

High Performance Computing

Lecture 05: Message Passing programming
Using MPI

Part B: 12-10-2023



“” **High Performance Computing**

2023-2024 Stefano Cozzini

Agenda

- Point-to-Points operation
- Collective operations

MPI routines Type signatures

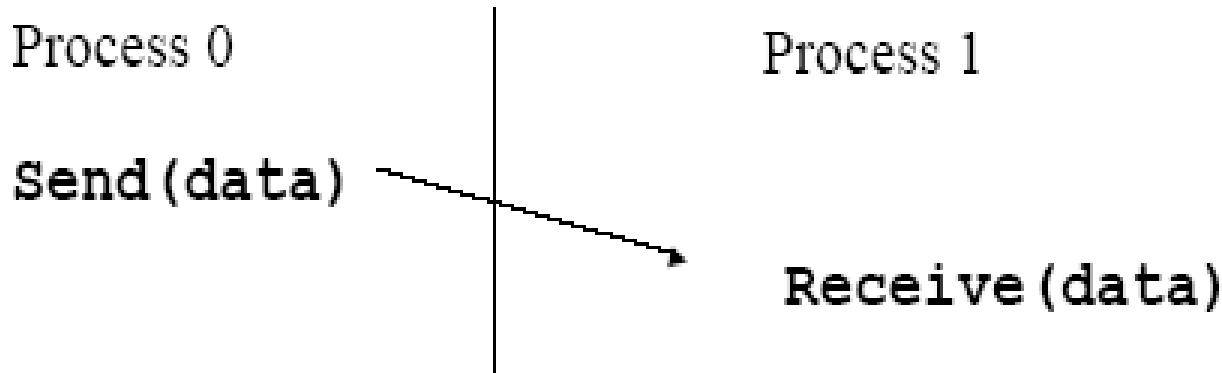
- All MPI routines have some similarities in their naming and in the parameters that they require. However, there are differences between C and Fortran, so let's look at these separately.
- C bindings
 - For C, the general type signature is
`rc = MPI_Xxxxx(parameter, ...)`
 - Note that case is important here, as it is with all C code. For example, MPI must be capitalized, as must the first character after the underscore. Everything after that must be lower case. All C MPI functions return an integer return code, which can be used to determine if the call succeeded.
 - If `rc == MPI_SUCCESS`, then the call was successful.
- C programs should include the file "mpi.h". This contains definitions for MPI functions and constants like `MPI_SUCCESS`.
- Fortran bindings
 - For Fortran, the general type signature is
`Call MPI_XXXXX(parameter,..., ierror)`
 - Note that case is not important here. So, an equivalent form would be
`call mpi_xxxxx(parameter,..., ierror)`
- Instead of having a function that returns an error code, as in C, the Fortran versions of MPI calls are subroutines that usually have one additional parameter in the argument list, `ierror`, which is the return code. Upon success, `ierror` is set to `MPI_SUCCESS`.

Basic elements of a message

- To send a message via mail we typically have:
 - An envelope (with possibly some hints on the content itself... i.e., advertisement, bills, greetings....)
 - A message
 - A destination address
 - A sender address
 - A tools to send the message (phone/mailer/ messenger)

For MPI it is exactly the same thing...

Basic Send/Receive



- How will “data” be described? datatypes
- How will processes be identified? rank/comm
- How will the receiver recognize messages? tag
- What will it mean for these operations to complete?
blocking/non-blocking ()

Describing data

- The data in a message to send or receive is described by a triple (address, count, datatype), where an MPI datatype is recursively defined as:
 - predefined, corresponding to a data type from the language (e.g., MPI_INT, MPI_DOUBLE)
 - a contiguous array of MPI datatypes
 - a strided block of datatypes
 - an indexed array of blocks of datatypes
 - an arbitrary structure of datatypes
- There are MPI functions to construct custom datatypes, in particular ones for subarrays

MPI data type: Fortran language

MPI Data type	Fortran Data type
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_DOUBLE_COMPLEX	DOUBLE COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_PACKED	
MPI_BYTE	

MPI data type: C language

MPI Data type	C Data type
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	Signed log 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 data tag



- Messages are sent with an accompanying user-defined integer tag, to assist the receiving process in identifying the message
- Messages can be screened at the receiving end by specifying a specific tag, or not screened by specifying `MPI_ANY_TAG` as the tag in a receive

Our first send message...

The simplest call:

```
MPI_send(buffer, count, data_type,  
destination, tag, communicator)
```

where:

BUFFER: data to send

COUNT: number of elements in buffer .

DATA_TYPE : which kind of data types in buffer ?

DESTINATION the receiver

TAG: the label of the message

COMMUNICATOR set of processors involved

And our first receiver..

The simplest call:

```
MPI_recv( buffer, count, data_type, source, tag,  
communicator, status)
```

Similar to send with the following differences:

- **SOURCE** is the sender ; can be set as **MPI_any_source** (receive a message from any processor within the communicator)
- **TAG** the label of message: can be set as **MPI_any_tag**: receive any kind of message
- **STATUS** integer array with information on message in case of error

The status array

Status is a data structure allocated in the user's program.

In C:

```
int recvd_tag, recvd_from, recvd_count;
MPI_Status status;
MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, ..., &status )
recvd_tag = status.MPI_TAG;
recvd_from = status.MPI_SOURCE;
MPI_Get_count( &status, datatype, &recvd_count );
```

In Fortran:

```
integer recvd_tag, recvd_from, recvd_count
integer status(MPI_STATUS_SIZE)
call MPI_RECV(..., MPI_ANY_SOURCE, MPI_ANY_TAG, .. status, ierr)
tag_recvd = status(MPI_TAG)
recvd_from = status(MPI_SOURCE)
call MPI_GET_COUNT(status, datatype, recvd_count, ierr)
```

A fortran example

```
Program MPI
  Implicit None
  !
  Include 'mpif.h'
  !
  Integer
  Integer :: rank :: buffer
  Integer, Dimension( 1:MPI_status_size ) :: status
  Integer :: error
  !
  Call MPI_init( error )
  Call MPI_comm_rank( MPI_comm_world, rank, error )
  !
  If( rank == 0 ) Then  buffer = 33
    Call MPI_send( buffer, 1, MPI_integer, 1, 10, &
      MPI_comm_world, error )
  End If
  !
  If( rank == 1 ) Then
    Call MPI_recv( buffer, 1, MPI_integer, 0, 10, &
      MPI_comm_world, status, error )  Print*, 'Rank
    ', rank, ' buffer=', buffer
    If( buffer /= 33 ) Print*, 'fail'  End If
  Call MPI_finalize( error )
End Program MPI
```

Questions for you:

- How many processors should I run this program on ?
- Can I run this program on 1000 processors ?

Blocking vs Non blocking calls

Q: When is a SEND instruction complete?

A: When it is safe to change the data that we sent.

Q: When is a RECEIVE instruction complete?

A: When it is safe to access the data we received.

Blocking vs Non blocking calls

With both communications (send and receive) we have two choices:

Start a communication and wait for it to complete:
BLOCKING approach

Start a communication and return control to the main program:
NON-BLOCKING approach

The Non-Blocking approach **REQUIRES** us to check for completion
before we can **modify/access** the **sent/received** data!!!

MPI_send/MPI_recv

MPI_SEND() and MPI_RECV()
are blocking operations.

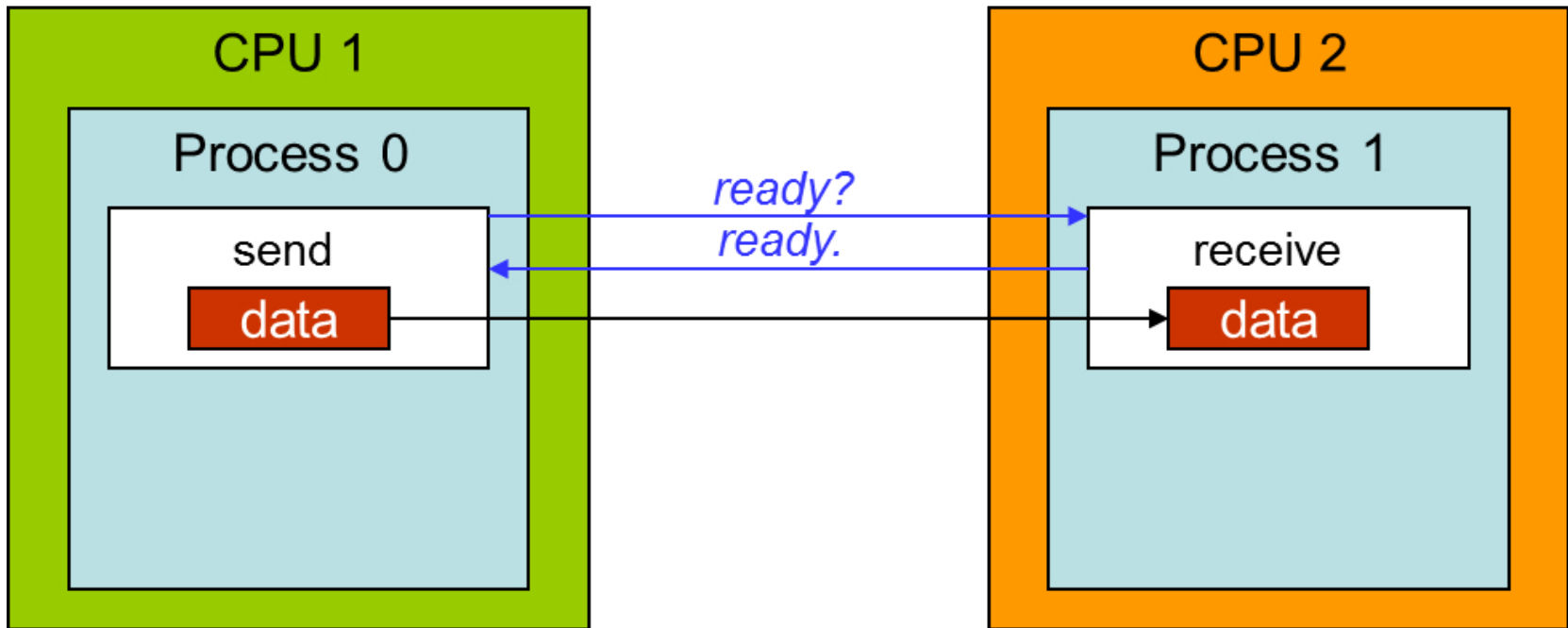
Communication mode and MPI routines

Mode	Completion Condition	Blocking subroutine	Non-blocking subroutine
Standard send	Message sent (receive state unknown)	<code>MPI_SEND</code>	<code>MPI_ISEND</code>
receive	Completes when a message has arrived	<code>MPI_RECV</code>	<code>MPI_Irecv</code>
Synchronous send	Only completes when the receive has completed	<code>MPI_SSEND</code>	<code>MPI_ISSEND</code>
Buffered send	Always completes, irrespective of receiver	<code>MPI_BSEND</code>	<code>MPI_IBSEND</code>
Ready send	Always completes, irrespective of whether the receive has completed	<code>MPI_RSEND</code>	<code>MPI_IRSEND</code>

Message overhead

- How much time your process will spend waiting for the blocking send (or receive) to return.
- Each mode has different overhead characteristics
- the overhead as having two sources:
 - System overhead: the time spent transferring buffer contents
 - Synchronization overhead' the time spent waiting for another process
- We can classify all of the time spent waiting as either system or synchronization overhead.

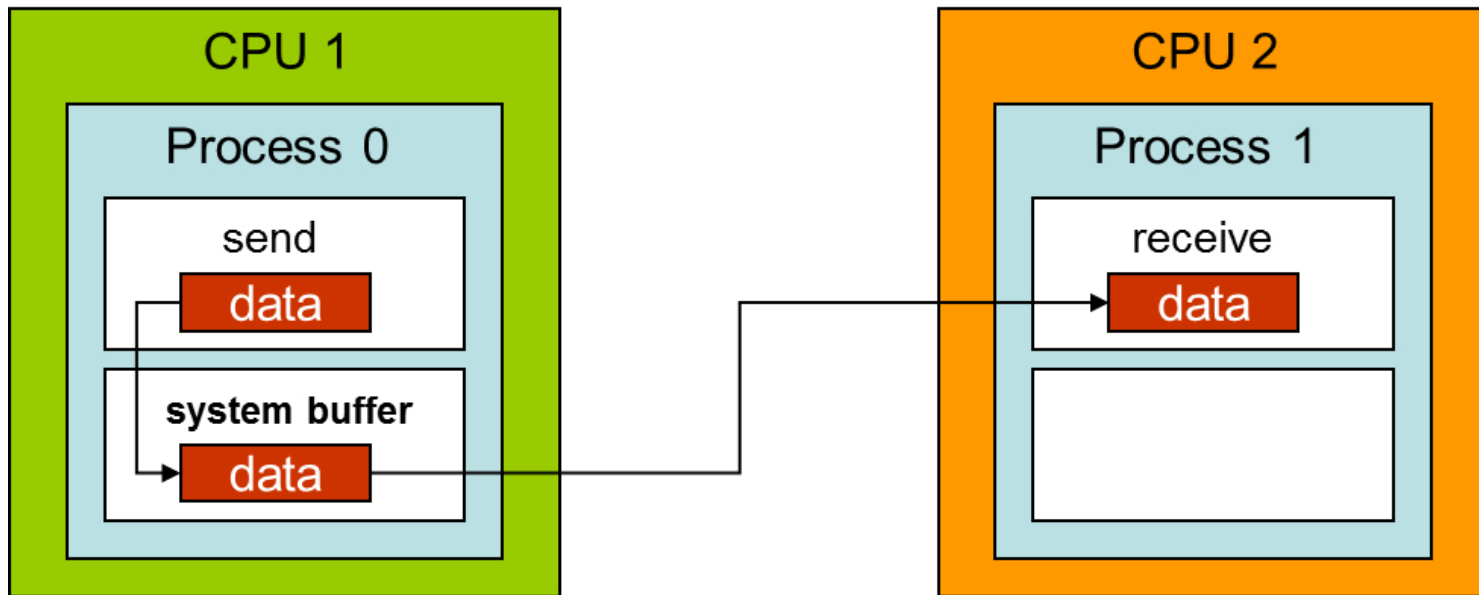
Synchronous send



the safest Point-To-Point communication method, as the sending process requires the receiving process to provide a "ready" signal, or the ***matching receive***, in order to initiate the send; therefore, the receiving process will always be ready to receive data from the sender.

Buffered Send

it copies the data from the message buffer to a user-supplied buffer, and then returns. The data will be copied from the user-supplied buffer over the network once the “ready to receive” notification has arrived.

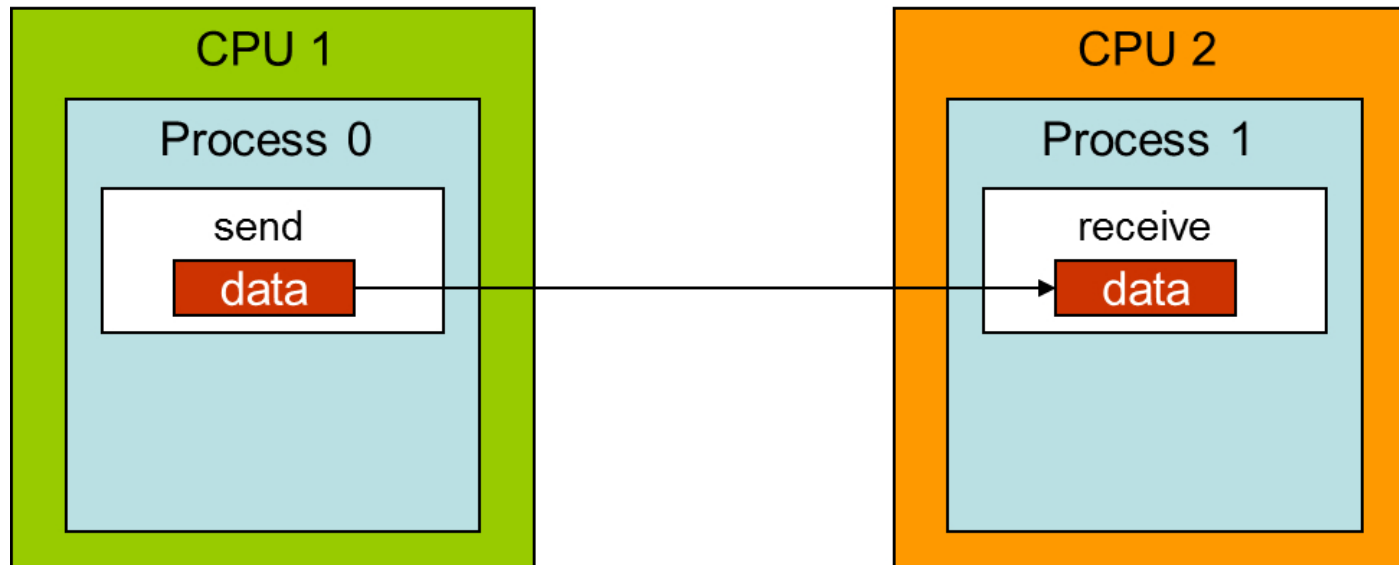


Implications:

- 1.timing of the corresponding receive is irrelevant
- 2.data in the original buffer can be modified
- 3.synchronization overhead on the sender is eliminated
- 4.system overhead is added

Ready Send

It attempts to reduce system and synchronization overhead by assuming that a ready-to-receive message has already arrived. Conceptually this results in a diagram that is identical to that first used to describe the simple view of point-to-point communication

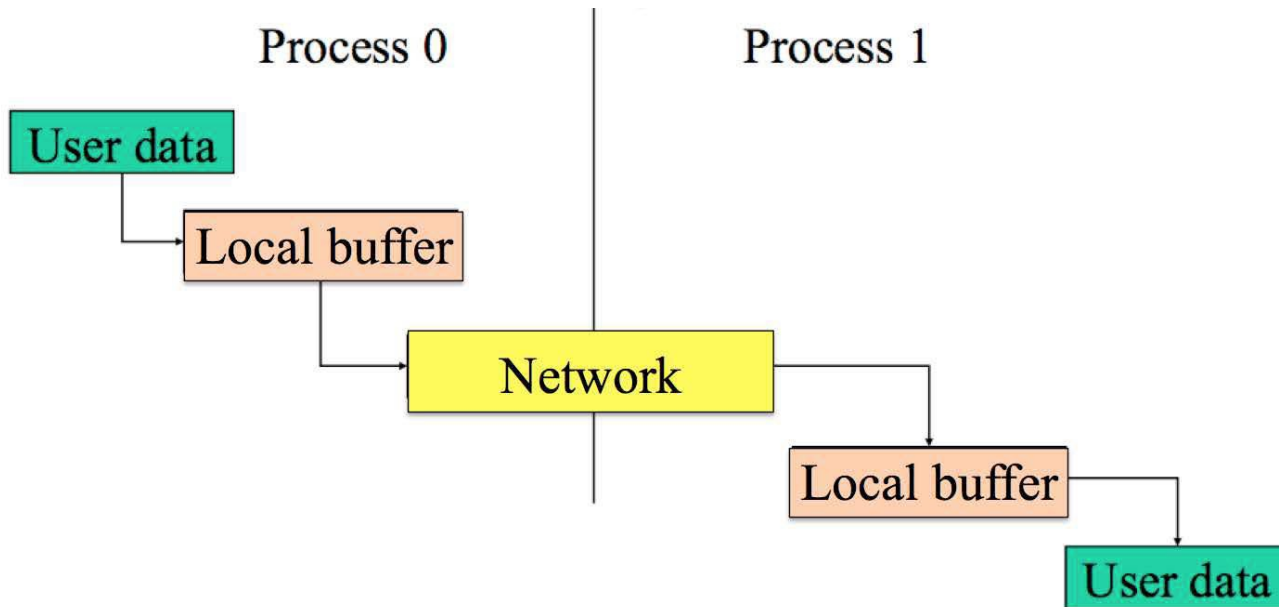


The idea is to have a blocking send that only blocks long enough to send the data to the network. However, if the matching receive has not already been posted when the send begins, an error will be generated.

Standard Send

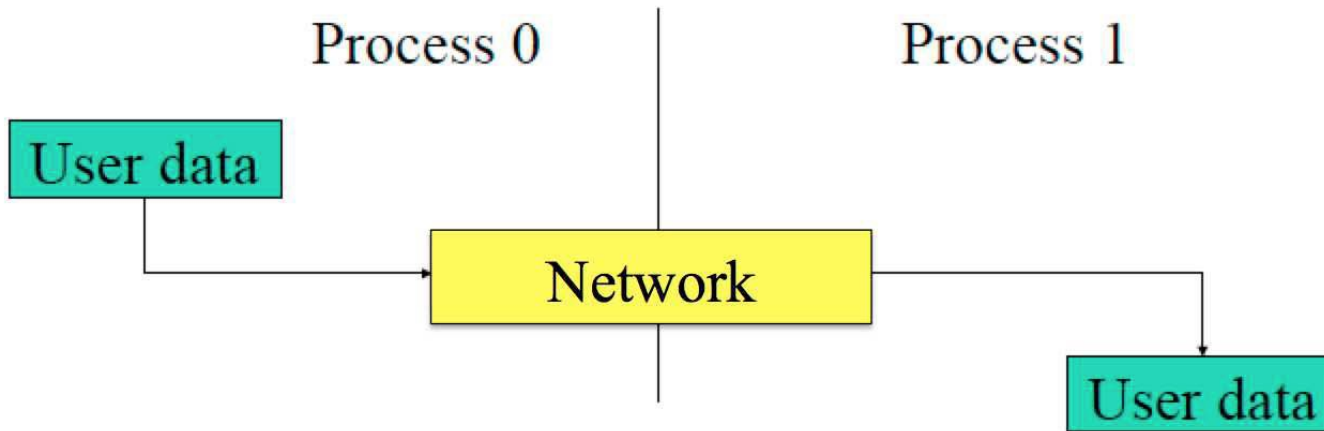
- the hardest to define.
- Its functionality is **left open to the implementer** in the MPI-1 and -2 specifications.
- Standard send, however, is intended to take advantage of specific optimizations that may be available via system enhancements.
- The strategy is to treat large and small messages differently.
- message buffering helps to reduce synchronization overhead, but it comes at a cost in memory.
- If messages are large, the penalty of putting extra copies in a buffer may be excessive and may even cause the program to crash
- Generally there is a threshold:
 - for intel MPI set by this env variable **I_MPI_EAGER_THRESHOLD**.

To make it clear:



Small messages make use
of system-supplied buffer

To make it clear:



Large message are
really blocking

Choosing a communication mode

Mode	Advantages	Disadvantages
Synchronous	<ul style="list-style-type: none">- Safest, therefore most portable- No need for extra buffer space- SEND/RECV order not critical	<ul style="list-style-type: none">- Can incur substantial synchronization overhead
Ready	<ul style="list-style-type: none">- Lowest total overhead- No need for extra buffer space- SEND/RECV handshake not required	<ul style="list-style-type: none">- RECV <i>must</i> precede SEND
Buffered	<ul style="list-style-type: none">- Decouples SEND from RECV- no sync overhead on SEND- Programmer can control size of buffer space- SEND/RECV order irrelevant	<ul style="list-style-type: none">- Copying to buffer incurs additional system overhead
Standard	<ul style="list-style-type: none">- Good for many cases- Compromise position	<ul style="list-style-type: none">- Protocol is determined by MPI implementation

Non blocking communication

- Nonblocking calls merely initiate the communication process.
- The status of the data transfer, and the success of the communication, must be verified at a later point in the program.
- The purpose of a nonblocking send is mostly to notify the system of the existence of an outgoing message: the actual transfer might take place later.
- It is up to the programmer to keep the send buffer intact until it can be verified that the message has actually been copied someplace else.
- Likewise, a nonblocking receive signals the system that a buffer is prepared for an incoming message, without waiting for the actual data to arrive.
- In either case, before trusting the contents of the message buffer, the programmer must check its status

Non blocking send

- Not really a “fire and forget” mechanism.
- Three reasons to check the status
 - You may want to modify/re-use the buffer
 - You may want to release resources involved.
 - You may want to check if the communication was successful

Non blocking call syntax

- Same syntax as the blocking ones, with two exceptions:
- The letter “I” (think of “initiate”) appears in the name of the call, immediately following the first underscore: e.g., MPI_Irecv.
- The final argument is a handle to an opaque (or hidden) request object that holds detailed information about the transaction. The request handle can be used for subsequent Wait and Test calls.

Non blocking send/receive (Fortran)

```
MPI_ISEND(buf, count, type, dest, tag, comm, req, ierr)
```

```
MPI_Irecv(buf, count, type, dest, tag, comm, req, ierr)
```

buf array of type **type** see table.

count (INTEGER) number of element of **buf** to be sent

type (INTEGER) MPI type of **buf**

dest (INTEGER) rank of the destination process

tag (INTEGER) number identifying the message

comm (INTEGER) communicator of the sender and receiver

req (INTEGER) output, identifier of the communications handle

ierr (INTEGER) output, error code (if **ierr=0** no error occurs)

Non blocking send/receive (C)

```
int MPI_Isend(void *buf, int count, MPI_Datatype type, int  
dest, int tag, MPI_Comm comm, MPI_Request *req);
```

```
int MPI_Irecv (void *buf, int count, MPI_Datatype type,  
int dest, int tag, MPI_Comm comm, MPI_Request *req);
```

Request handle

- All nonblocking calls in MPI return a *request handle* in lieu of a status variable.
- The purpose of the handle is to let you check on the status of your message at a later time.
- In MPI, the status variable becomes defined only after your message data is ready.
- Analogy: buzzer at the fastfood
 - You submit your request at the counter and a buzzer is given to you..



Function for non blocking communication

- **MPI_Test** - You go to the counter every two minutes ask the guy if your buzzer is not working.
- **MPI_Wait** - You hang around at the table with your friends till the buzzer sounds
- **MPI_Request_free** - You give back the buzzer without learning the status of your food. (Perhaps you've canceled your order; or perhaps the order is obviously ready and you already know what's in it, so you can just grab it without further checking.)

Waiting for completion...

Fortran:

MPI_WAIT(req, status, ierr)

- A call to this subroutine cause the code to wait until the communication pointed by req is complete.
- **Req** (INTEGER) input/output, communication handler (initiated by **MPI_ISEND** or **MPI_IRECV**).
- **Status** (INTEGER) array of size **MPI_STATUS_SIZE**,
- if **Req** was associated to a call to **MPI_IRECV**, **status** contains informations on the received message, otherwise **status** could contain an error code.
- **ierr** (INTEGER) output, error code (if **ierr=0** no error occurs).

C:

```
int MPI_Wait(MPI_Request *req, MPI_Status  
*status);
```

Testing for completion

Fortran:

MPI_TEST(req, flag, status, ierr)

- A call to this subroutine sets flag to .true. if the communication pointed by req is complete, sets flag to .false. otherwise.
- **Flag**(LOGICAL) output, .true. if communication req has completed .false. otherwise
- **Req** (INTEGER) input/output, communication handler (initiated by **MPI_ISEND** or **MPI_IRECV**).
- **Status** (INTEGER) array of size **MPI_STATUS_SIZE**,
- **ierr** (INTEGER) output, error code (if **ierr=0** no error occurs).

C:

```
int MPI_Wait(MPI_Request *req, int  
*flag, MPI_Status *status);
```

Pros and Cons of Non-Blocking Send/ Receive

- Non-Blocking communications allows the separation between the initiation of the communication and the completion.
- Advantages:
 - between the initiation and completion the program could do other useful computation (latency hiding).
- Disadvantages:
 - the programmer has to insert code to check for completion.

Overview of MPI send modes

- Recommendation:
 - The best performance is likely if you can write your program so that you could use just MPI_Ssend; in that case, an MPI implementation can completely avoid buffering data.
 - Use MPI_Send instead; this allows the MPI implementation the maximum flexibility in choosing how to deliver your data
 - If nonblocking routines are necessary, then try to use MPI_Isend or MPI_Irecv. Use MPI_Bsend only when it is too inconvenient to use MPI_Isend
 - The remaining routines, MPI_Rsend, MPI_Ssend, etc., are rarely used but may be of value in writing system-dependent message-passing code entirely within MPI.

Deadlock

- Often encountered in parallel processing..
- In communications, a typical scenario involves two processes wishing to exchange messages: each is trying to give a message to the other, but neither of them is ready to accept a message.
- the deadlock phenomenon is most common with blocking communication.
- It is relatively easy to get into a situation where multiple tasks are waiting for events that haven't been initiated yet—and never can be.

MPI: a case study

Problem: exchanging data between two processes

```
If( rank == 0 ) then
    Call MPI_send( buffer1, 1, MPI_integer, 1, 10, &
                  MPI_comm_world, error )
    Call MPI_recv( buffer2, 1, MPI_integer, 1, 20, &
                  MPI_comm_world, status, error )
Else If( rank == 1 )then
    Call MPI_send( buffer2, 1, MPI_integer, 0, 20, &
                  MPI_comm_world, error )
    Call MPI_recv( buffer1, 1, MPI_integer, 0, 10, &
                  MPI_comm_world, status, error )
End If
```

DEADLOCK

Solution A

USE BUFFERED SEND: **bsend**
send and go back so the deadlock is avoided

```
If( rank == 0 ) then
    Call MPI_Bsend( buffer1, 1, MPI_integer, 1, 10, &
                    MPI_comm_world, error )
    Call MPI_recv( buffer2, 1, MPI_integer, 1, 20, &
                  MPI_comm_world, status, error )
Else If( rank == 1 )then
    Call MPI_Bsend( buffer2, 1, MPI_integer, 0, 20, &
                    MPI_comm_world, error )
    Call MPI_recv( buffer1, 1, MPI_integer, 0, 10, &
                  MPI_comm_world, status, error )
End If
```

Requires a copy therefore is not efficient
for large data set memory problems

Solution B

Use non blocking SEND : **isend**
send go back but now is not safe to change the buffer

```
If( rank == 0 ) then
    Call MPI_Isend( buffer1, 1, MPI_integer, 1, 10, &
                    MPI_comm_world, error )
    Call MPI_recv( buffer2, 1, MPI_integer, 1, 20, &
                  MPI_comm_world, status, error )
Else If( rank == 1 )then
    Call MPI_Isend( buffer2, 1, MPI_integer, 0, 20, &
                    MPI_comm_world, error )
    Call MPI_recv( buffer1, 1, MPI_integer, 0, 10, &
                  MPI_comm_world, status, error )

End If
Call MPI_wait( REQUEST, status ) ! Wait until send is complete
```

- 1 A **handle** is introduced to test the status of message.
2. More efficient of the previous solutions

Solution C

Just rearrange the order of call and all is done

```
If( rank == 0 ) then
    Call MPI_send( buffer1, 1, MPI_integer, 1, 10, &
                  MPI_comm_world, error )
    Call MPI_recv( buffer2, 1, MPI_integer, 1, 20, &
                  MPI_comm_world, status, error )
Else If( rank == 1 )then
    Call MPI_recv( buffer1, 1, MPI_integer, 0, 10, &
                  MPI_comm_world, status, error )
    Call MPI_send( buffer2, 1, MPI_integer, 0, 20, &
                  MPI_comm_world, error )
End If
```

the most efficient one and the recommended one

Strategies to avoid deadlock

1. Go with buffered mode

Use buffered sends so that computation can proceed after copying the message to the user-supplied buffer. This will allow the receives to be executed. This method resolves deadlock problems the same way that strategy 2 does, though with somewhat different performance characteristics.

2. Use nonblocking calls

Have each task post a nonblocking receive before it does any other communication. This allows each message to be received, no matter what the task is working on when the message arrives or in what order the sends are posted.

3. Arrange for a different ordering of calls between tasks

Have one task post its receive first and the other post its send first. That clearly establishes that the message in one direction will precede the other.

4. Try MPI's coupled Sendrecv methods

MPI_Sendrecv and MPI_Sendrecv_replace can be elegant solutions. In the _replace version, the system allocates some buffer space to deal with the exchange of messages.

Often, deadlock is simply a result of not carefully thinking out your parallelization scheme so solution 1 should always be your first stop.

SendRecv / SendRecv_replace

- functions that combine a blocking send and receive into a single API call.
- This is useful for “swap” communications:
 - messages must be exchanged between processes;
 - “chain” communications, where messages are being passed down the line.
- There are two versions of these combined calls which differ in the number of buffers.

Sendrecv

```
int MPI_Sendrecv ( \
    const void *sendbuf, int sendcount, \
    MPI_Datatype sendtype, int dest, int sendtag, \
    void *recvbuf, int recvcount, MPI_Datatype recvtype, \
    int source, int recvtag, \
    MPI_Comm comm, MPI_Status *status)
```

Parameters:

- *sendbuf:** The address of the send buffer.
- sendcount:** Number of elements in the send buffer.
- sendtype:** Data type of the send buffer.
- dest:** Destination process number (rank number).
- sendtag:** Sent message tag.
- *recvbuf:** The memory address of the receiving buffer.
- recvcount:** Number of elements in the receive buffer.
- recvtype:** Data type of the receiving buffer.
- source:** Source process rank number.
- recvtag:** Receiving message tag.
- comm:** MPI communicator.
- *status:** Status of object.

Collective Operations

- Collective routines provide a higher-level way to organize a parallel program
- Each process executes the same communication operations
- MPI provides a rich set of collective operations...

Collective Operations

- Communications involving group of processes in a communicator.
- Groups and communicators can be constructed “by hand” or using topology routines.
- No non-blocking collective operations.
- Three classes of operations:
 - synchronization,
 - data movement
 - collective computation

Collective communication characteristics

- Involve coordinated communication within a *group* of processes identified by an MPI communicator
- Substitute for a more complex sequence of point-to-point calls
- For blocking calls, must block until they have completed ***locally***
- *May, or may not,* use synchronized communications (implementation dependent)
- Specify a *root* process to originate or receive all data (in some cases)
- Must exactly match amounts of data specified by senders and receivers
- Do not use message tags

Three Types of routines

- Synchronization
 - Barrier synchronization
- Data Movement
 - Broadcast from one member to all other members
 - Gather data from an array spread across processes into one array
 - Scatter data from one member to all members
 - All-to-all exchange of data
- Global Computation
 - Global reduction (e.g., sum, min of distributed data elements)
 - Scan across all members of a communicator

MPI_barrier()

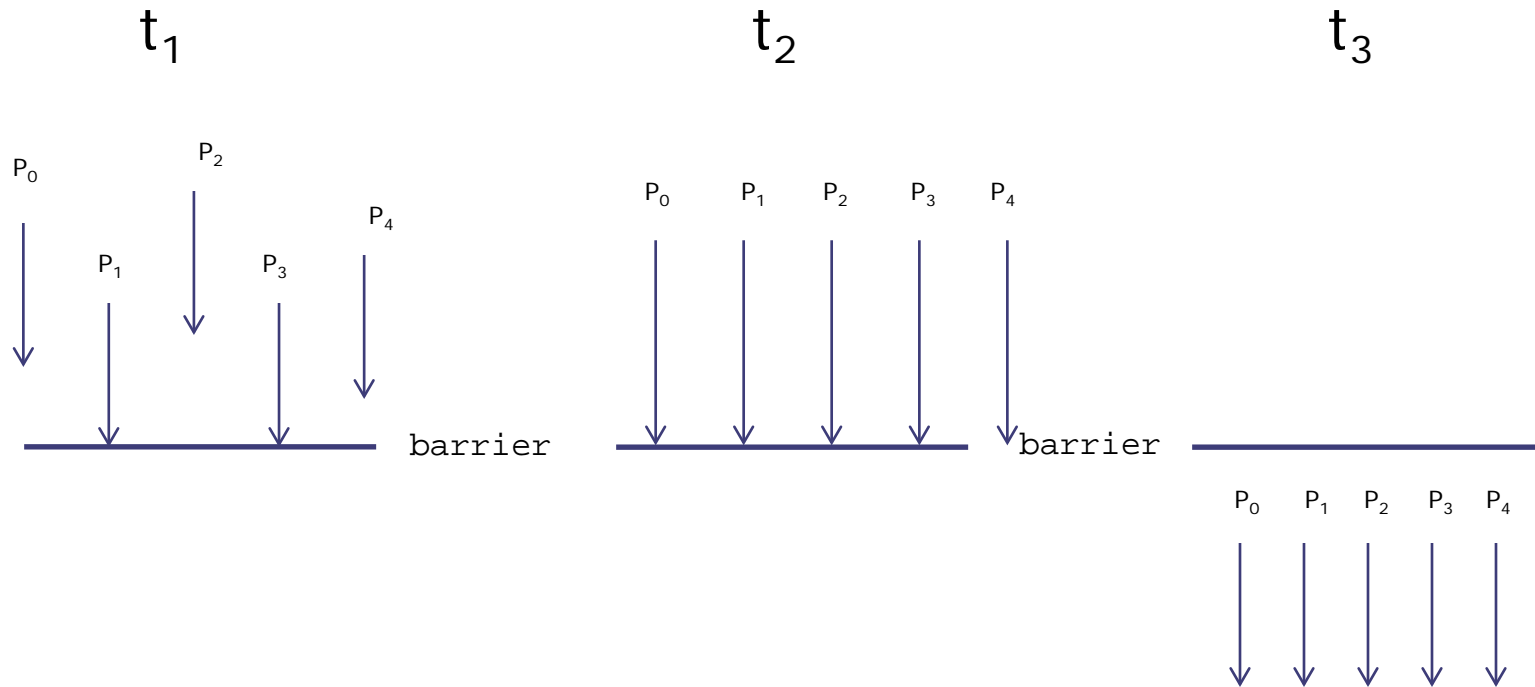
- Stop processes until all processes within a communicator reach the barrier
- Almost never required in a well-done parallel program
- Useful in measuring performance and load balancing and debugging
- Fortran:

```
CALL MPI_BARRIER(comm, ierr)
```

- C

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

MPI_barrier(): graphical view



Data movements

- one process either sends to or receives from all processes
 - **Broadcast**
 - **Gather**
 - **Scatter**
- All processors both send and receive:
 - **Allgather**
 - **Alltoall**
- Variable data versions:
 - **Gatherv/Scatterv**
 - **Allgatherv/alltoallv**

Broadcast (MPI_bcast)

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

```
INTEGER count, type, root, comm, ierr  
CALL MPI_BCAST(buf,count,type,root,comm,ierr)
```

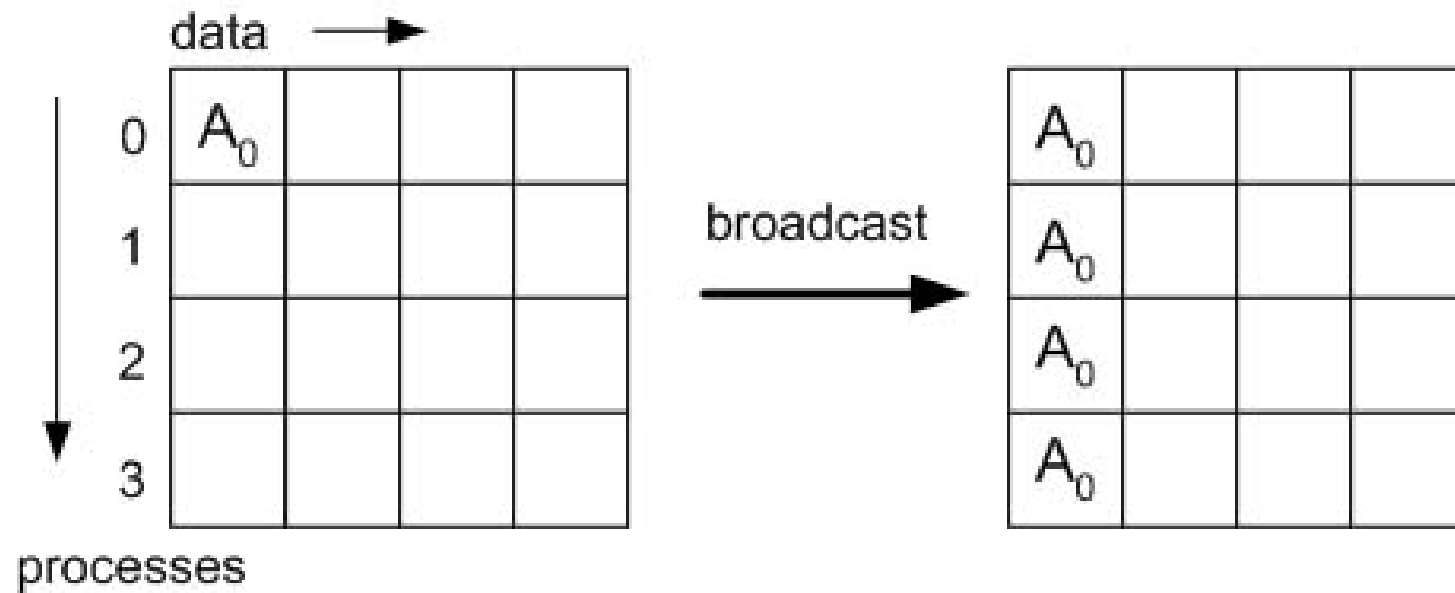
Buf array of type **type**

- C:

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

- All processes must specify same **root**, **rank** and **comm**

Graphical view..

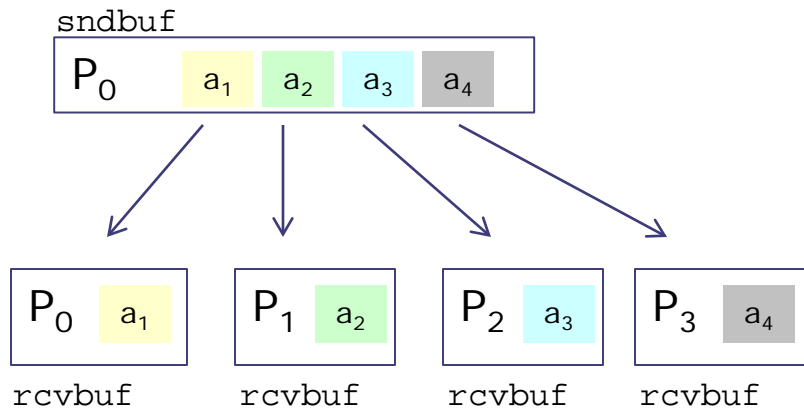


Gather/Scatter

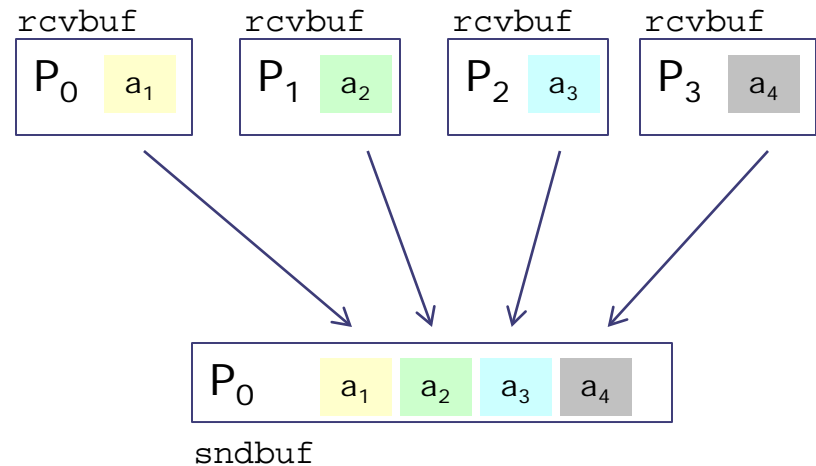
- Gather purpose
 - If an array is scattered across all processes in the group and one wants to collect each piece of the array into a specified array on a single process, the call to use is GATHER.
- Scatter purpose:
 - On the other hand, if one wants to distribute the data into n segments, where the i th segment is sent to the i th process in the group which has n processes, use SCATTER. Think of it as the inverse to GATHER.

Scatter and Gather operations

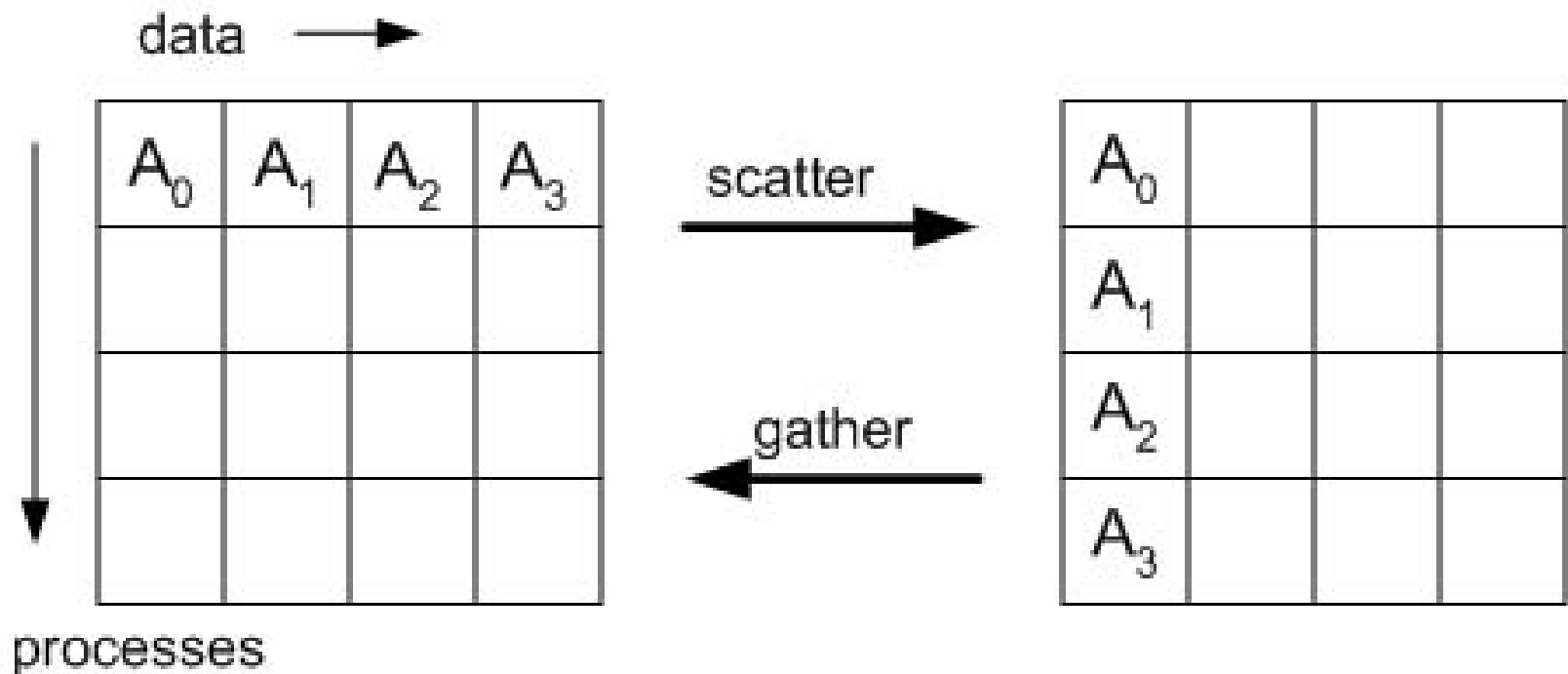
Scatter



Gather

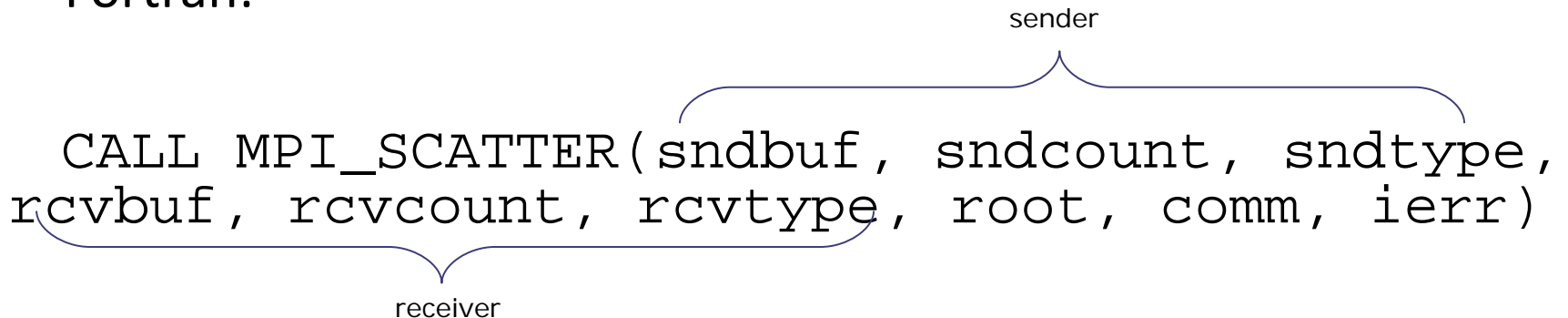


Gather/Scatter with matrix-style representation



MPI_scatter

- One-to-all communication: different data sent from root process to all others in the communicator
- Fortran:

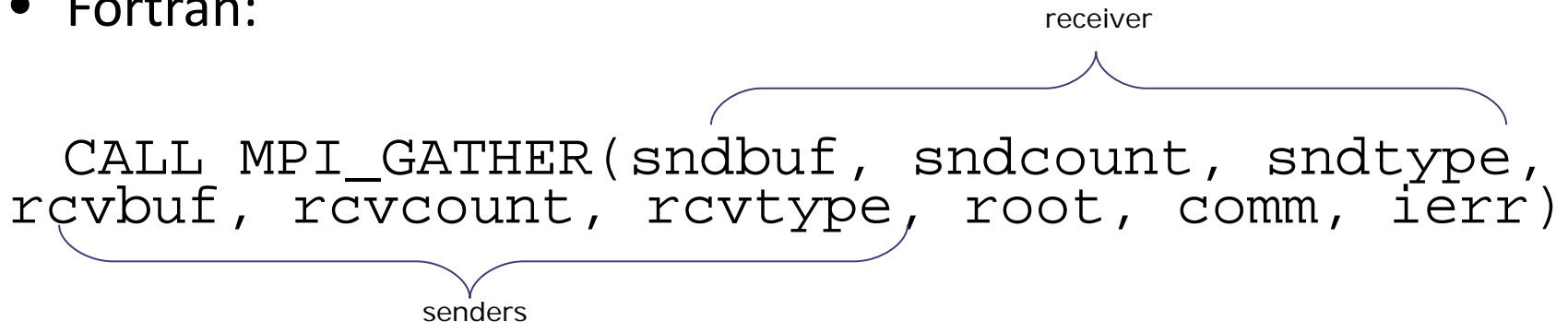
The diagram shows the Fortran subroutine call `CALL MPI_SCATTER(sndbuf, sndcount, sndtype, rcvbuf, rcvcount, rcvtype, root, comm, ierr)`. A bracket above the arguments `sndbuf`, `sndcount`, and `sndtype` is labeled "sender". A bracket below the arguments `rcvbuf`, `rcvcount`, and `rcvtype` is labeled "receiver".

```
CALL MPI_SCATTER(sndbuf, sndcount, sndtype,  
rcvbuf, rcvcount, rcvtype, root, comm, ierr)
```

- Arguments definition are like other MPI subroutine
- **sndcount** is the number of elements sent to each process, not the size of **sndbuf**, that should be **sndcount** times the number of process in the communicator
- The sender arguments are significant only at root

MPI_gather

- One-to-all communication: : different data collected by the root process, from all others processes in the communicator. Is the opposite of Scatter
- Fortran:

The diagram shows the Fortran subroutine call `CALL MPI_GATHER(sndbuf, sndcount, sndtype, rcvbuf, rcvcount, rcvtype, root, comm, ierr)`. A bracket above the last three arguments (`rcvbuf, rcvcount, rcvtype`) is labeled "receiver". A bracket below the first three arguments (`sndbuf, sndcount, sndtype`) is labeled "senders".

```
CALL MPI_GATHER(sndbuf, sndcount, sndtype,  
rcvbuf, rcvcount, rcvtype, root, comm, ierr)
```

- Arguments definition are like other MPI subroutine
- **rcvcount** is the number of elements collected from each process, not the size of **rcvbuf**, that should be **rcvcount** times the number of process in the communicator
- The receiver arguments are significant only at root

Example: Matrix-vector in parallel..

- Matrix is distributed by rows (i.e. row-major order)
- Product vector is needed in entirety by one process
- MPI_Gather will be used to collect the product from processes

$$\begin{array}{c} A \\ \left[\begin{array}{c} \text{Process 0} \\ \text{-----} \\ \text{Process 1} \\ \text{-----} \\ \text{Process 2} \\ \text{-----} \\ \text{Process 3} \end{array} \right] \end{array} * \begin{array}{c} b \\ \left[\begin{array}{c} \\ \\ \\ \end{array} \right] \end{array} = \begin{array}{c} c \\ \left[\begin{array}{c} 0 \\ \text{-----} \\ 1 \\ \text{-----} \\ 2 \\ \text{-----} \\ 3 \end{array} \right] \end{array}$$

C- Example: Matrix-vector in parallel..

```
float Apart[25,100], b[100], cpart[25], ctotat[100];
int root;
root=0;
:
/* Code that initializes Apart and b */
:
for(i=0; i<25; i++)
{
    cpart[i]=0;
    for(k=0; k<100; k++)
    {
        cpart[i] = cpart[i] + Apart[i,k] * b[k];
    }
}
MPI_Gather(cpart, 25, MPI_FLOAT, ctotat, 25,
MPI_FLOAT, root, MPI_COMM_WORLD);
```

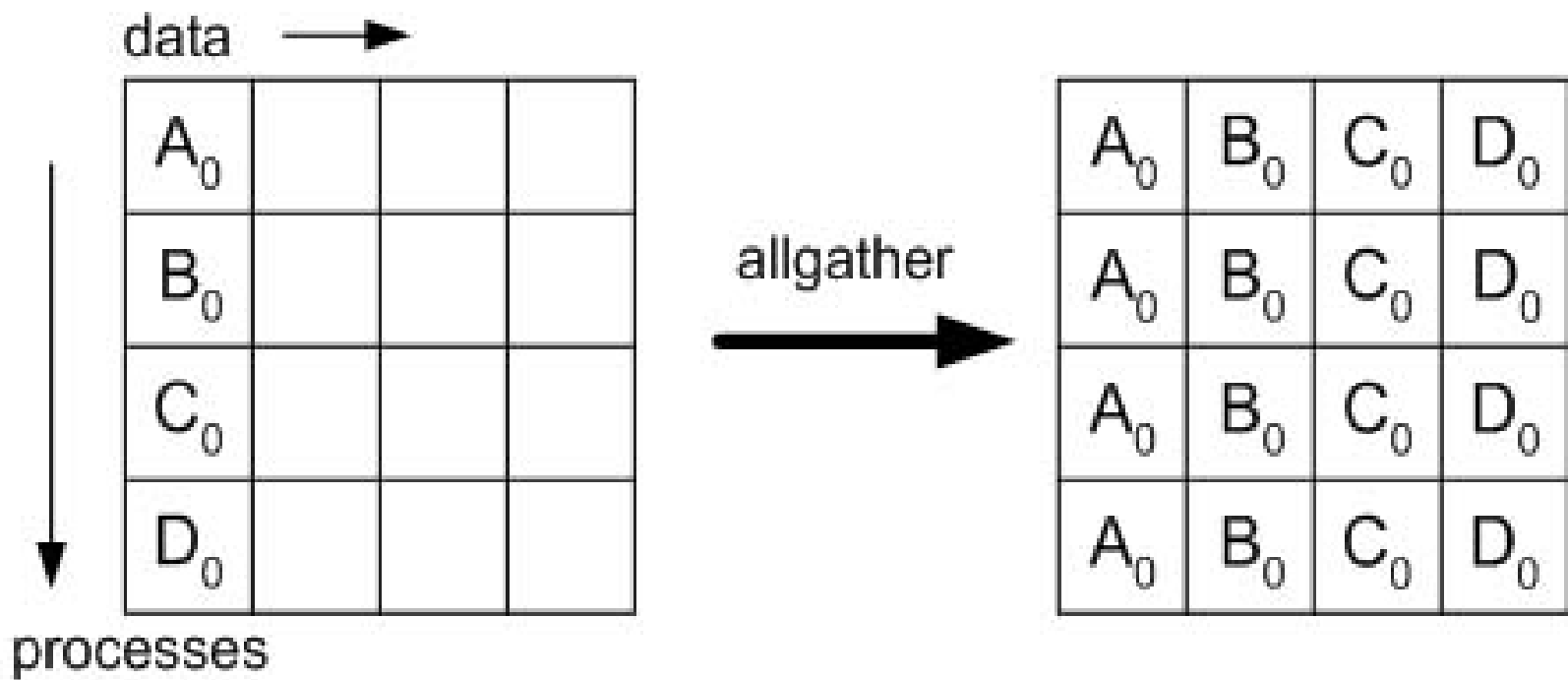
Gatherv and Scatterv

- `MPI_Gatherv` and `MPI_Scatterv` are the variable-message-size versions of `MPI_Gather` and `MPI_Scatter`.
- They permit a varying count of data from/to each process.
- Obtained by changing the `count` argument from a single integer to an integer array and providing a new argument `displs` (an array).

MPI_Allgather/ MPI_Allgatherv

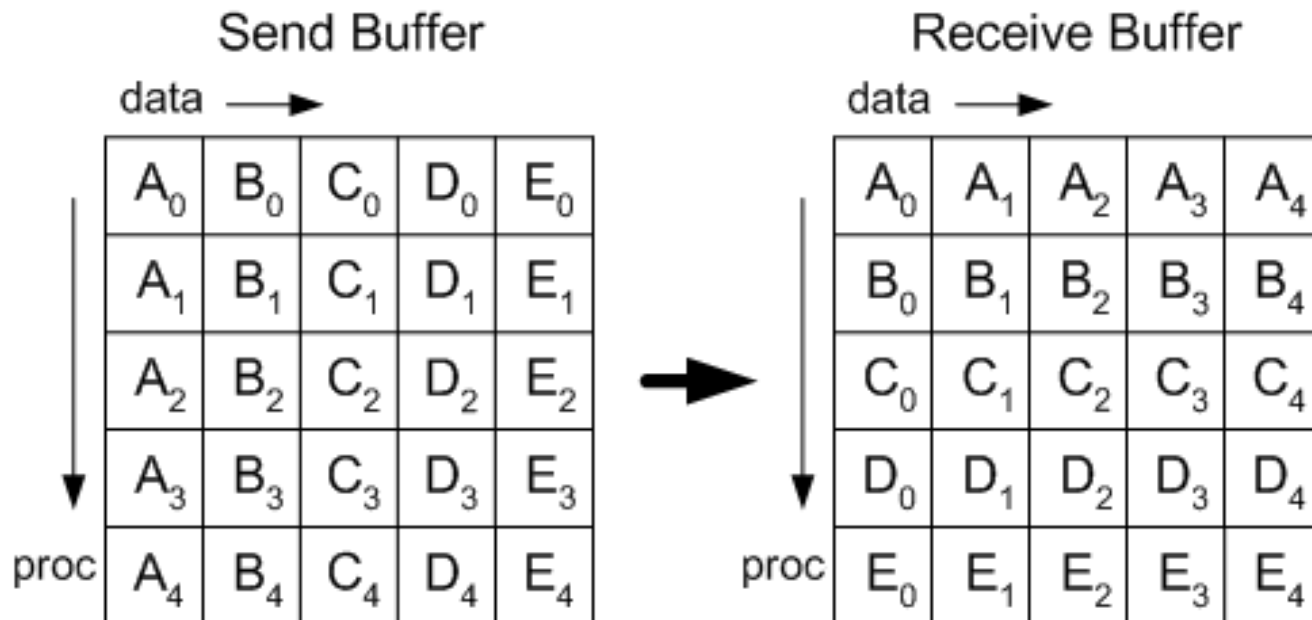
- An MPI_Gather where all processes, not just the root, receive the result.
- The j_{th} block of the receive buffer is reserved for data sent from the j_{th} rank; all the blocks are the same size.
- MPI_Allgatherv works similarly, except the block size can depend on rank j , in direct analogy to MPI_Gatherv.
- Syntax of MPI_Allgather and MPI_Allgatherv quite close to MPI_Gather and MPI_Gatherv, respectively.
- The main difference is that the argument root is dropped

MPI_Allgather



MPI_Alltoall

- an extension to MPI_Allgather where each process sends distinct data to each receiver.
- The j th block from process i is received by process j and stored in the i -th block



MPI_Alltoall/MPI_Alltoallv

- C

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

int MPI_Alltoallv(void *sbuf, int *scounts, \
    int *sdispls, MPI_Datatype stype, void *rbuf, \
    int *rcounts, int *rdispls, MPI_Datatype rtype, \
    MPI_Comm comm)
```

- Fortran

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

MPI_ALLTOALLV(sbuf, counts, sdispls, stype, rbuf, rcounts,
    rdispls, rtype, comm, ierr)
```

Note: Alltoall has the same specification as Allgather, except sbuf must contain scount*NPROC elements.

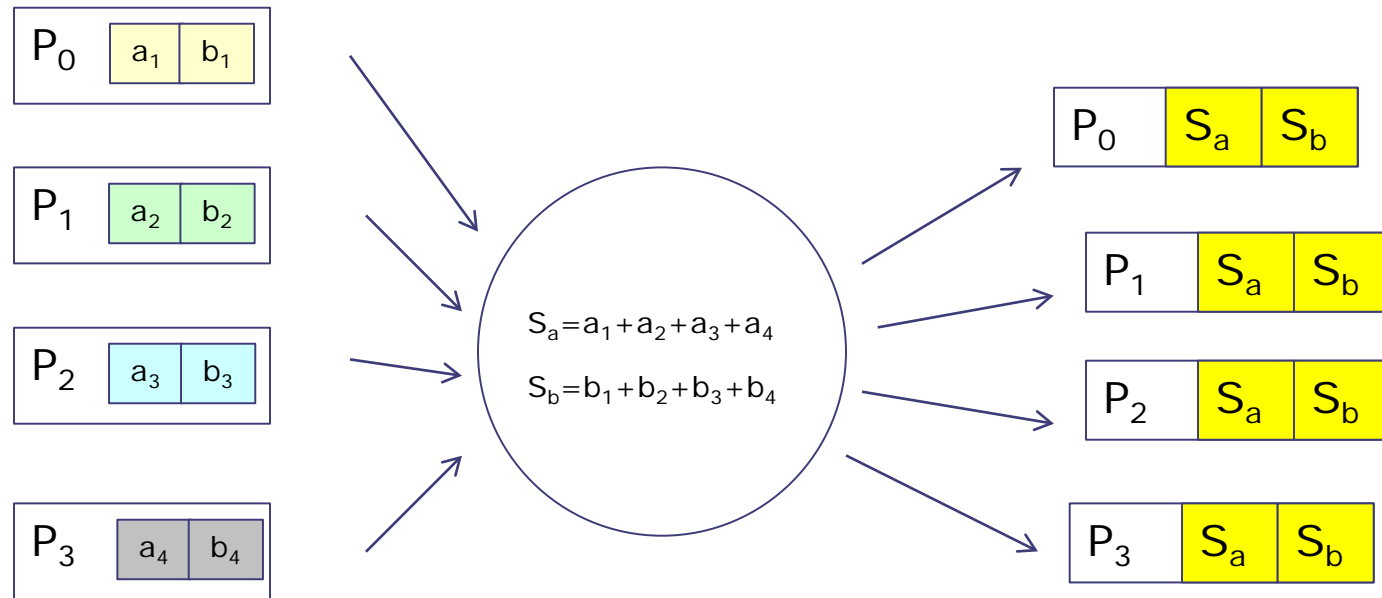
Global computing

- Two types of global computation routines: reduce and scan.
- Reduce: output the full results of applying an operation to a distributed data array.
- A scan or prefix-reduction operation performs partial reductions on distributed data.
- operation: argument of the function
- It could be predefined or user-supplied one.

Reduction

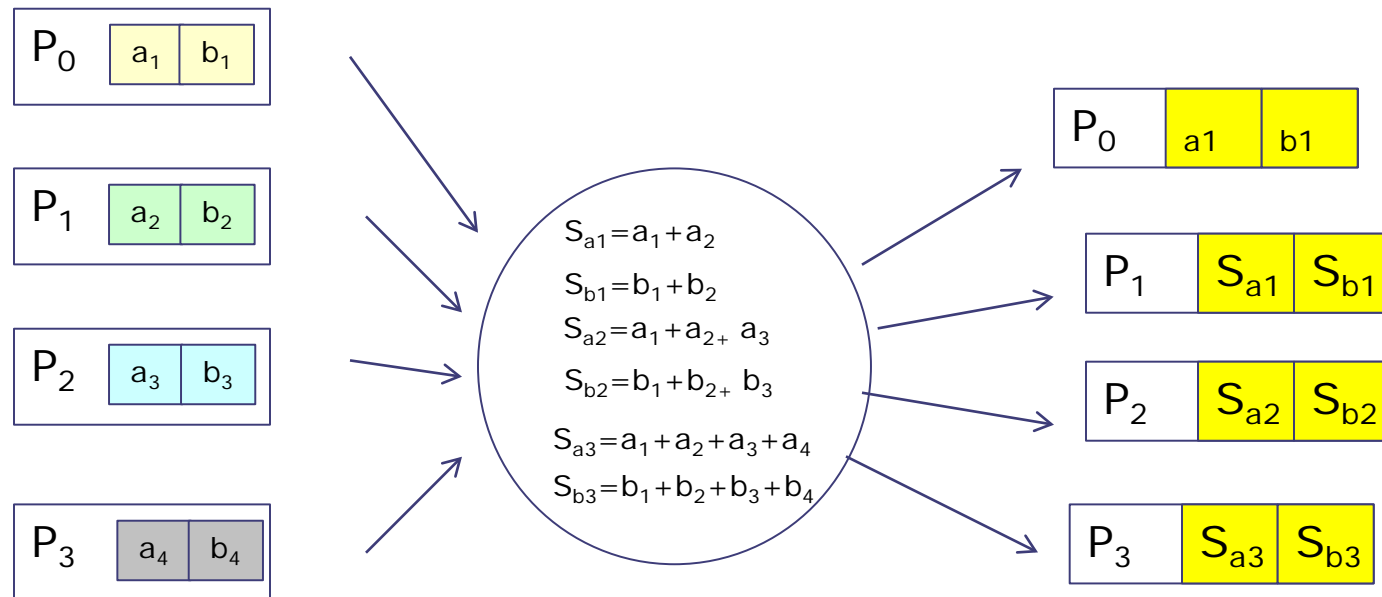
- The reduction operation allows to:
 - Collect data from each process
 - Reduce the data to a single value
 - Store the result on the root processes
 - Store the result on all processes

Reduce: parallel sum



Reduction function works with arrays other operation: product, min, max, and,

Scan: partial parallel sum



Scan function works with arrays other operation: product, min, max, and,

MPI_reduce/MPI_allreduce

- C:

```
int MPI_Reduce(void * snd_buf, void * rcv_buf, int count,  
MPI_Datatype type, MPI_Op op, int root, MPI_Comm comm)  
  
int MPI_Allreduce(void * snd_buf, void * rcv_buf, int count,  
MPI_Datatype type, MPI_Op op, MPI_Comm comm)
```

- Fortran

```
MPI_REDUCE(snd_buf,rcv_buf,count,type,op,root,comm,ierr)  
  
MPI_ALLREDUCE( snd_buf,rcv_buf,count,type,op,comm,ierr)
```

MPI_reduce/MPI_allreduce

- List of parameter for Fortran

`snd_buf` input array of type `type` containing local values.

`rcv_buf` output array of type `type` containing global results

`count` (INTEGER) number of element of `snd_buf` and `rcv_buf`

`type` (INTEGER) MPI type of `snd_buf` and `rcv_buf`

`op` (INTEGER) parallel operation to be performed

`root` (INTEGER) MPI id of the process storing the result

`comm` (INTEGER) communicator of processes involved in the operation

`ierr` (INTEGER) output, error code (if `ierr=0` no error occurs)

Predefined collective operations

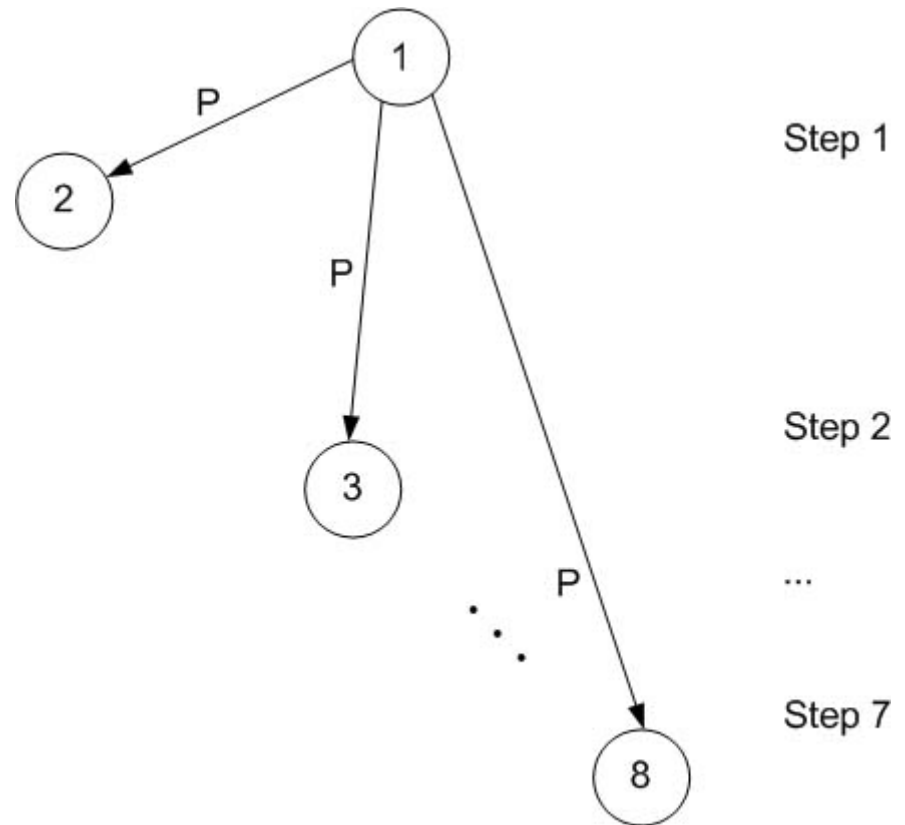
• MPI op	• Function
• MPI_MAX	• Maximum
• MPI_MIN	• Minimum
• MPI_SUM	• Sum
• MPI_PROD	• Product
• MPI LAND	• Logical AND
• MPI_BAND	• Bitwise AND
• MPI_LOR	• Logical OR
• MPI BOR	• Bitwise OR
• MPI_LXOR	• Logical exclusive OR
• MPI_BXOR	• Bitwise exclusive OR
• MPI_MAXLOC	• Maximum and location
• MPI_MINLOC	• Minimum and location

Performance issues for Collective Operation

- 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, it may not always be best to use collective communication

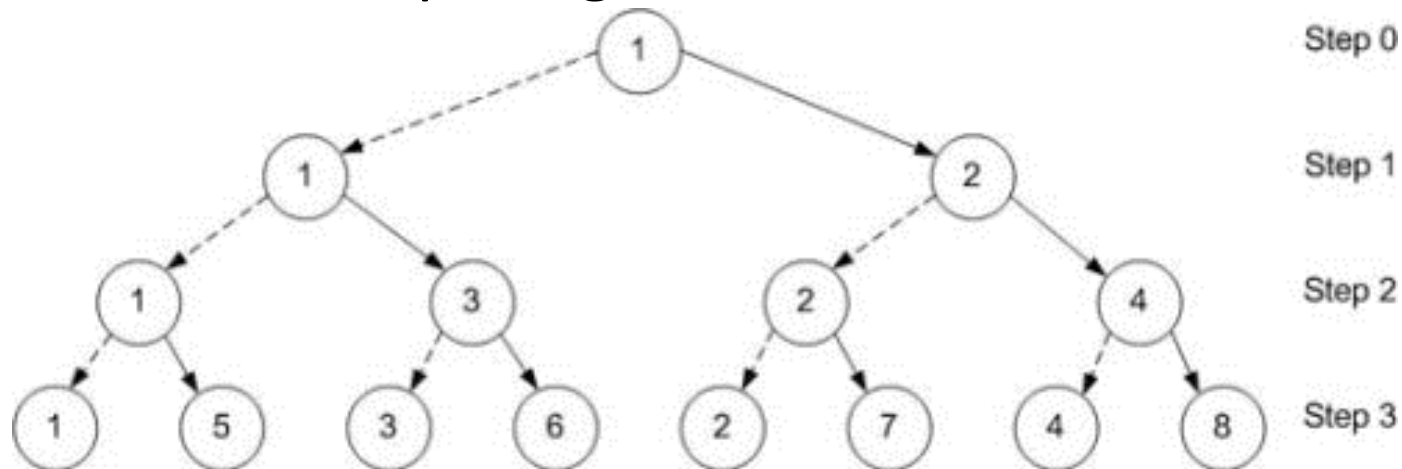
Broadcast: naïve approach

- One process broadcasts the same message to all the other processes, one at a time.
- Amount of data transferred: $(N-1)*p$
 - N = number of processes
 - p = size of message
- Number of steps: $N-1$



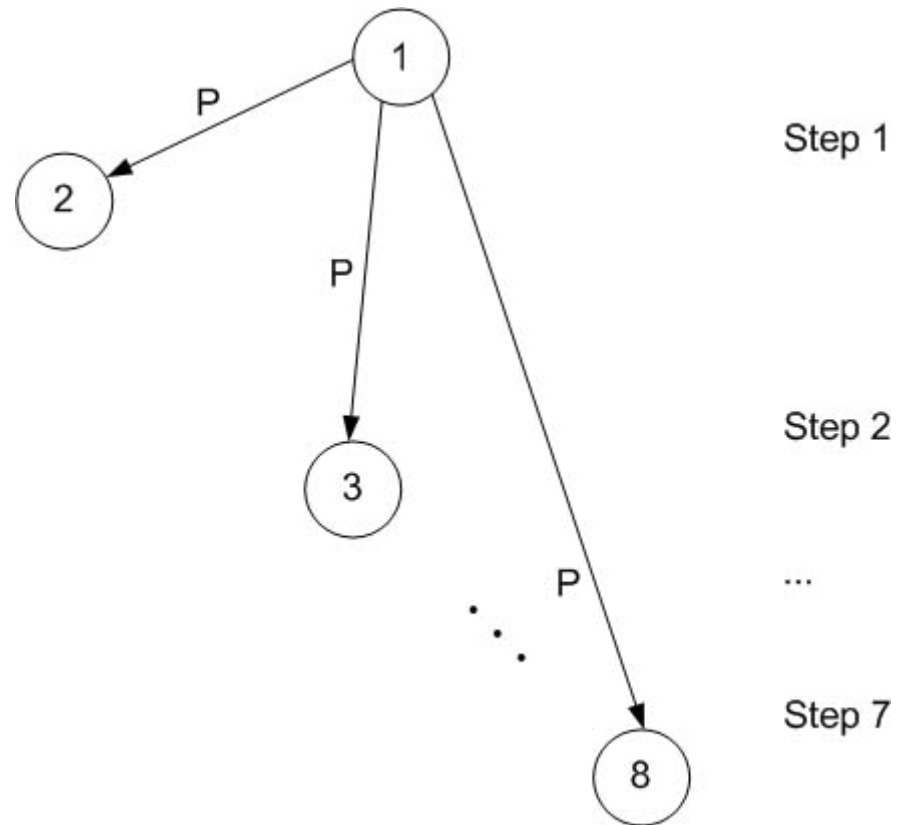
Broadcast: smarter approach

- Let other processes help propagate
- Amount of data transferred: $(N-1)*p$
 - N = number of processes
 - p = size of message
- Number of steps: $\log N$



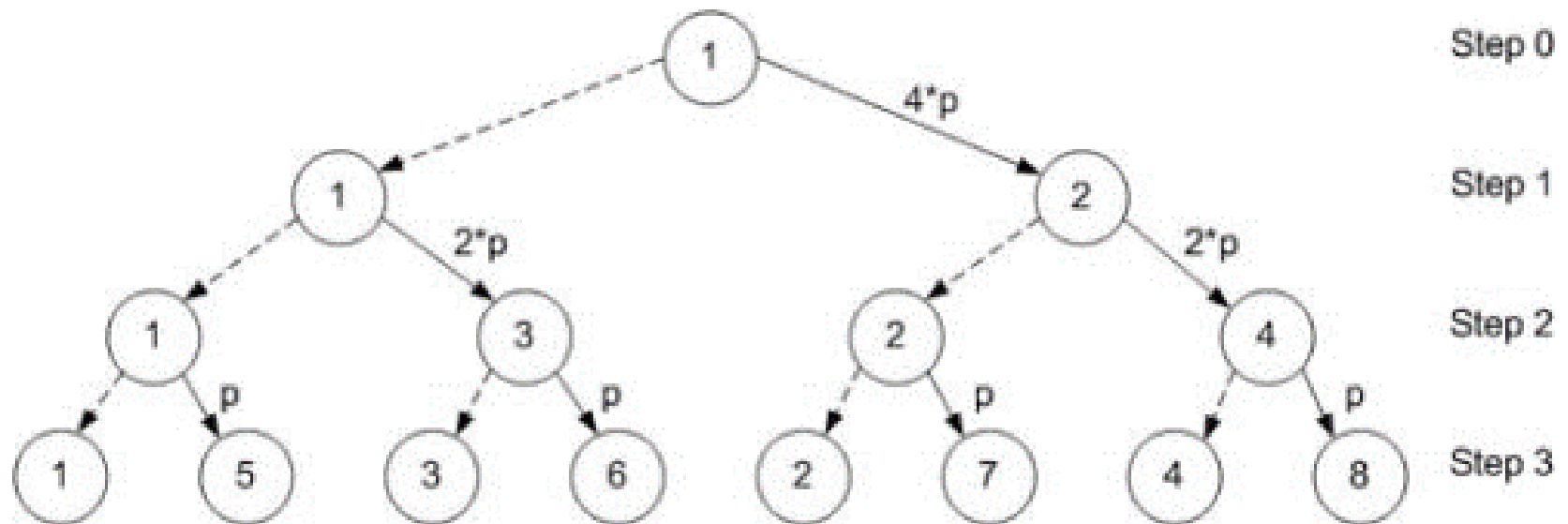
Scatter naïve approach

- One process scatter the N message to all the other processes, one at a time.
- Amount of data transferred: $(N-1)*p$
 - N = number of processes
 - p = size of message
- Number of steps: $N-1$



scatter smarter approach

- Let other processes help propagate
- In the first step, process 1 sends half of the data (size $4 \cdot p$) to process 2. In the second step, both processes can now participate: process 1 sends the half of the data it kept in step 1 (size $2 \cdot p$) to process 3, while process 2 sends one half of the data it received in step 1 to process 4. At the third step, four processes (1, 3, 2, and 4) are sending the final messages (size p) to the remaining four processes (5, 6, 7, and 8).



Performance consideration

- Amount of data transferred: $\log_2 N * N * p/2$
 - N = number of processes
 - p = size of message
- Number of steps: $\log_2 N$
- The latter scales better as N increases but for very large message sizes, there is clearly redundancy in data transfer using this second approach
- Be careful about bandwidth !