Lecture 6
MPI Collective Operations

Stefano Cozzini- AreaSciencePark

"Foundation of HPC - basic" course

DATA SCIENCE &
SCIENTIFIC COMPUTING

2022-2023 @ Università di Trieste

Agenda

- Collective operations

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()

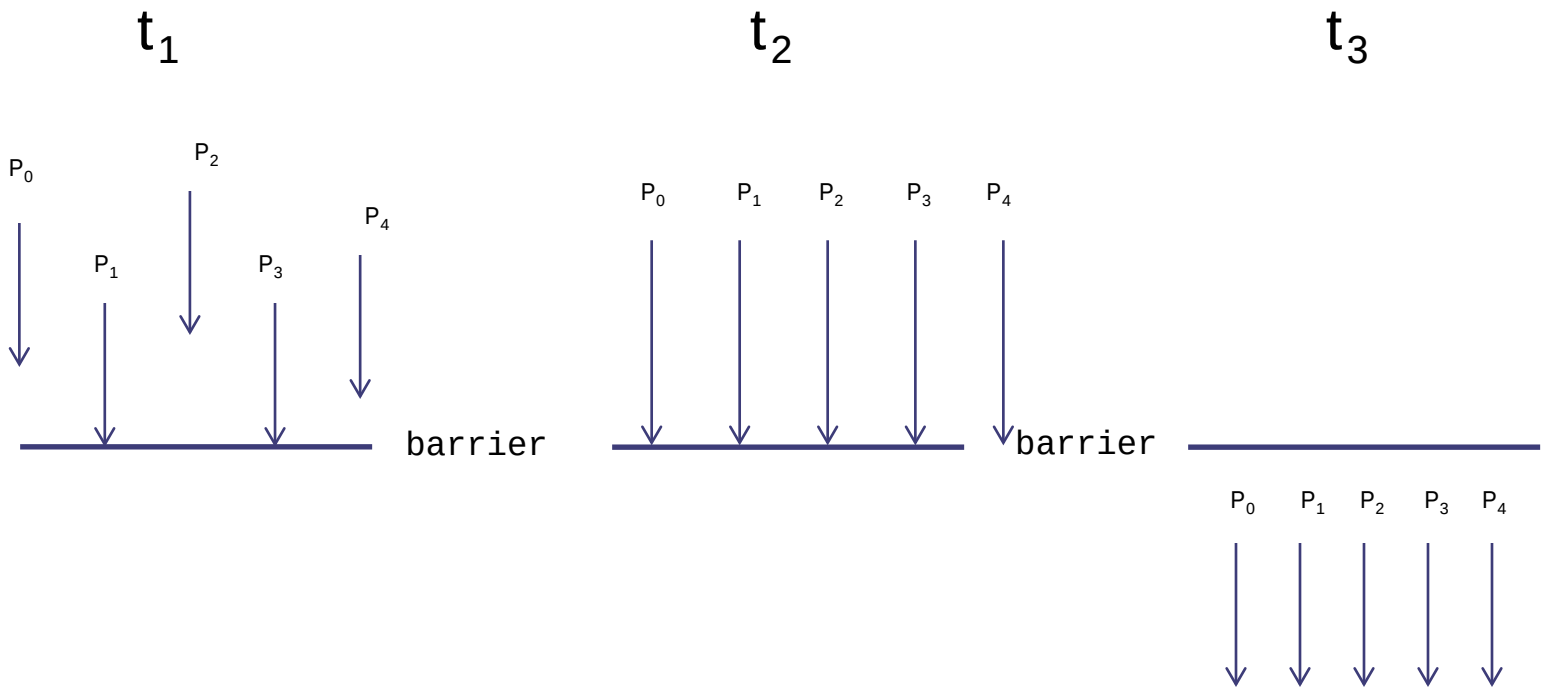
- Stop processes until all processes within a communicator reach the barrier
- Almost never required in a well-done parallel program
- Useful in measuring performance and load balancing and debugging
- Fortran:

```
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)
```

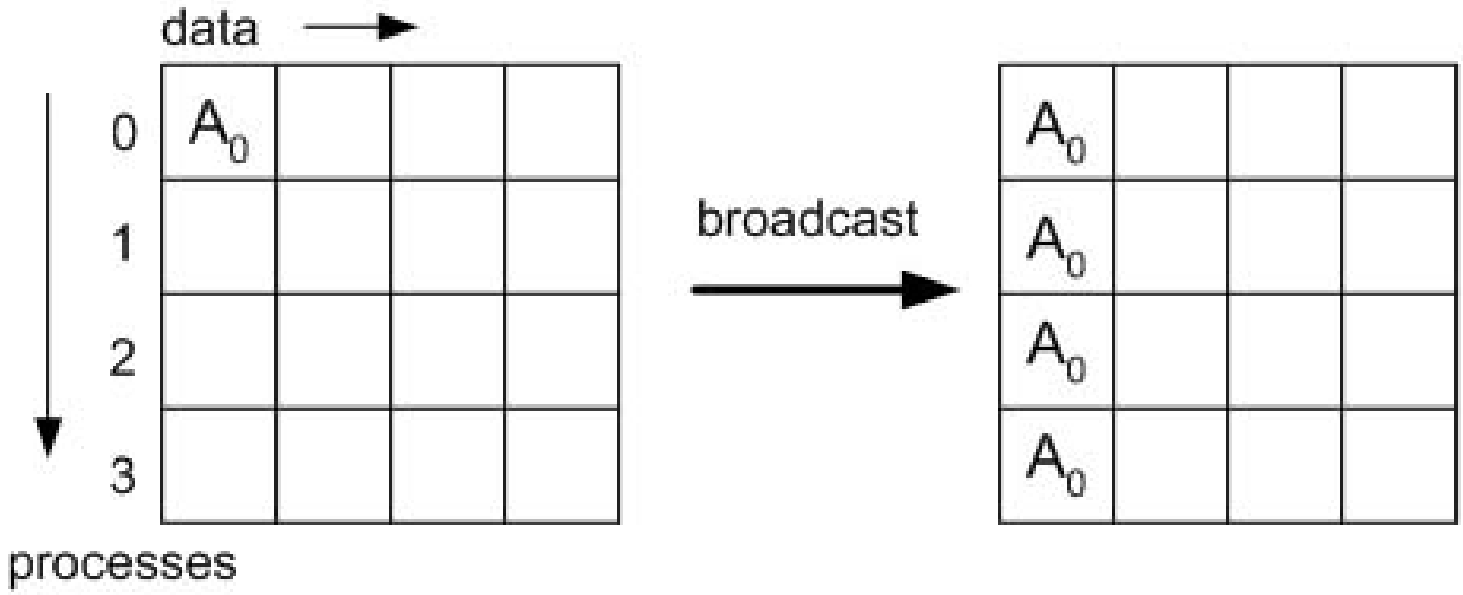
Buf array of type **type**

- C:

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

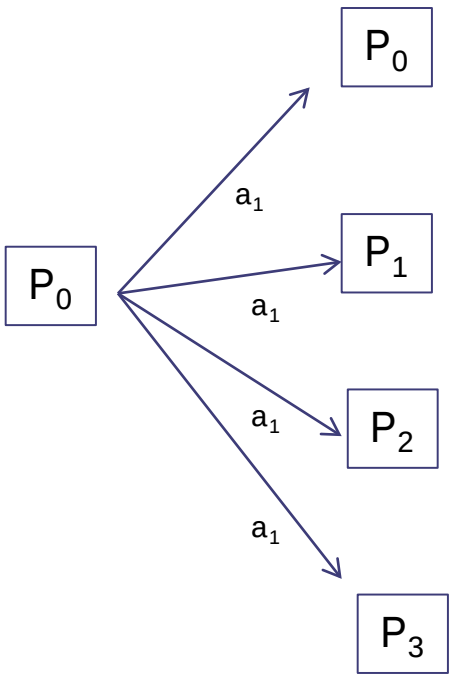
- All processes must specify same **root, rank** and **comm**

Graphical view..



MPI_bcast example

```
PROGRAM broad_cast
 INCLUDE 'mpif.h'
 INTEGER ierr, myid, nproc, root
 INTEGER status(MPI_STATUS_SIZE)
 REAL A(2)
 CALL MPI_INIT(ierr)
 CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
 CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
 root = 0
 WRITE(6,*) myid, ': a(1)=', a(1), 'a(2)=', a(2)
 IF( myid .EQ. 0 ) THEN
   a(1) = 2.0
   a(2) = 4.0
 END IF
 WRITE(6,*) myid, ': a(1)=', a(1), 'a(2)=', a(2)
 CALL MPI_BCAST(a, 2, MPI_REAL, 0, MPI_COMM_WORLD,
ierr)
 WRITE(6,*) myid, ': a(1)=', a(1), 'a(2)=', a(2)
 CALL MPI_FINALIZE(ierr)
 END
```

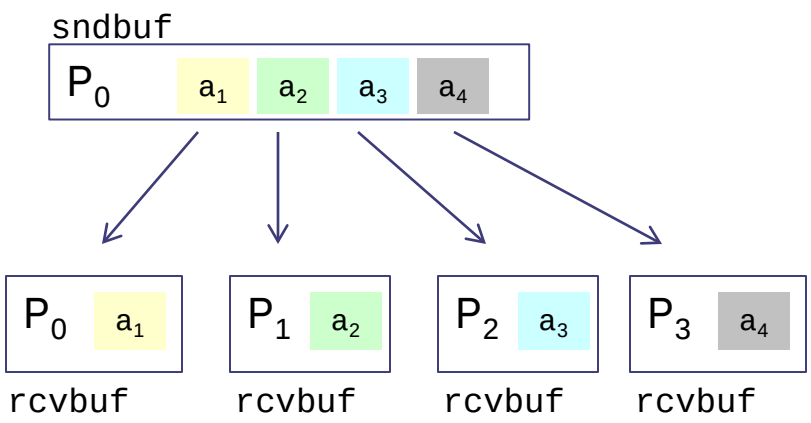


Gather/Scatter

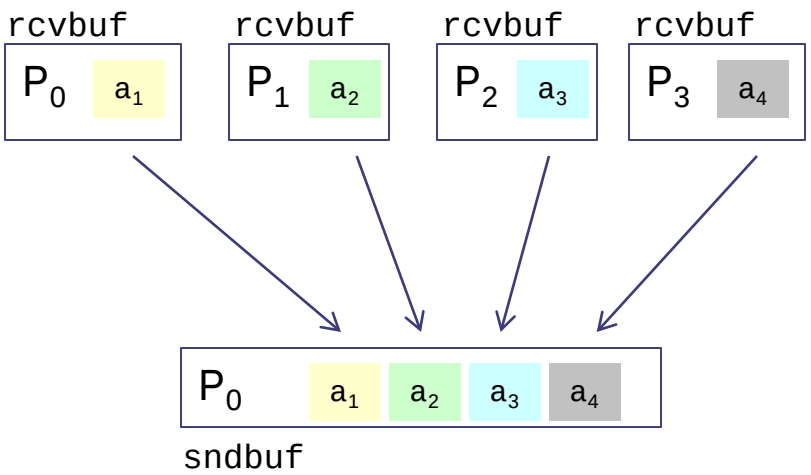
- Gather purpose
  - If an array is scattered across all processes in the group and one wants to collect each piece of the array into a specified array on a single process, the call to use is GATHER.
- Scatter purpose:
  - On the other hand, if one wants to distribute the data into n segments, where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

Scatter and Gather operations

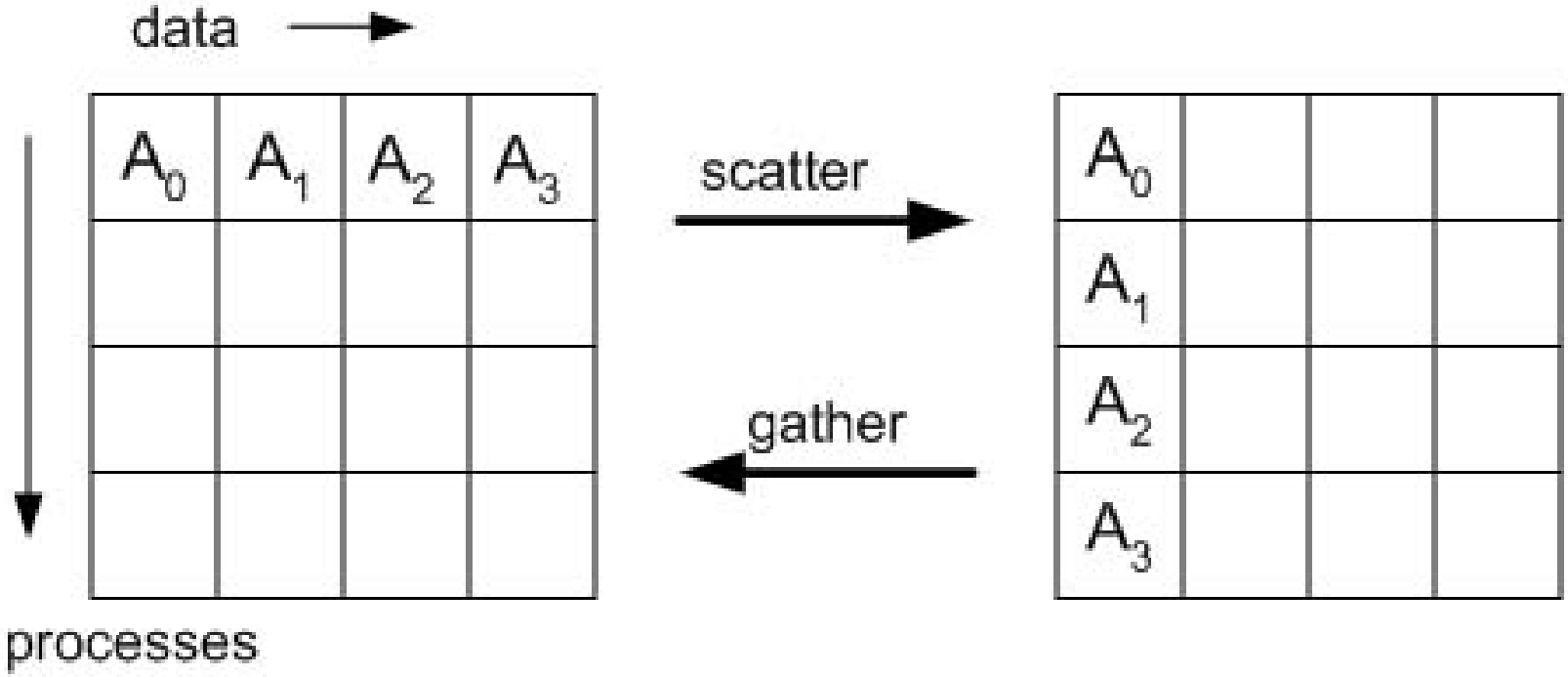
Scatter



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:

sender

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

receiver

- 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

Scatter: example

```
PROGRAM scatter
 INCLUDE 'mpif.h'
 INTEGER ierr, myid, nproc, nsnd, I, root
 INTEGER status(MPI_STATUS_SIZE)
 REAL A(16), B(2)
 CALL MPI_INIT(ierr)
 CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
 CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
 root = 0
 IF( myid .eq. root ) THEN
   DO i = 1, 16
     a(i) = REAL(i)
   END DO
 END IF
 nsnd = 2
 CALL MPI_SCATTER(a, nsnd, MPI_REAL, b, nsnd,
& MPI_REAL, root, MPI_COMM_WORLD, ierr)
 WRITE(6,*) myid, ': b(1)=', b(1), 'b(2)=', b(2)
 CALL MPI_FINALIZE(ierr)
 END
```

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:

receiver

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

senders

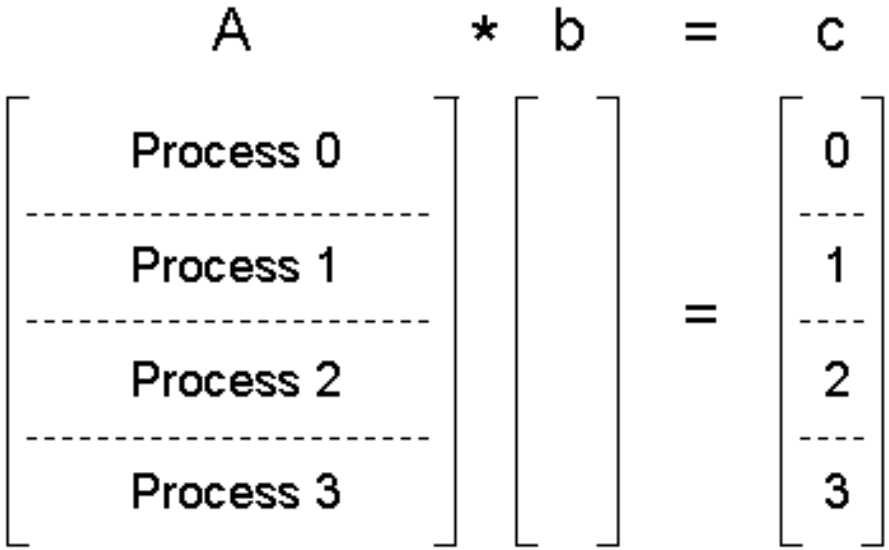
- Arguments definition are like other MPI subroutine
- **`rcvcount`** is the number of elements collected from each process, not the size of **`rcvbuf`**, that should be **`rcvcount`** times the number of process in the communicator
- The receiver arguments are significant only at root

MPI_gather example

```
PROGRAM gather
 INCLUDE 'mpif.h'
 INTEGER ierr, myid, nproc, nsnd, I, root
 INTEGER status(MPI_STATUS_SIZE)
 REAL A(16), B(2)
 CALL MPI_INIT(ierr)
 CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
 CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
 root = 0
 b(1) = REAL( myid )
 b(2) = REAL( myid )
 nsnd = 2
 CALL MPI_GATHER(b, nsnd, MPI_REAL, a, nsnd,
&  MPI_REAL, root MPI_COMM_WORLD, ierr)
 IF( myid .eq. root ) THEN
   DO i = 1, (nsnd*nproc)
     WRITE(6,*) myid, ': a(i)=', a(i)
   END DO
 END IF
 CALL MPI_FINALIZE(ierr)
 END
```

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



C- Example: Matrix-vector in parallel..

```
float Apart[25,100], b[100], cpart[25], ctotal[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, ctotal, 25,
MPI_FLOAT, root, MPI_COMM_WORLD);
```

Fortran- Example: Matrix-vector in parallel..

```
! Fortran, unlike C, stores matrices in column-major order,
! so for speed, each submatrix of A is stored as its
transpose, ApartT
REAL ApartT(100,25), b(100), cpart(25), ctotal(100)
INTEGER root
DATA root/0/
⋮
! Code that initializes ApartT and b
⋮
DO I=1,25
     cpart(I) = 0.
     DO K=1,100
          cpart(I) = cpart(I) + ApartT(K,I) * b(K)
     END DO
END DO
CALL MPI_GATHER(cpart, 25, MPI_REAL, ctotal, 25, MPI_REAL, &
      root, MPI_COMM_WORLD, ierr)
```

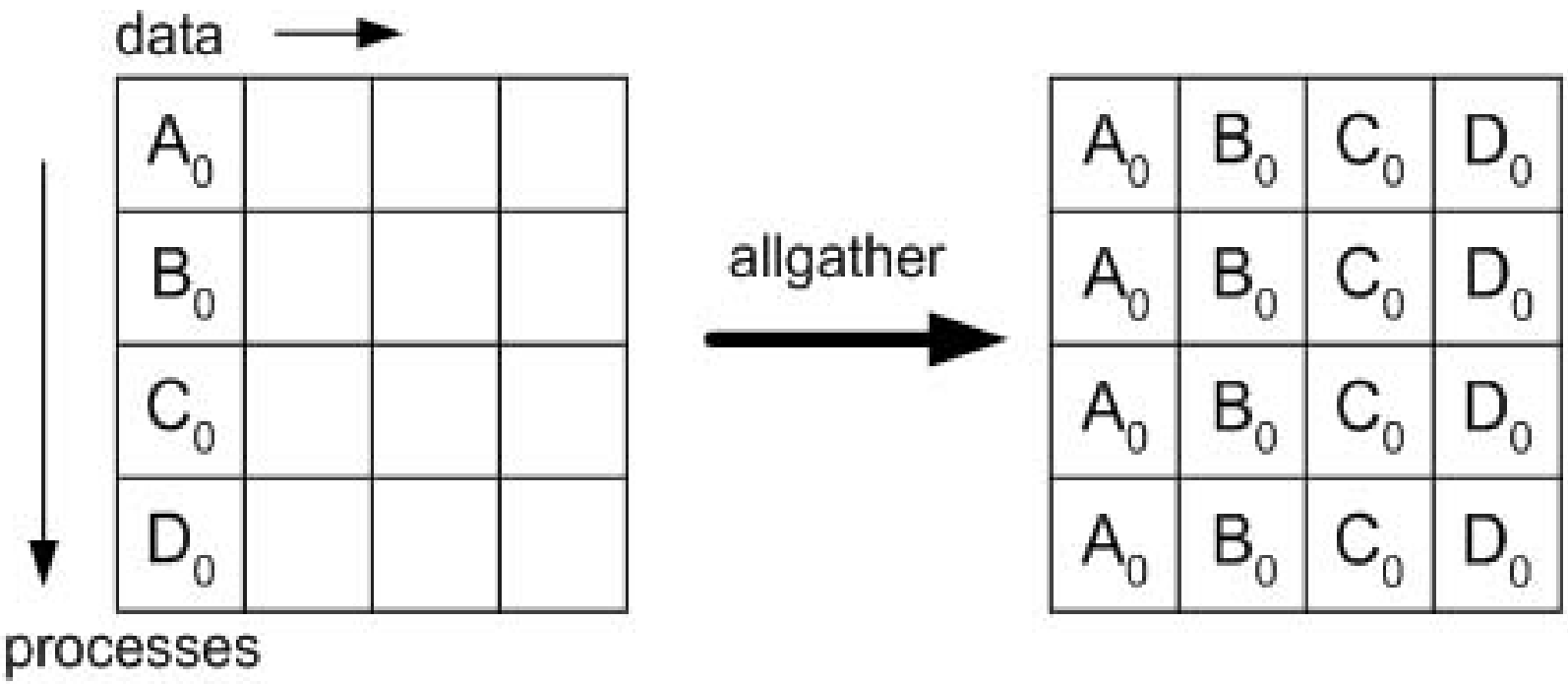
Gatherv and Scatterv

- 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

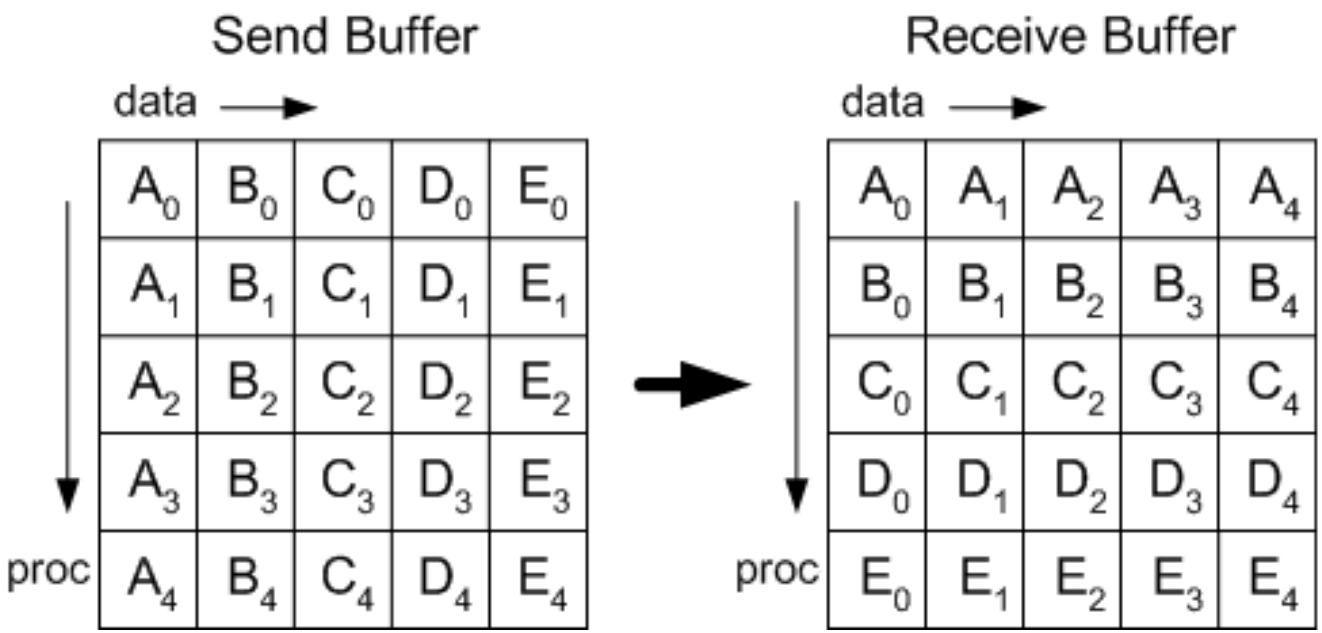
- An MPI_Gather where all processes, not just the root, receive the result.
- The j_{th} block of the receive buffer is reserved for data sent from the j_{th} rank; all the blocks are the same size.
- MPI_Allgatherv works similarly, except the block size can depend on rank j, in direct analogy to MPI_Gatherv.
- 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 jth 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, scounts, sdispls, stype, rbuf, rcounts,
     rdispls, rtype, comm, ierr)
```

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

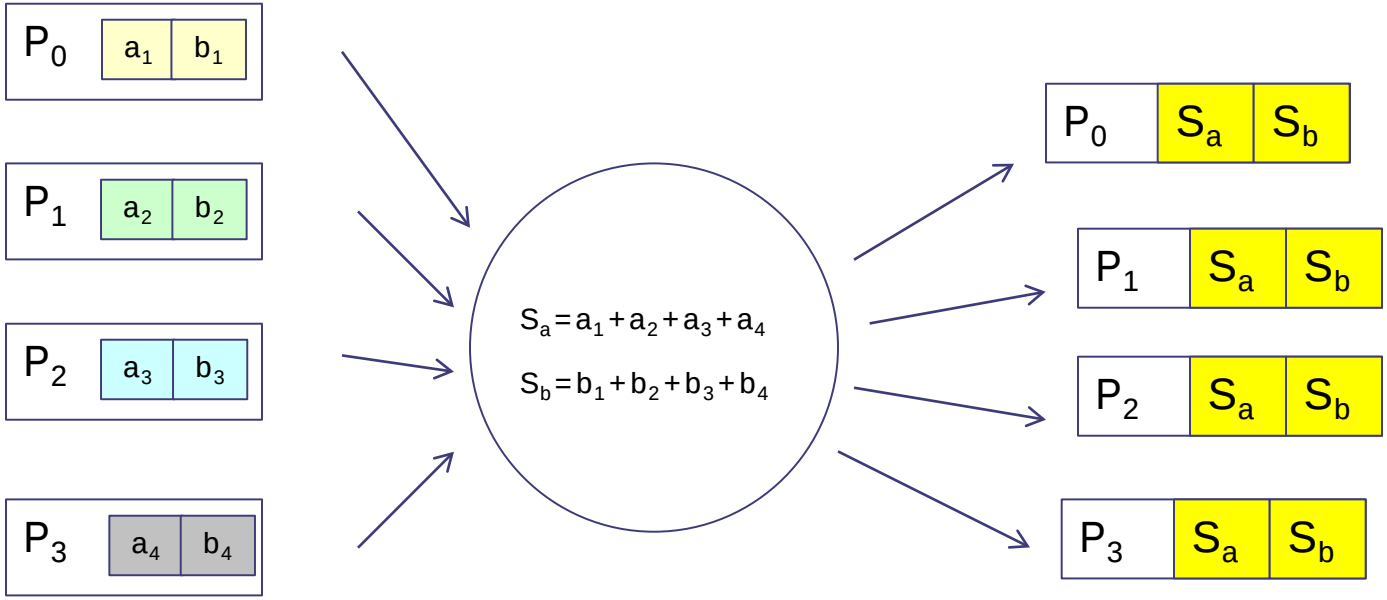
Global computing

- 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 predefinite or user-supplied one.

Reduction

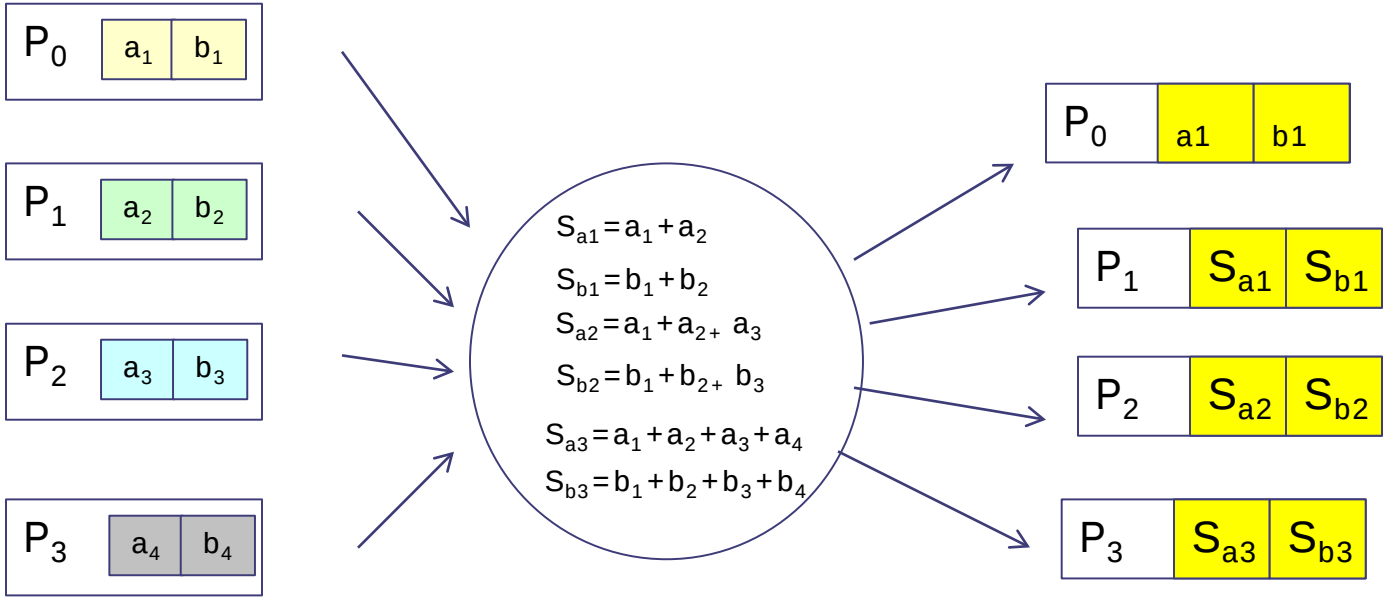
- The reduction operation allows to:
  - Collect data from each process
  - Reduce the data to a single value
  - Store the result on the root processes
  - Store the result on all processes

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 occours)

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

Reduce: example

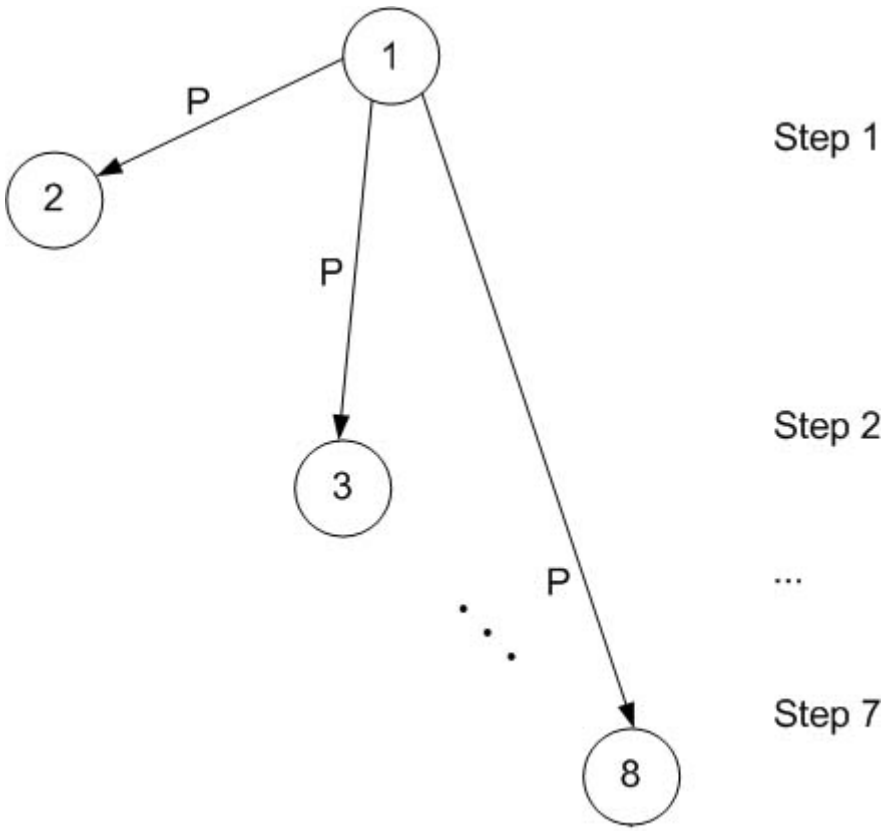
```
PROGRAM reduce
      INCLUDE 'mpif.h'
      INTEGER ierr, myid, nproc, root
      INTEGER status(MPI_STATUS_SIZE)
      REAL A(2), res(2)
      CALL MPI_INIT(ierr)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
      root = 0
      a(1) = 2.0
      a(2) = 4.0
      CALL MPI_REDUCE(a, res, 2, MPI_REAL, MPI_SUM, root,
     &  MPI_COMM_WORLD, ierr)
      IF( myid .EQ. 0 ) THEN
        WRITE(6,*) myid, ': res(1)=', res(1), 'res(2)=', res(2)
      END IF
      CALL MPI_FINALIZE(ierr)
  END
```

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

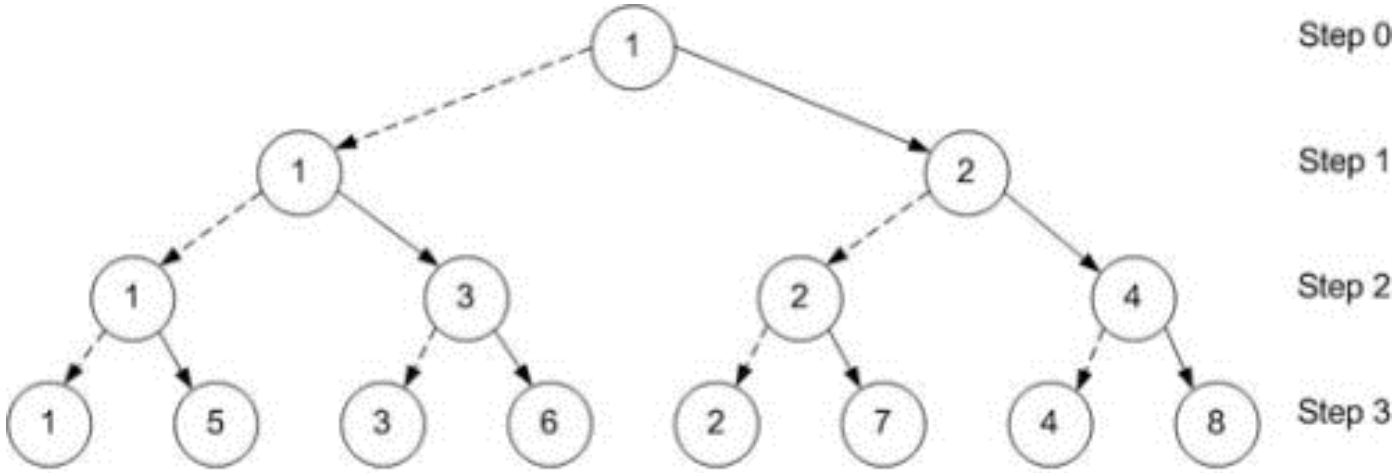
Broadcast: naïve approach

- 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



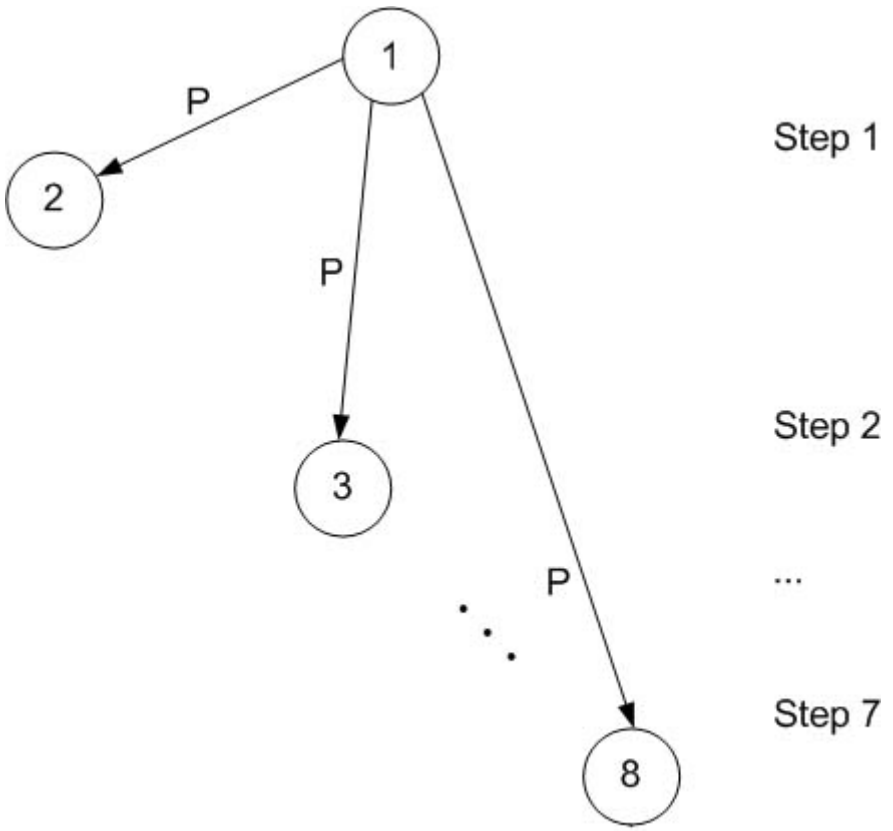
Broadcast: smarter approach

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



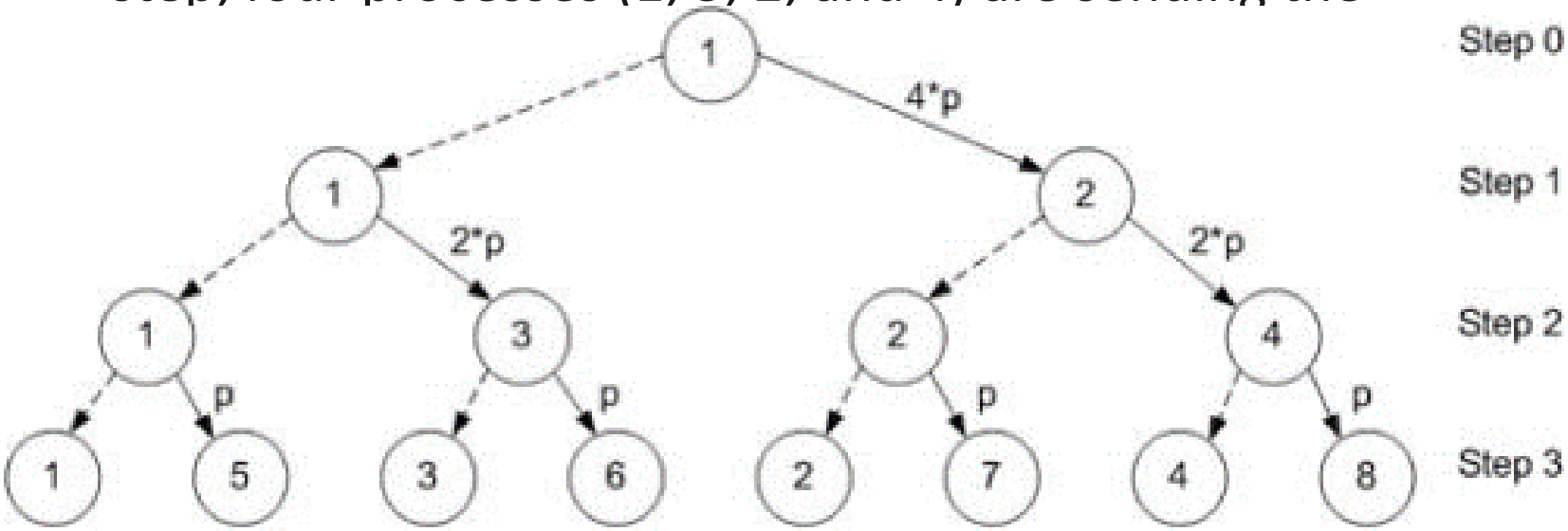
Scatter naïve approach

- 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*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*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



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 !

Tutorial/exercises

- execute and understand the simple program in the collective folder
- Play with broadcast program and check different performance of the operation manually implemented against the official routine provided by MPI implementation.
- See **`mpi_bcastcompare.c`** file
- Implement the binary tree in the program and test performance again
- Do the same for the scatter operation