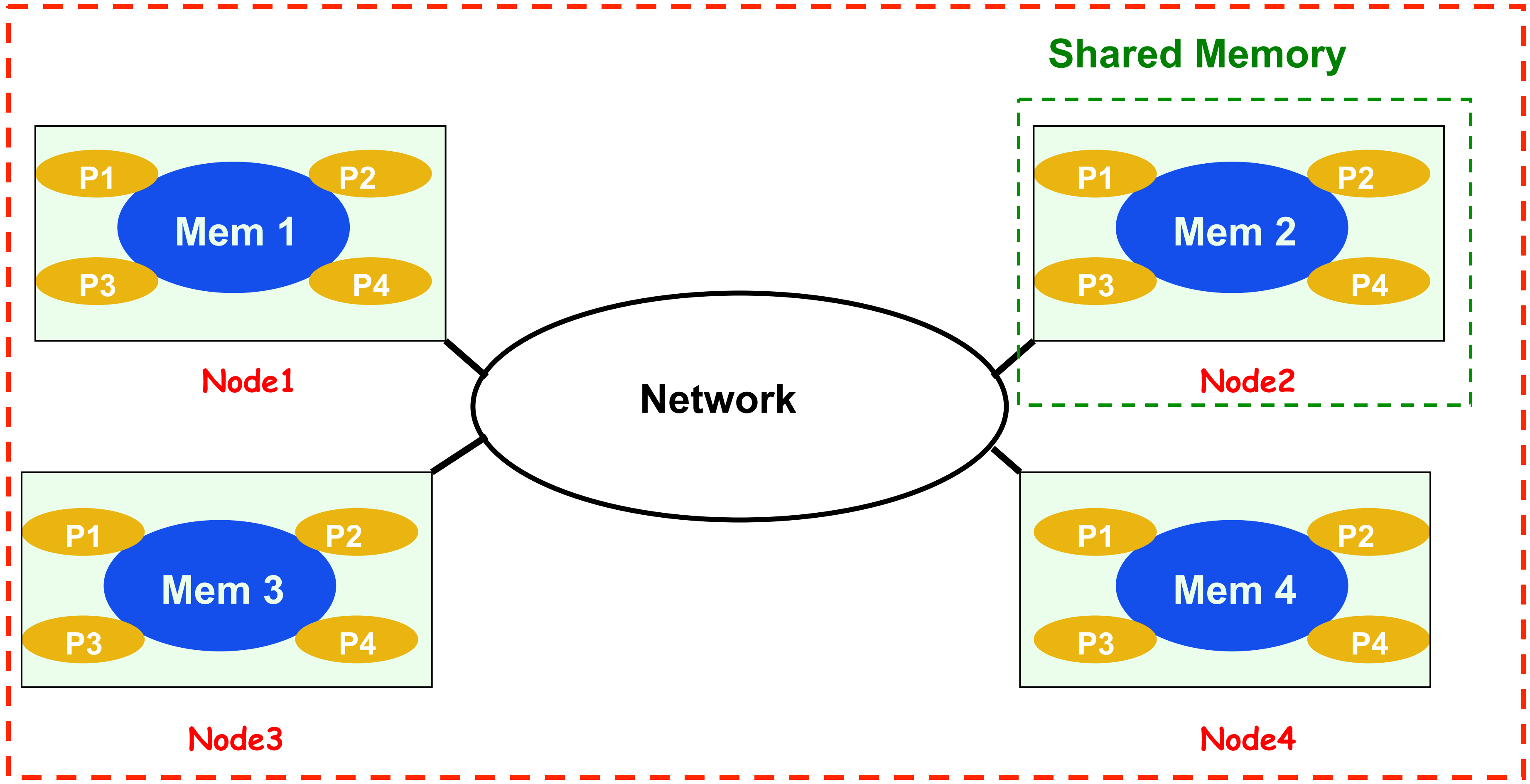
Data is shared across network using message passing

User code drives communication

Advantages:
Scalability, Each processor can rapidly access its own memory without interference

Disadvantages:
Programmer responsible for send/receive data between processes

Compute Cluster



Message Passing

MPI : Message Passing Interface

- A message passing library specification
- Model for distributed memory platforms
- Not a compiler
- For multi-core, clusters, and heterogeneous networks
- Permits development of parallel software libraries
- Provides access to advanced parallel hardware
- End uses

  Applications

  Libraries

  Toolkits

MPI

- Widely accepted standard for distributed memory computing
- Support by all major vendors
- Efficient implementations exists for most parallel hardware
- Code that uses MPI is highly portable
- Very extensive and flexible interface that leaves most of the implementation details up to vendors
- Just a small subset of the functions (6 routines) can be used to write many applications

Parallel programming paradigms

SPMD (Single Program Multiple Data)
•All processes follow essentially the same execution path
•Data-driven execution

MPMD (Multiple Programs Multiple Data)
•Master and slave processes follow distinctly different execution paths
•Heterogeneous computing (GPU, PHI, …)

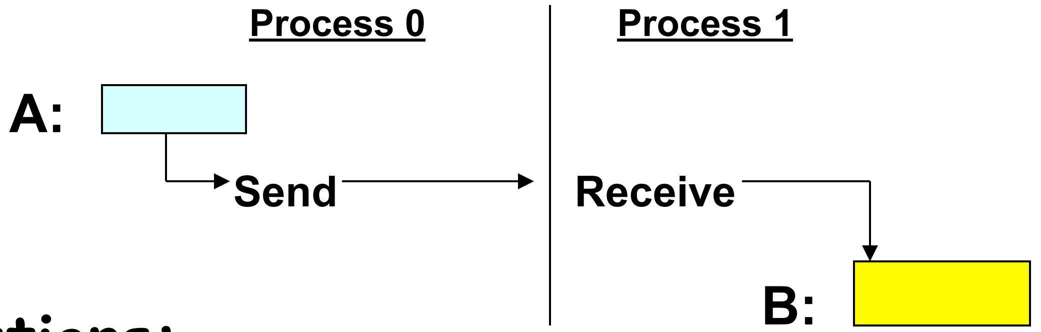
MPI supports both

MPI Blocking Communication

Sending and Receiving Messages

Basic message passing :

One process send a message
Another 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
- data type

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_Xxxxx** (…   )

- Case sensitive (as it always is in C)
- All MPI calls are functions: most return integer error code
- Program must include "mpi.h"
- Parameters are passed by value ➔ pass pointers to data buffers

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)

MPI C Datatypes

MPI datatype	C datatype
MPI_CHAR	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	byte
MPI_PACKED	

MPI Fortran Datatypes

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

MPI Process Identifier

- **MPI Application:** runs on a group of processes.
- **RANK:** one processes in this group
- **Rank NUMBER:** unique number for the process

In MPI communication:

- **Destination** is specified by **rank number**
- **Can point to all ranks:**   `MPI_ANY_SOURCE`

- Processes are named according to their rank in the group
- Can have more than one group in an MPI application
- Groups are pointed to by a “communicator”

MPI Communicators

- A communicator
    denotes a group of processes in an MPI application
- MPI_COMM_WORLD
    predefined communicator
    includes all processes in an MPI application
- New communicators
    can be created in an MPI program
    can point to some or all MPI “ranks
    can point to a re-ordering of ranks
- Most MPI programs only use MPI_COMM_WORLD

MPI Message Tag

Tags allow programmers to
- Organize / classify MPI messages
- Distinguish messages from the same source

The MPI standard guarantees that tags are
- integers in the range 0 ~ 32,767 (at least)
- most implementations allow a much larger range of tags
- upper bound on tag value: `MPI_TAG_UB`

`MPI_ANY_TAG` can be used as a wild card

MPI Blocking Communication Semantics

- MPI_SEND does not complete until buffer is empty (available for reuse)
- MPI_RECV does not complete until buffer is full (available for use)
- Completion of communication generally depends on the message size, system memory & network
- Blocking communication is simple to use but can be slow or cause deadlocks (if you are not careful).
- A blocking or nonblocking send can be paired to a blocking or nonblocking receive

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();
}
```

Hands-on

https://www.msi.umn.edu/content/mpi-hands-workshop

# Get example and build

```
cp -r /home/tech/public/examples/hello_mpi .
cd hello_mpi
module load intel    impi
make
```

# Run interactively

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

# Set the following for large-memory jobs

```
ulimit -s unlimited
```

MPI Non-Blocking Communication

Blocking Communication

MPI_SEND does not complete until buffer is copied out. Time depends on send mode and MPI implementation.

MPI_RECV does not complete until the message is completely received.

Completion of communication generally depends on the message size and the system buffer size.

Blocking communication is simple to use but may be prone to deadlocks.

Initiation of blocking communication may suffer from high latency → poor scaling to many MPI ranks.

Deadlocks

Two or more processes (MPI ranks) wait for each other to act before they act. They each stop before getting to the part of the code where they would have taken the action needed for other ranks to keep going.

Common example:

Two ranks call a blocking mpi_recv to each other.
Each waits for data form the other.
Neither ever sends it.

To avoid deadlocks

Different ordering of calls between ranks
Non-blocking calls
Use of MPI_SendRecv
Buffered mode

Send modes

Standard Mode (MPI_Send)
The standard MPI Send, the send will not complete until it is safe to modify the send buffer: buffer has at least been copied to a supplied system buffer. MPI may or may not buffer: depends on many details.

Synchronous mode (MPI_Ssend)
The send does not complete until after a matching receive has been posted

Buffered mode (MPI_Bsend)
User supplied buffer space is used for system buffering.
The send will complete as soon as the send buffer is copied to the user supplied buffer.

Ready mode (MPI_Rsend)
The send will send eagerly under the assumption that a matching receive has already been posted (an error results otherwise).

Non Blocking Communication

- **Non-blocking calls return immediately**
- **A completion call is needed to ensure the operation is finishes.**

For example:

```
MPI_ISEND( start, count, datatype, dest, tag, comm, request1)
MPI_WAIT( request1, status )

MPI_IRECV( start, count, datatype, src, tag, comm, request2)
MPI_WAIT( request2, status)
```

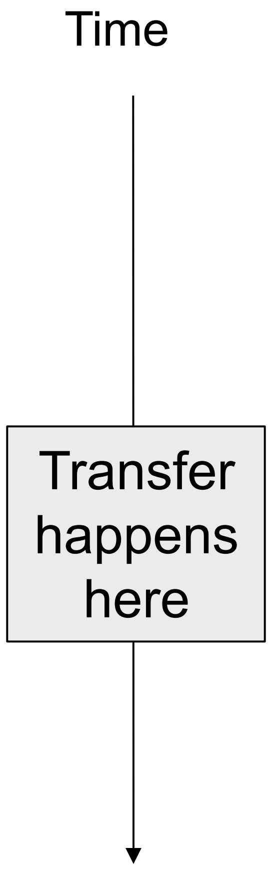
Or use for all non-blocking communications

```
MPI_WAITALL (count, request_array, status_arrsy)
```

One can also test the status without waiting using MPI_TEST

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

Non Blocking Communication Example



Process A

...
<sbuf is ready to send>
Isend(sbuf, ... B, ... Hsend)
<do not change sbuf>
...
<can do work here>
<can do other MPI calls>
...
Wait(Hsend, ...)
<safe to modify sbuf>
...

Process B

...
Irecv(rbuf, ... A, ... Hrecv)
<do nothing with rbuf>
...
<can do work here>
<can do other MPI calls>
...
Wait(Hrecv, ...)
<can use rbuf>
...

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:

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

[IN buf] initial address of send buffer (any data type)
[IN count] number of elements in send buffer (integer)
[IN datatype] datatype of each send buffer element (defined constant)
[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:

```
  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 (defined constant)
[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	call MPI_WAIT (request, status, ierror)
C:	ierror = MPI_Wait (request, status)

Test: MPI_TEST

Returns immediately with information about a non-blocking send or receive. Gives immediate answer to: **is send or receive done?**

MPI_TEST Syntax

Fortan:	call MPI_TEST (request, flag, status, ierror)
C:	ierror = MPI_Test (request, flag, status)

Non-blocking Communication completion calls

A request object can be deallocated at any time
Use the following operation:

MPI_REQUEST_FREE(request)
[INOUT request] communication request (handle)

C: ierror = MPI_Request_free(MPI_Request *request)

FORTRAN: call MPI_REQUEST_FREE(REQUEST, IERROR)

Non-blocking Communication Examples

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 some computation to mask latency****
        CALL MPI_WAIT(request, status, ierr)
ELSE
        CALL MPI_IRECV(a(1), 10, MPI_REAL, 0, tag, comm, Request, ierr)
        ****do some computation to mask latency****
        CALL MPI_WAIT(Request, status, ierr)
END IF
```

```
INCLUDE “mpif.h”
INTEGER ierror, rank, size, status(MPI_STATUS_SIZE), requests (2)

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)
   ! Can do computations which do not  overwrite a
   b = b + a
   CALL MPI_WAIT(requests(1), status, ierror)
   d = b + c
ELSE
   a = 14.2
   s = 18.4
   CALL MPI_IRECV(c,1,MPI_REAL,0,101,MPI_COMM_WORLD, requests(2), ierror)
   !  Do not read from or overwrite c till wait
   CALL MPI_WAIT(requests(2), status,ierror)
   c = a + c
END IF

CALL MPI_FINALIZE(ierror)
STOP
END
```

Non-blocking Communication

Gain

- Avoid Deadlocks
- Decrease Synchronization Overhead
- Can Reduce System Overhead
- Post non-blocking sends/receives early and do waits late
- Recommended: do MPI_IRECV before the MPI_Rsend is called.

Be careful with reads and writes

- Avoid writing to send buffer between MPI_ISEND and MPI_WAIT
- Avoid reading or 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

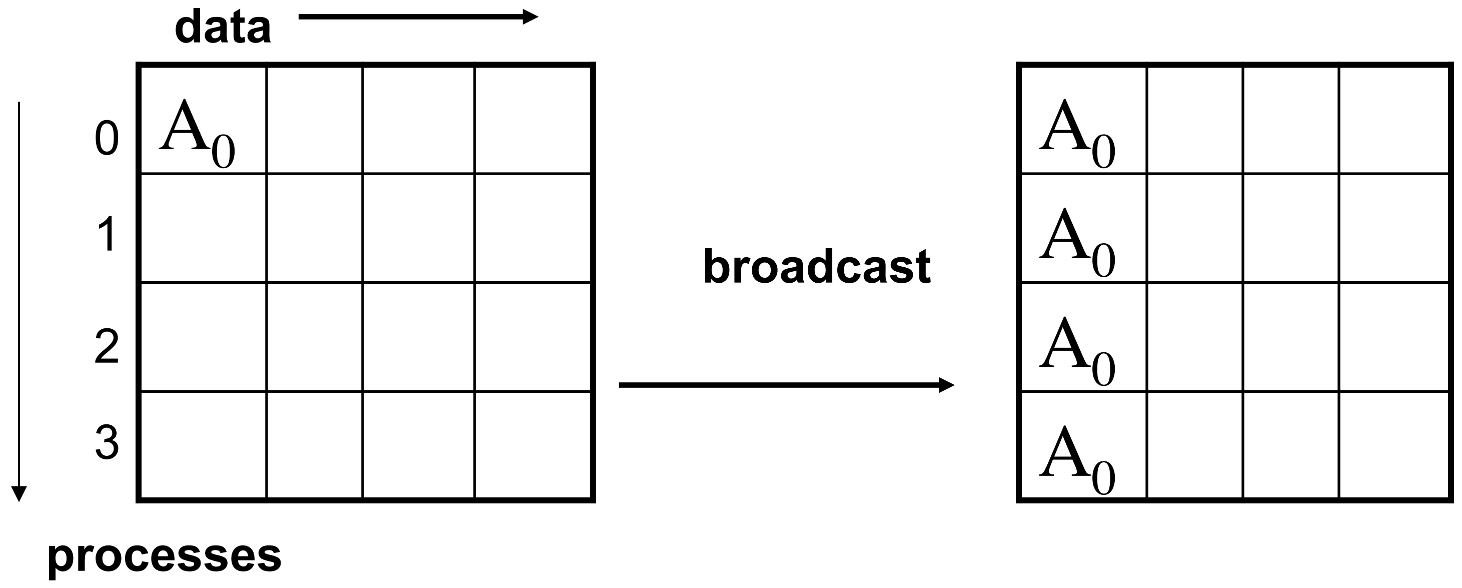
Collective computations (reductions):
Data from all members in a group is “reduced” to produce a global result (min, max, sum, ...).

Synchronization:
processes 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 rank sends (broadcasts) a block of data to all the ranks 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

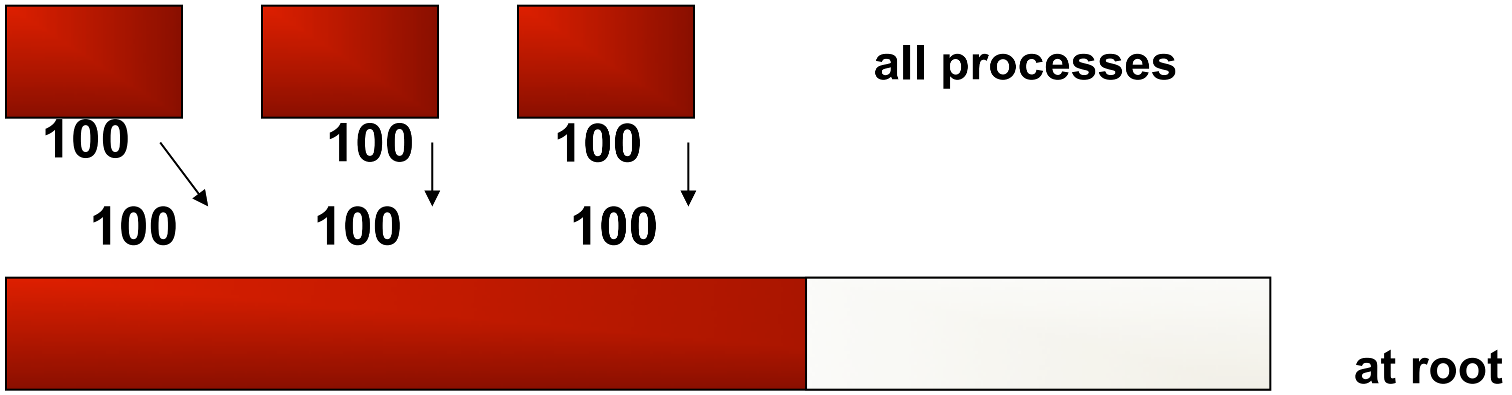
root: is an integer indicating the rank of broadcast root process

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.

MPI Collective Communication
Gather

Data is distributed throughout all processors in the group.
Collect distributed data to a specified process (rank).



```
real a(100), rbuf(MAX)
call mpi_gather(a, 100, MPI_REAL, rbuf, 100, MPI_REAL, root, comm, ierr)
```

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)
```

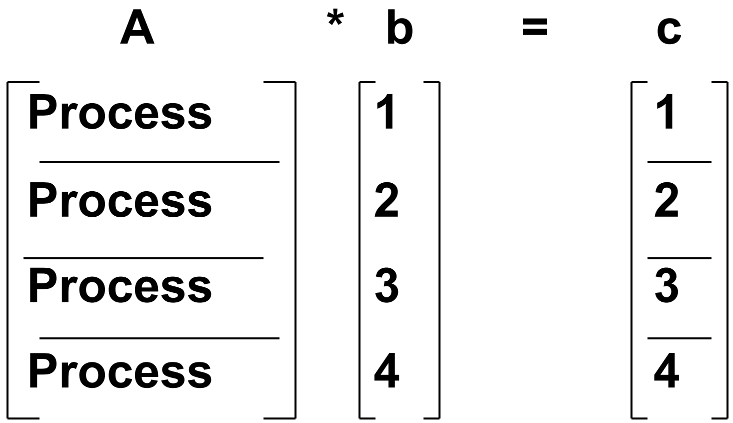
FORTRAN:
```
MPI_GATHER (sbuf, scount, stype, rbuf, rcount, rtype, root, comm, ierr)
```
where:

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

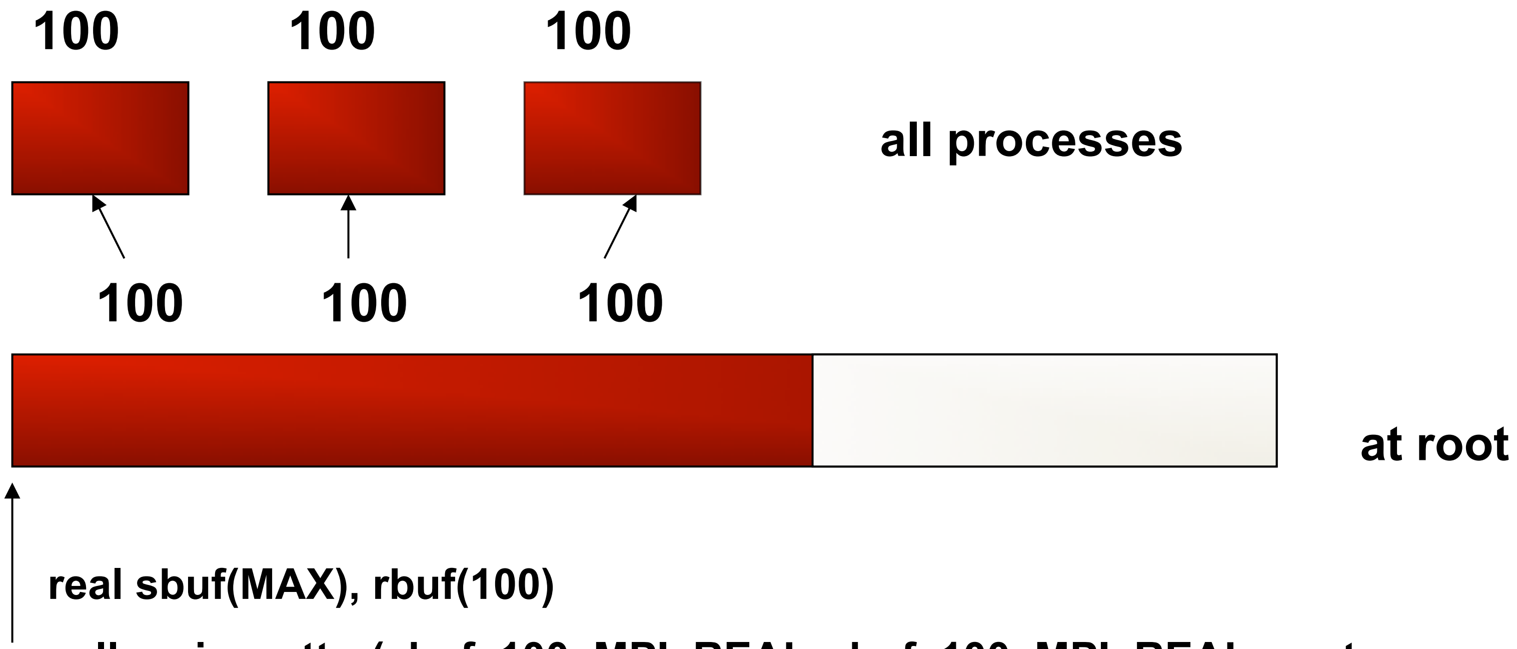
```
INCLUDE ‘mpif.h’
DIMENSION A(25, 100), b(100), cpart(25), ctotal(100)
INTEGER root, rank
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, np, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
root=1
Call set_a(A,rank)
Call set_b(B)

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, ctotal, 25,
&            MPI_REAL, root, MPI_COMM_WORLD, ierr)
If(rank.eq.root) print*, (ctotal(I),I=1,100)
CALL MPI_FINALIZE(ierr)
END
```



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

MPI Collective Communication
Scatter

Distribute data of to all the processes (ranks) in a group.



```
real sbuf(MAX), rbuf(100)
call mpi_scatter(sbuf, 100, MPI_REAL, rbuf, 100, MPI_REAL, root, comm, ierr)
```

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)
```

FORTRAN:

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

where:

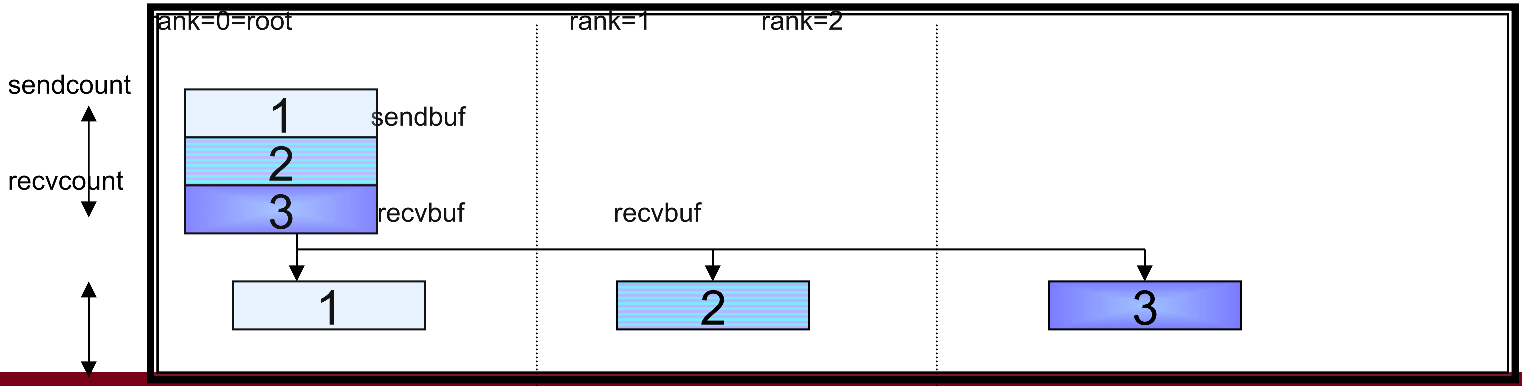
sbuf:	is the address of the send buffer,
scount:	is the number of elements sent to each process,
stype:	is the data type of the send buffer elements,
rbuf:	is the address of the receive buffer,
rcount:	is the number of elements in the receive buffer,
rtype:	is the data type of the receive buffer elements,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

Note: sbuf is significant for root process only

Sample execution

```
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
```

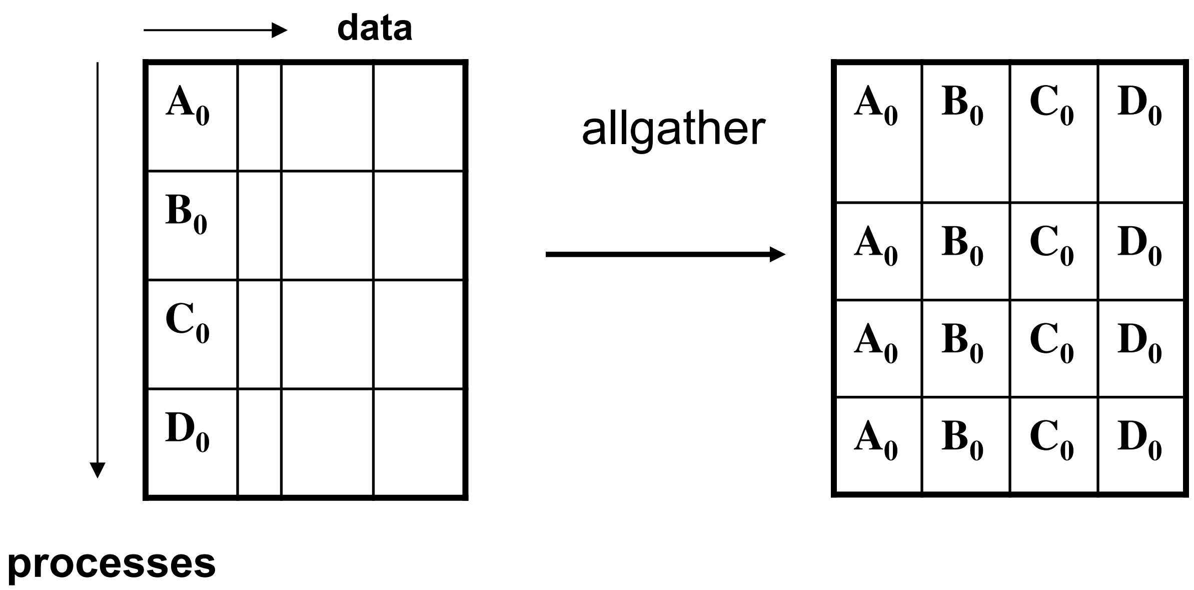
```
$ mpirun -np 3 ./a.out
0: irecv = 1
1: irecv = 2
2: irecv = 3
```



MPI Collective Communication

All Gather

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



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 scount, MPI_Datatype stype, void* rbuf, int rcount,
                   MPI_Datatype rtype, MPI_Comm comm)
```

FORTRAN

```
MPI_ALLGATHER (sbuf, scount, 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?

Replace

```
CALL MPI_GATHER (cpart, 25, MPI_REAL, ctotal, 25,
                 MPI_REAL, root, MPI_COMM_WORLD, ierr)
```

With

```
CALL MPI_ALLGATHER (cpart, 25, MPI_REAL, ctotal, 25,
                    MPI_REAL, MPI_COMM_WORLD, ierr)
```

MPI Collective Communication

Alltoall

Send Buffer

Data $\longrightarrow$

Processes

A_0	B_0	C_0	D_0	E_0	F_0
A_1	B_1	C_1	D_1	E_1	F_1
A_2	B_2	C_2	D_2	E_2	F_3
A_3	B_3	C_3	D_3	E_3	F_3
A_4	B_4	C_4	D_4	E_4	F_4
A_5	B_5	C_5	D_5	E_5	F_5

Receive Buffer

Data $\longrightarrow$

Processes

A_0	A_1	A_2	A_3	A_4	A_5
B_0	B_1	B_2	B_3	B_4	B_5
C_0	C_1	C_2	C_3	C_4	C_5
D_0	D_1	D_2	D_3	D_4	D_5
E_0	E_1	E_2	E_3	E_4	E_5
F_0	F_1	F_2	F_3	F_4	F_5

MPI Collective Communication
Alltoall Syntax

C:

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

FORTRAN:

```
MPI_ALLTOALL (sbuf, scount, 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.

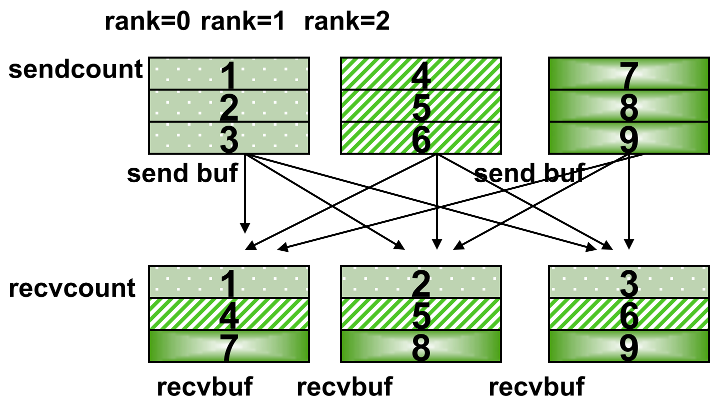


Figure of MPI_ALLTOALL

```
$ 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

https://www.msi.umn.edu/content/mpi-hands-workshop

# Get example and build

```
cp -r /home/tech/public/examples/hello_mpi .
cd hello_mpi
module load intel  impi
make
```

# Run interactively

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

# Set the following for large-memory jobs

```
ulimit -s unlimited
```

MPI
Collective Computations and Synchronization

MPI_Reduce

These 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.

Fortran

MPI_REDUCE (sbuf, rbuf, count, stype, op, root, comm, ierr)

MPI_ALLREDUCE (sbuf, rbuf, count, stype, op, comm, ierr)

MPI_ REDUCE_SCATTER (sbuf, rbuf, rcounts, stype, op, comm, ierr)

where

sbuf:	is the address of send buffer,
rbuf:	is the address of receive buffer,
count:	is the number of elements in send buffer,
stype:	is the data type of elements of send buffer,
op:	is the reduce operation (which may be MPI predefined, or your own),
root:	is the rank of the root processes, and
comm:	is the group communicator.

C:

```
int MPI_Reduce (void* sbuf, void* rbuf, int count, MPI_Datatype stype,
 MPI_Op op, int root, MPI Comm comm)

int MPI_Allreduce(void* sbuf, void* rbuf, int count, MPI Datatype stype, MPI_Op op,
                 MPI Comm comm)

int MPI_Reduce_scatter (void* sbuf, void* rbuf, int* rcounts, MPI Datatype stype,
                        MPI_Op op, MPI Comm comm)
```

where

sbuf:	is the address of send buffer,
rbuf:	is the address of receive buffer,
count:	is the number of elements in send buffer,
stype:	is the data type of elements of send buffer,
op:	is the reduce operation (which may be MPI predefined, or your own),
root:	is the rank of the root processes, and
comm:	is the group communicator.

MPI Predefined Reduce Operations

	Name Meaning	C type	FORTRAN 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

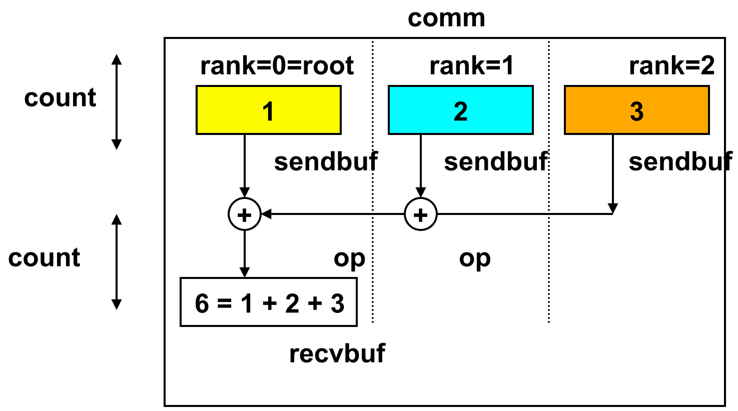


Figure: MPI_REDUCE for Scalar Variables

Sample Program

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

Sample execution

```
% a.out -procs 3
% 0: irecv = 6
```

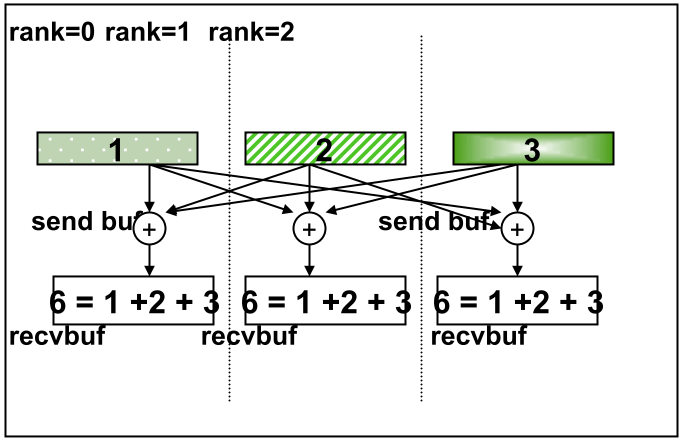
MPI_ALLREDUCE

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

comm

count

count



Sample program

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

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 execution

```
$ a.out -procs 3

0: irecv = 6
1: irecv = 6
2: irecv = 6
```

•Supercomputing Institute
•for Advanced Computational Research

UNIVERSITY OF MINNESOTA
Driven to Discover℠

MPI_REDUCE_SCATTER

Usage: CALL MPI_REDUCE_SCATTER(sendbuf, recvbuf, recvcounts, 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 **recvcounts**(*)	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 = Σ, recvcounts(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 recvcounts(i) elements. the ith segment is sent to process I and stored in the receive buffer defined by recvbuf, recvcounts(i) and datatype. MPI_REDUCE_SCATTER is functionally equivalent to MPI_REDUCE with count equal to the sum of recvcounts(i) followed by MPI_SCATTERV with sendcounts equal to recvcounts. All processes in comm need to call this routine.

CALL MPI_REDUCE_SCATTER (sendbuf, recvbuf, recvcounts, datatype, op, comm, ierror)

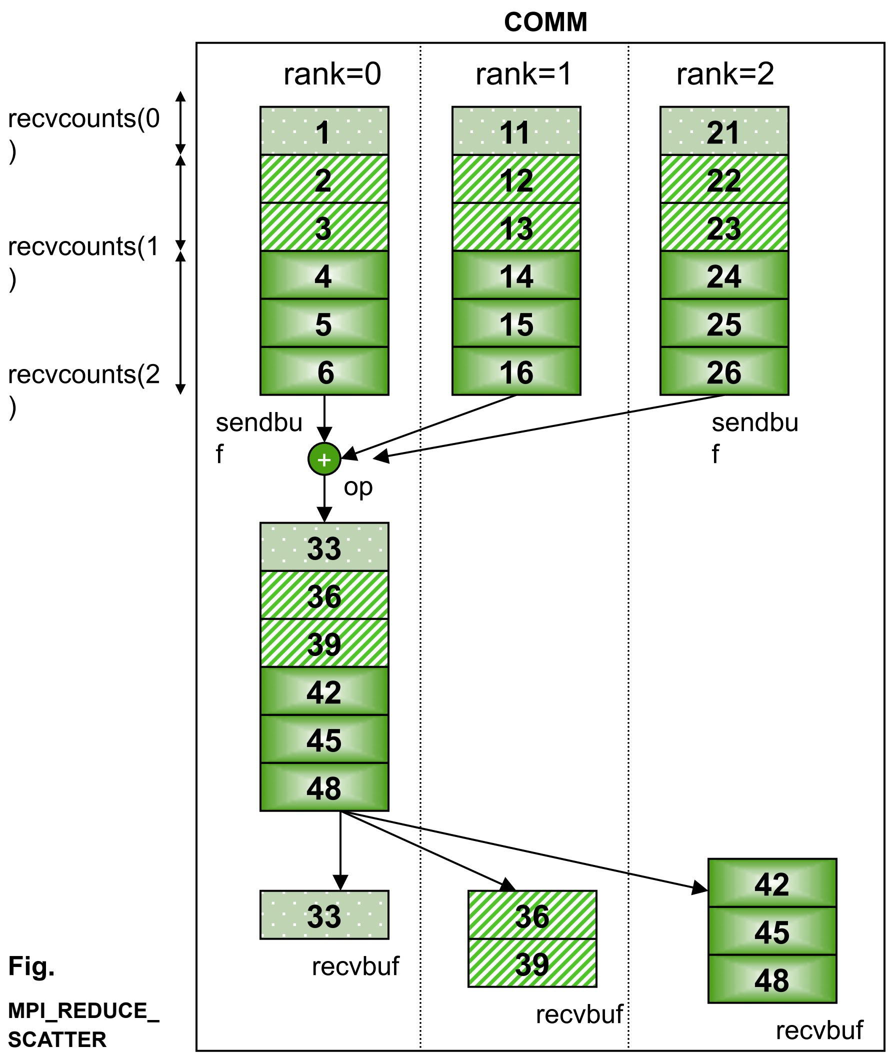


Fig. MPI_REDUCE_SCATTER

Sample Program

```
PROGRAM reduce_scatter
INCLUDE 'mpif.h'
INTEGER isend (6), irecv (3)
INTEGER ircnt (0:2)
DATA ircnt/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
  isend (i) = i + myrank * 10
ENDDO
CALL MPI_REDUCE_SCATTER(isend, irecv, ircnt, MPI_INTEGER,
&                                   MPI_SUM, MPI_COMM_WORLD,
ierr)
 PRINT *, 'irecv ='. irecv
 CALL MPI_FINALIZE(ierr)
 END
```

```
$ a.out -procs 3
  0: irecv = 33   0    0
  1: irecv = 36   39   0
  2: irecv = 42   45   48
```

•Supercomputing Institute
•for Advanced Computational Research

UNIVERSITY OF MINNESOTA
Driven to Discover℠

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

FORTRAN: 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.

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

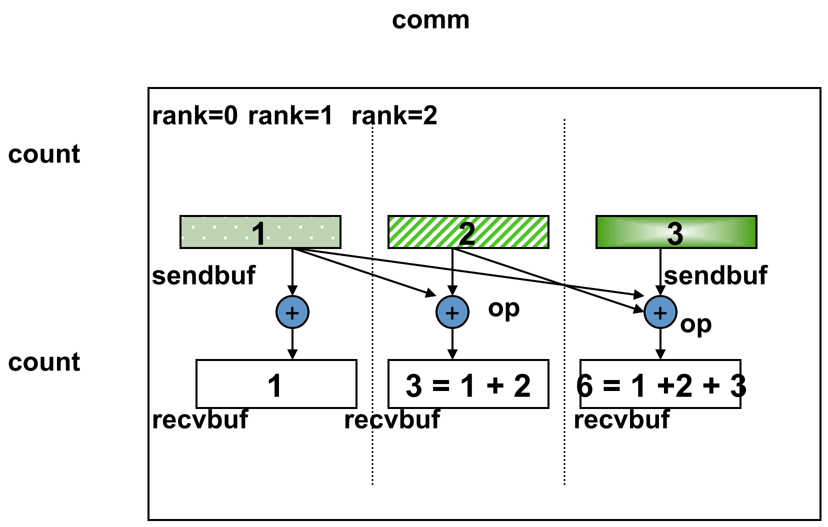


Figure. MPI_SCAN

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’
        CALL MPI_INIT (ierr)
        CALL MPI_COMM_SIZE (MPI_COMM_WORLD, nprocs, ierr)
        CALL MPI_COMM_RANK (MPI_COMM_WORLD, myrank, ierr)
        isend = myrank + 1
        CALL MPI_SCAN (isend, irecv, 1, MPI_INTEGER,
  &                  MPI_SUM, MPI_COMM_WORLD, ierr)
        PRINT *, ‘irecv =‘ irecv
        CALL MPI_FINALIZE(ierr)
        END
```

Sample execution

```
  $ a.out -procs 3
        0: irecv = 1
        0: irecv = 3
        0: irecv = 6
```

User-defined Operations

User can define his/her own reduce operation
Makes use of the MPI_OP_CREATE function

Performance Issues

A great deal of hidden communication takes place with collective communication. Performance depends greatly on the particular implementation of MPI. Because there may be forced synchronization, not always best to use collective communication.

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 comm)

- FORTRAN

  MPI_BARRIER (comm, ierr)

where:

MPI_Comm:	is an MPI predefined stucture of communicators,
comm:	is an integer denoting a communicator
ierr:	is an integer return error code.

Hands-on

https://www.msi.umn.edu/content/mpi-hands-workshop

# Get example and build

```
cp -r /home/tech/public/examples/hello_mpi .
cd hello_mpi
module load intel    impi
make
```

# Run interactively

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

# Set the following for large-memory jobs

```
        ulimit -s unlimited
```

THANK YOU

More info at www.msi.umn.edu

612-626-0802