MPI

MESSAGE PASSING INTERFACE

David COLIGNON, ULiège

CÉCI - Consortium des Équipements de Calcul Intensif

http://www.ceci-hpc.be

Outline

- Introduction
- From serial source code to parallel execution
- MPI functions I
  - Global Environment
  - Point-to-Point Communication
- Exercice
- MPI functions II
  - Collective Communication
  - Global Reduction Operations
  - Communication Modes

Slides & Examples

- On every CÉCI cluster :

  `/CECI/proj/INFO0939/MPI/C/`

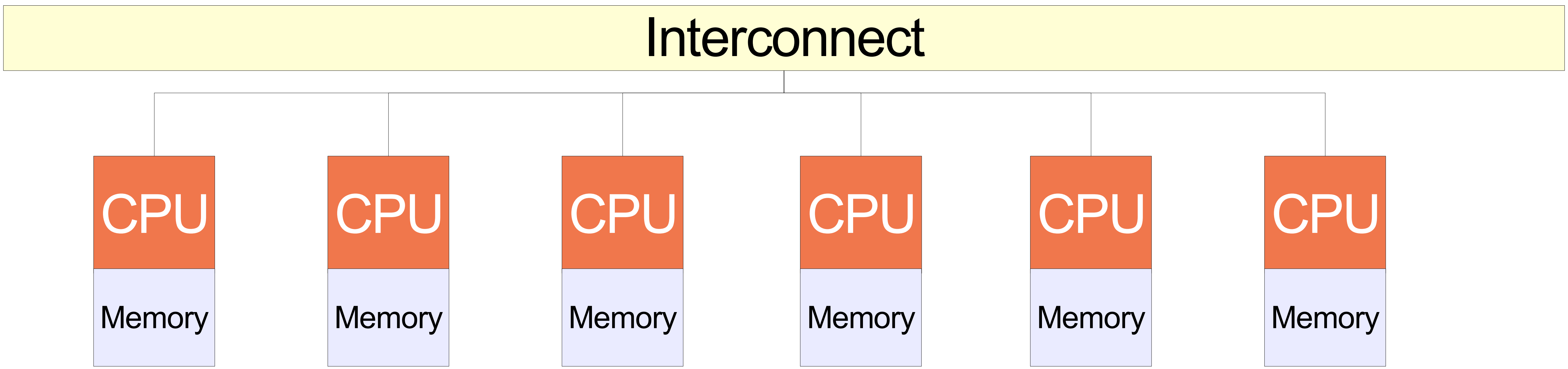
- And on NIC4:

  `module add openmpi/1.6.4/gcc-4.8.1`

  (more on this later)

Introduction: Target

Distributed Memory



Each server/node has its own memory

From one cpu per node to 2 or 4 multicore cpus...

→ Each core has its separate address space

Introduction: Goal

- What is parallel programming ?
- What are the (your) goals ?
    -
    -

Introduction: Goal

- What is parallel programming ?
- What are the (your) goals ?
    - Decrase the total execution time
    - Solve bigger problems
- Solution: Partition the work so that all nodes/cpus work together at the same time
- Partitioning a problem into workable subproblems: OK

Introduction: the MPI solution

- How to partition efficiently my problem to solve it in // ?
- How can we get nodes/cpus to work in // ?
- Solution: by **exchanging messages**
- To achieve a common parallel task, data are shared by sending/receiving "messages"
- **Message Passing Interface**: most widely used standard for parallel programming

Introduction: the MPI solution

- It's not a new programming language: it's a library of normalized functions for inter-process communication (can be called from Fortran, C, C++, ...)
    - Every cpu runs the same executable
    - Processes communicate with each other through the "infrastructure" provided by MPI.
- At first, no need to know the details of the implementation. You just need to know how to take advantage of it

Introduction: the MPI solution

- Carefully designed to permit maximum performance on a wide variety of systems
- Emphasis on **Portability** and **Efficiency**
- Hides many details but exposes many others to the programmer
- Sometimes called the "assembly language" of parallel computing

Introduction: the MPI solution

- each core (pure MPI, one process per core) runs the **same executable** and works on its **local data**
- Data are shared by sending/receiving "messages"
- MPI functions
  - Global management of the communications
  - Point-to-Point communication
  - Global communication

From serial source code to parallel execution

```c
// serial

#include <stdio.h>


int main(int argc,
         char **argv) {



printf("Hello, World !");


}

//  gcc hello_1.c

// ./a.out
```

```c
// parallel

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

int main(int argc,
         char**argv) {

MPI_Init(&argc,&argv);

printf("Hello, World !");

MPI_Finalize();
}

//  mpicc hello_mpi_1.c

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

Tips & Tricks

- Questions: Which compilers are available ? Where ?
  Is MPI installed ? Where ?
  Which version should I use ?

- Answers: **R T F M !**

```
echo $PATH
mpi "+ TAB"
module available ; module add...
mpicc -show
gcc -v ; icc -V
which mpirun ; type mpirun
```

MPI : Overview

- All MPI function begins with **MPI_**
- All MPI function returns an error code
  (= MPI_SUCCES if OK)

```
int err;
err = MPI_*( * ) ;
```

- Each node/core runs exactly the same executable:

```
mpirun -np 4 prog.exe < input.txt

mpirun -np 4 /path_to/prog.exe < /path_to/input.txt
```

MPI: Global Environment

- A minimal MPI program (like hello_mpi.c) contains:
  - `#include <mpi.h>`
  - `MPI_Init(&argc, &argv);`
    before any call to a MPI function, in order to initialize the environment
  - `MPI_Finalize();`
    after the last call to a MPI function

MPI: Global Environment

- A **Communicator** is a pool of processes that can communicate together
- MPI_COMM_WORLD is the default communicator which contains all the active processes

  `mpirun -np 8 [-machinefile mach.txt] ./a.out`
- In a communicator, each process get identified by his **rank** , from 0 to (np - 1)

MPI: Global Environment

- A process can get back the total number of processes in a communicator with:

```
int nbproc ;
MPI_Comm_size( MPI_COMM_WORLD, &nbproc )
```

- A process can know his rank inside the communicator with:

```
int myrank ;
MPI_Comm_rank( MPI_COMM_WORLD, &myrank )
```

Hello, World ! 2.0

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char **argv) {
  int nbproc, myrank ;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank( MPI_COMM_WORLD, &myrank)
  MPI_Comm_size( MPI_COMM_WORLD, &nbproc)
  printf("Hello, World ! from proc %d of %d",
                                  myrank, nbproc );
  MPI_Finalize();
}
```

Point-to-Point Communication

- Bilateral communication between two processes (emitter and receiver), identified by their rank in their common communicator
- SEND and RECEIVE are mandatory
- Message = Envelope + Body
- Envelope:
    - rank of the source process
    - rank of the destination process
    - tag (integer) to classify the messages
    - communicator

Point-to-Point Communication

- Message = Envelope + Body
- Body:
  - buffer (start of the memory area where the data are located)
  - count (number of elements)
  - datatype

MPI Type	C Type
MPI_INT	int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_CHAR	signed char
MPI_PACKED	

MPI_Send & MPI_Recv

- `MPI_Send( BUF, COUNT, DTYPE,`
  `DEST, TAG, COMM )`

- `MPI_Recv( BUF, COUNT, DTYPE,`
  `SOURCE, TAG, COMM, STATUS )`

  - envelope (`SOURCE, TAG, COMM`) determines which message can be received
  - "wild cards" `MPI_ANY_SOURCE` & `MPI_ANY_TAG` can be used
  - `STATUS` contains `SOURCE` & `TAG` (if wild cards were used), and number of received data

MPI_Send & MPI_Recv

- `MPI_Recv( BUF, COUNT, DTYPE, SOURCE, TAG, COMM, STATUS )`
- STATUS is a structure (`MPI_Status mystatus`) that contains three fields named `MPI_SOURCE` , `MPI_TAG` , and `MPI_ERROR` (The structure may contain additional fields.) `mystatus.MPI_SOURCE` , `mystatus.MPI_TAG` and `mystatus.MPI_ERROR` contain the source, tag, and error code of the received message.
- The count argument specified to the receive routine is the number of elements for which there is space in the receive buffer. This will not always be the same as the number of elements actually received.
- `MPI_Get_count( STATUS , DTYPE , COUNT )`

MPI_Send & MPI_Recv

- `MPI_Send( BUF, COUNT, DTYPE,`
  `DEST, TAG, COMM )`
- `MPI_Recv( BUF, COUNT, DTYPE,`
  `SOURCE, TAG, COMM, STATUS )`
  - error if received message longer than expected (by `COUNT` and `DTYPE`)
  - `DTYPE` of `MPI_SEND` and `MPI_RECV` must match

MPI_Send & MPI_Recv

- `MPI_Send` and `MPI_Recv` are **blocking** ! operation must be completed (...) before jump to next instruction
- **Asynchronous** communication: possible delay between Send and Receive, sent data could be buffered. Even if Send is completed, it doesn't always mean that message has already been received
- Be cautious with **Deadlocks**: two processes waiting for a message that never come

MPI_Send & MPI_Recv : Deadlocks

```
// this code hangs !
if( myrank == 0 ) {
   MPI_Recv( &b, 100, MPI_DOUBLE,
                  1, 39, MPI_COMM_WORLD, status );
   MPI_Send( &a, 100, MPI_DOUBLE,
                  1, 17, MPI_COMM_WORLD);

else if ( myrank ==1 ) {
   MPI_Recv( &b, 100, MPI_DOUBLE,
                  0, 17, MPI_COMM_WORLD, status );
   MPI_Send( &a, 100, MPI_DOUBLE,
                  0, 39, MPI_COMM_WORLD );
}
```

MPI_Send & MPI_Recv : Deadlocks

```
// this code hangs !
if( myrank == 0 ) {
   MPI_Recv( &b, 100, MPI_DOUBLE, &
                  1, 39, MPI_COMM_WORLD, &status );
   MPI_Send( &a, 100, MPI_DOUBLE, &
                  1, 17, MPI_COMM_WORLD ); }

else if ( myrank == 1 ) {
   MPI_Send( &a, 100, MPI_DOUBLE,
                  0, 39, MPI_COMM_WORLD );
   MPI_Recv( &b, 100, MPI_DOUBLE,
                  0, 17, MPI_COMM_WORLD, &status ); }
```

Exercice: Sum of the first N Integers

```
// serial solution

int main() {

  int N = 1000, sum = 0, i ;

  for( i = 1 ; i<= N ; i++)

     sum = sum + i ;

  }

  printf(" The sum from 1 to %d is: %d", N, sum);

}
```

Sum of the first N Integers: Parallel

- How to Partition ?
- Magic Formula:

```
startval = N * myrank      / nbproc + 1
endval   = N * (myrank+1) / nbproc
```

- ! Caution ! Integer division
- Process of rank 0 receive partial sums and add (and also calculate its part !)

Exercice: Sum of the first N Integers

```
// SOME HINTS
#include <stdio.h>
#include <mpi.h>
int myrank, nbproc, mytag=23, N=1000 ;
MPI_Status  mystatus ;
MPI_Init(&argc, &argv) ;
MPI_Comm_rank( MPI_COMM_WORLD , &myrank ) ;
MPI_Comm_size( MPI_COMM_WORLD , &nbproc ) ;
MPI_Send( &aaaa, 1, MPI_INT,
                dest, mytag, MPI_COMM_WORLD ) ;
MPI_Recv( &bbbb,  1, MPI_INT,
                from, mytag, MPI_COMM_WORLD, mystatus ) ;
MPI_Finalize();
```

Exercice: Sum of the first N Integers

```
// parallel solution
#include <stdio.h>
#include <mpi.h>
int main(int argc, char *argv[]){
  int myrank, np, i, j ;
  int startval, endval, partial_sum, temp_sum, N=1000 ;
  MPI_Status  mystatus1 ;
  MPI_Init( &argc , &argv );
  MPI_Comm_size( MPI_COMM_WORLD , &np );
  MPI_Comm_rank( MPI_COMM_WORLD , &myrank );
  startval = N *  myrank    / np + 1
  endval  =  N * (myrank+1) / np
  partial_sum = 0 ; tmp_sum = 0
  for( i=startval ; i<=endval ; i++ )
       partial_sum = partial_sum + i ;
  printf("Partial sum from %d to %d on proc %d equals %d",
          startval, endval, myrank,  partial_sum ) ;
```

Exercice: Sum of the first N Integers

```
  if( myrank != 0 )
      MPI_Send( &partial_sum , 1 , MPI_INT ,
                0 , 23 , MPI_COMM_WORLD ) ;
  else
      for( j=1 ; j<np ; j=j+1 ) {
          MPI_Recv( &temp_sum , 1 , MPI_INT ,
                    j , 23 , MPI_COMM_WORLD , &mystatus1);
          partial_sum = partial_sum + temp_sum ;
      }
  if( myrank == 0 )
    printf(" The sum from  1  to  %d  is:  %d  ",
            N , partial_sum );
  MPI_Finalize();
}
```

The standard send has the following form

```
MPI_SEND (buf, count, datatype, dest, tag, comm)
```

where

- `buf` is the address of the data to be sent.
- `count` is the number of elements of the MPI datatype which `buf` contains.
- `datatype` is the MPI datatype.
- `dest` is the destination process for the message. This is specified by the rank of the destination process within the group associated with the communicator `comm`.
- `tag` is a marker used by the sender to distinguish between different types of messages. Tags are used by the programmer to distinguish between different sorts of message.
- `comm` is the communicator shared by the sending and receiving processes. Only processes which have the same communicator can communicate.

The format of the standard blocking *receive* is:

```
MPI_RECV (buf, count, datatype, source, tag, comm, status)
```

where

- `buf` is the address where the data should be placed once received (the receive buffer). For the communication to succeed, the receive buffer *must* be large enough to hold the message without truncation — if it is not, behaviour is undefined. The buffer may however be longer than the data received.
- `count` is the number of elements of a certain MPI datatype which `buf` can contain. The number of data elements actually received may be less than this.
- `datatype` is the MPI datatype for the message. This must match the MPI datatype specified in the send routine.
- `source` is the rank of the source of the message in the group associated with the communicator `comm`. Instead of prescribing the source, messages can be received from one of a number of sources by specifying a *wildcard*, `MPI_ANY_SOURCE`, for this argument.
- `tag` is used by the receiving process to prescribe that it should receive only a message with a certain tag. Instead of prescribing the tag, the wildcard `MPI_ANY_TAG` can be specified for this argument.
- `comm` is the communicator specified by both the sending and receiving process. *There is no wildcard option for this argument*.
- If the receiving process has specified wildcards for both or either of `source` or `tag`, then the corresponding information from the message that was actually received may be required. This information is returned in `status`, and can be queried using routines described later.

Outline

- Introduction
- From serial source code to parallel execution
- MPI functions I
  - Global Environment
  - Point-to-Point Communication
- Exercice
- MPI functions II
  - Collective Communication
  - Global Reduction Operations
  - Communication Modes

Collective Communications

- Global function called within **all** the processes
  of a specified communicator
- cannot interfere with p2p communications
- All the processes must call the <u>same sequence</u> of global functions in the <u>same order</u>
- `MPI_Barrier( MPI_COMM_WORLD) ;`

  blocks the calling process untill all others

Broadcast: MPI_Bcast

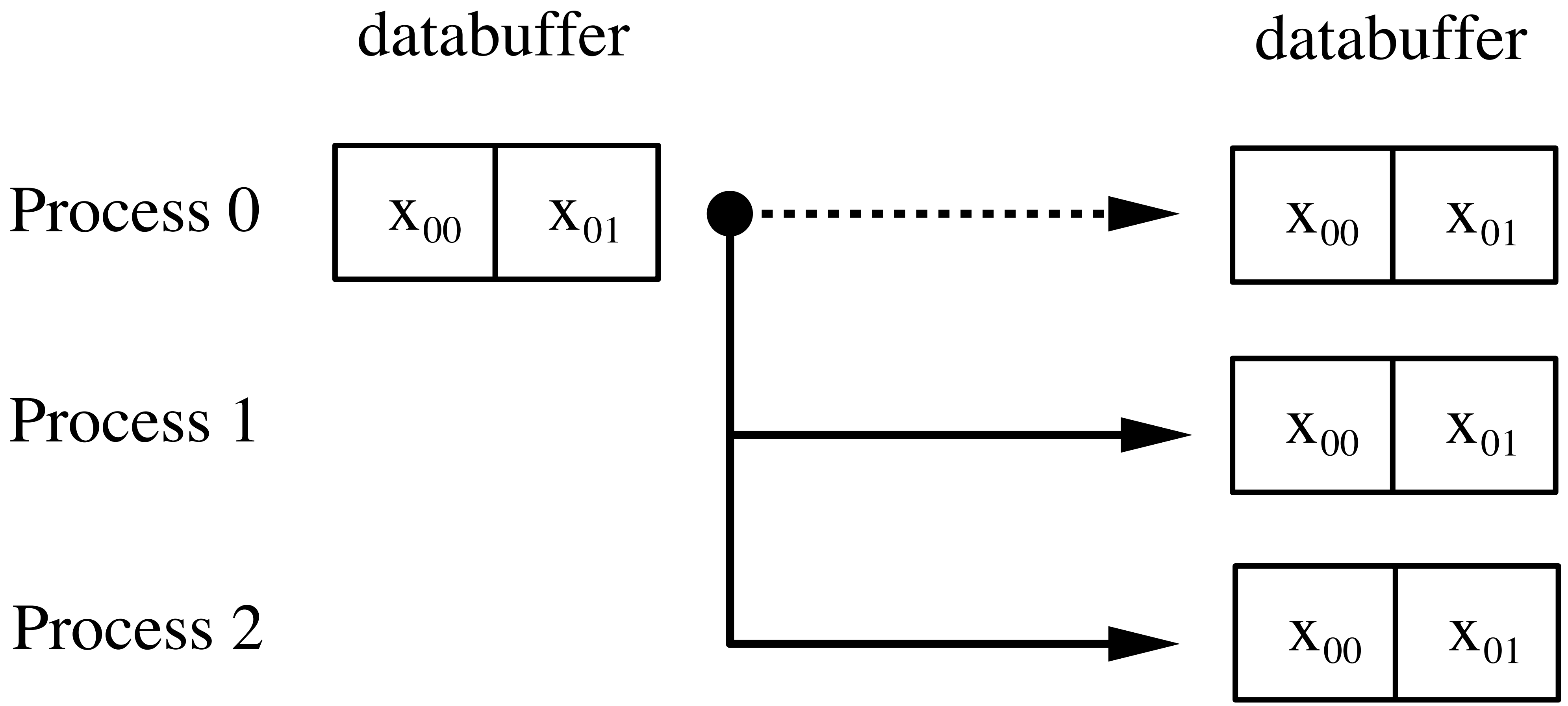
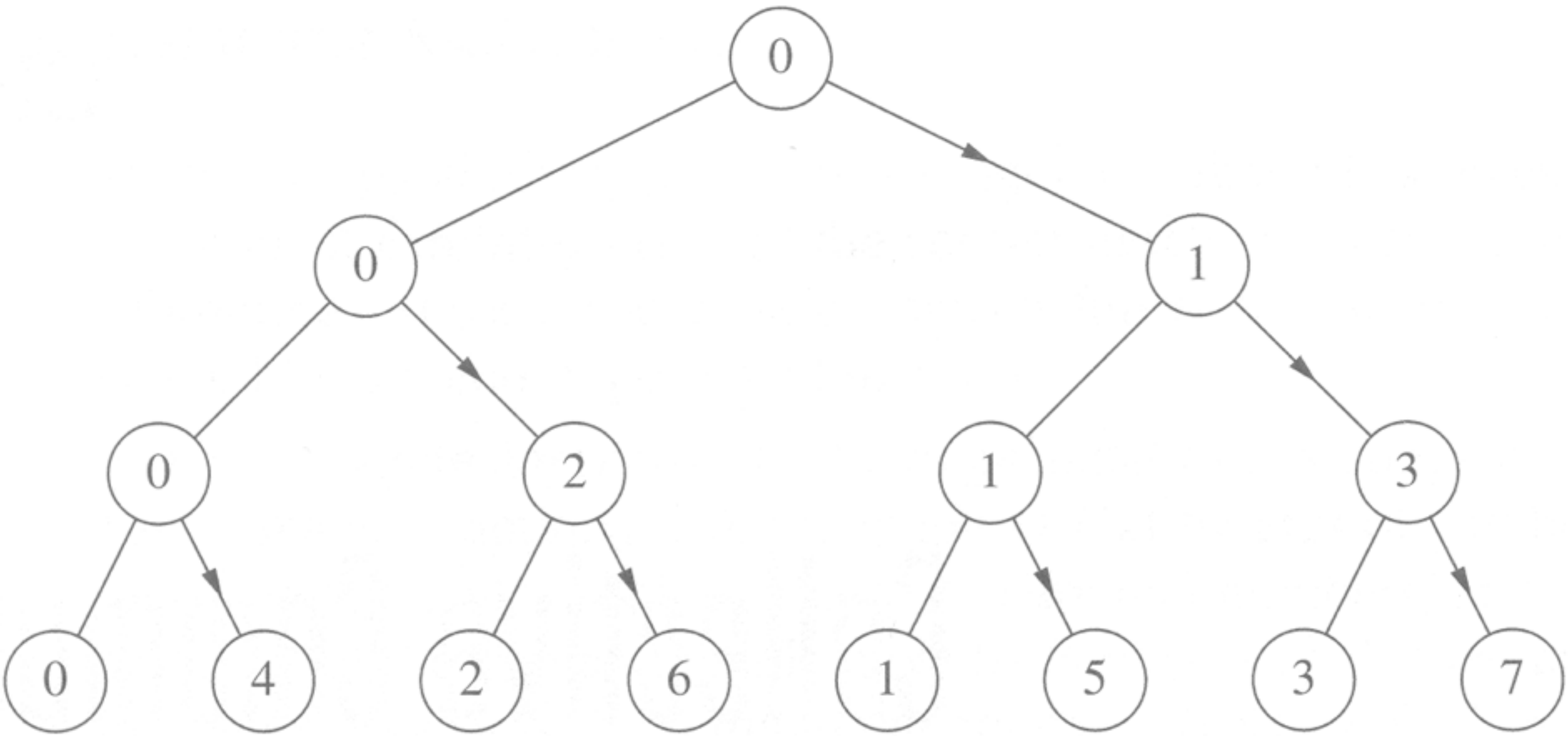


Figure 9.7: *MPI_Bcast* schematic demonstrating a broadcast of two data objects from process zero to all other processes.

MPI_Bcast



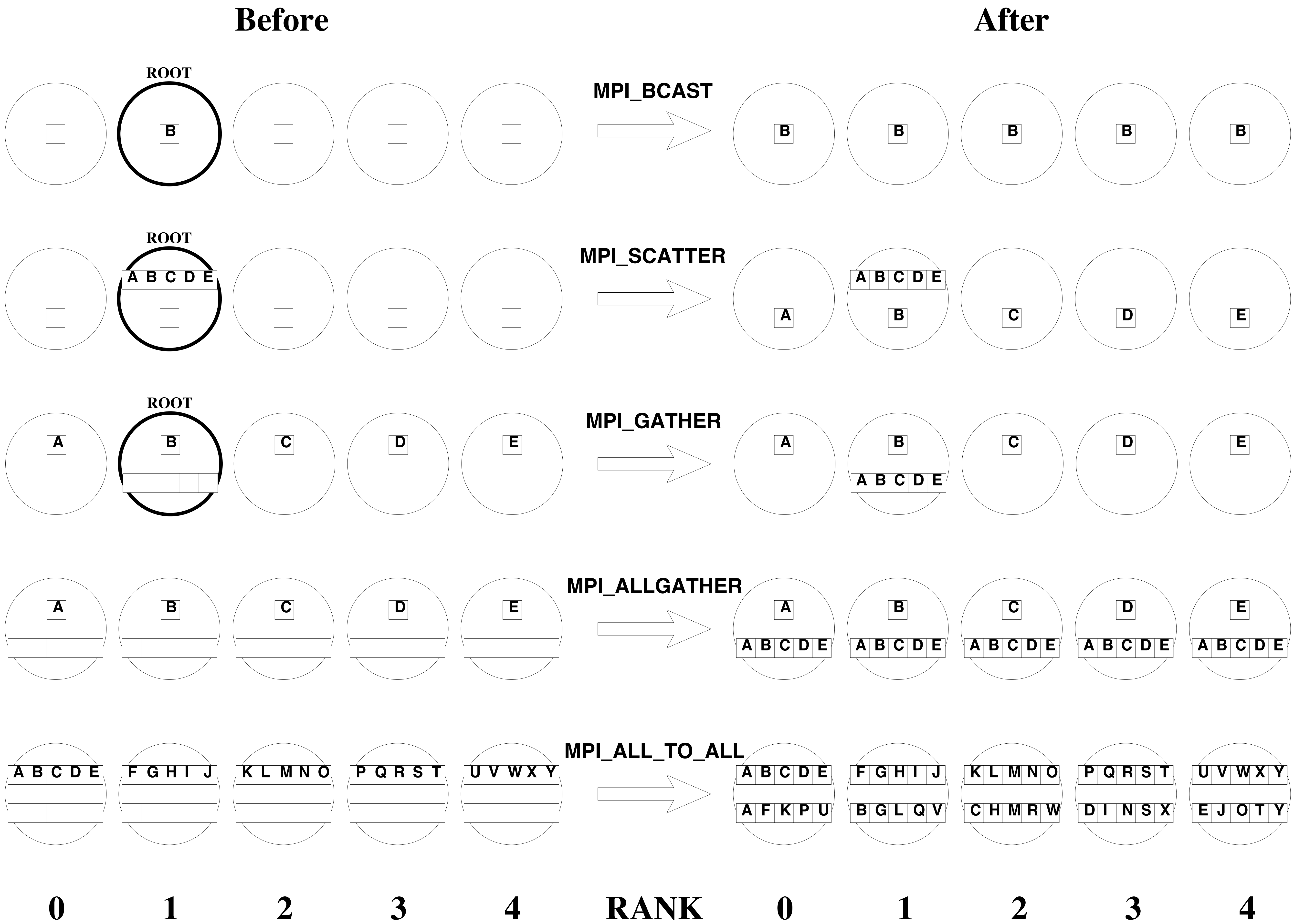
Tree Implementation

MPI_Bcast

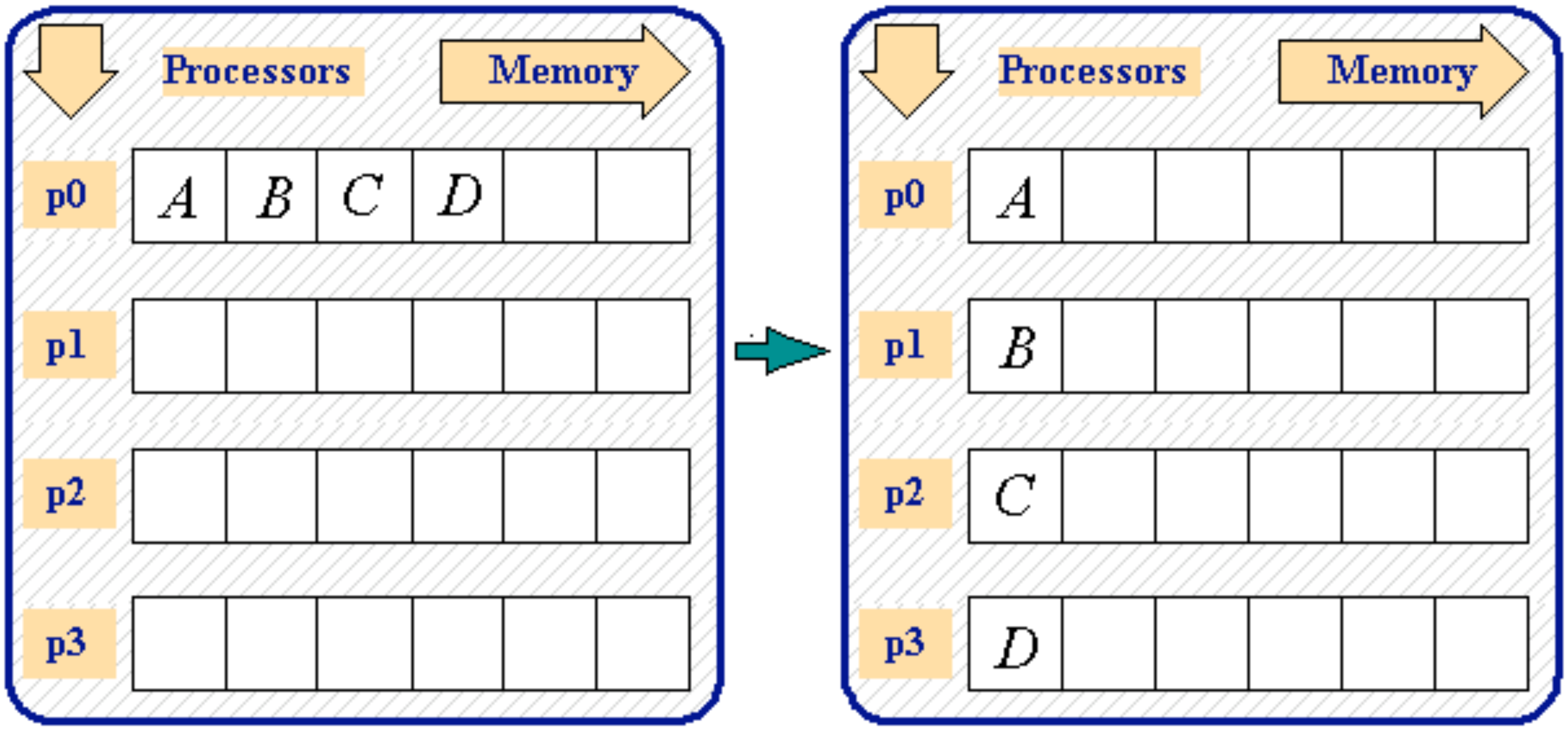
- `MPI_Bcast( buffer, count, datatype, root, comm)`
- A broadcast has a specified root process and every process receives one copy of the message from the root.
- All processes must specify the same root (and communicator).
- The root argument is the rank of the root process.
- The buffer, count and datatype arguments are treated as in a point-to-point send on the root and as in a point-to-point receive elsewhere.

MPI_Bcast Example

```
#include <stdio.h>
#include <mpi.h>
int main( int argc , char *argv[] ) {
  int myrank, np ;
  double param=0 ;

  MPI_Init( &argc , &argv );
  MPI_Comm_size( MPI_COMM_WORLD , &np );
  MPI_Comm_rank( MPI_COMM_WORLD , &myrank );

  if( myrank == 0 ) param = 23.7853 ;

  MPI_Bcast( &param , 1 , MPI_DOUBLE,
             0 , MPI_COMM_WORLD ) ;

  printf("On proc %d , after broadcast, param = %g" ,
          myrank, param ) ;

  MPI_Finalize() ;
}
```


Before
After
ROOT
MPI_BCAST
B
ROOT
MPI_SCATTER
A B C D E
ROOT
MPI_GATHER
MPI_ALLGATHER
MPI_ALL_TO_ALL
A B C D E
F G H I J
K L M N O
P Q R S T
U V W X Y
A F K P U
B G L Q V
C H M R W
D I N S X
E J O T Y
0
1
2
3
4
RANK

MPI_Scatter



Processors
Memory
p0
A
B
C
D
p1
p2
p3
Processors
Memory
p0
A
p1
B
p2
C
p3
D

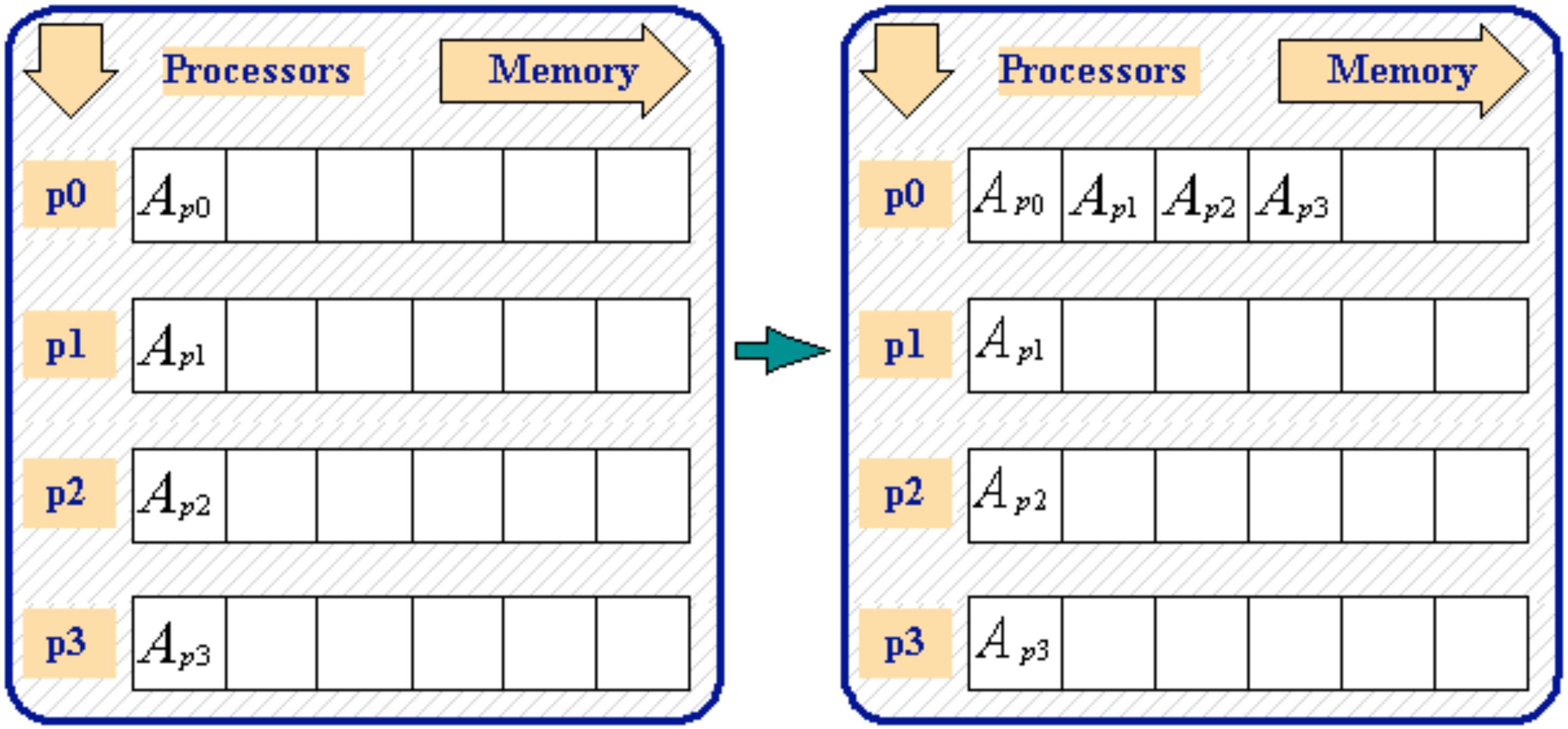
MPI_Scatter

- **`MPI_Scatter( sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)`**
- Specify a root process and all processes must specify the same root (and communicator)
- The main difference from **`MPI_Bcast`** is that the send and receive details are in general different and so must both be specified in the argument lists
- Note that the sendcount (at the root) is the number of elements to send to each process, not to send in total. Therefore if sendtype = recvtype, sendcount = recvcount.
- The sendbuf, sendcount, sendtype arguments are significant only at the root

MPI_Scatter Example

```
#include <stdio.h>
#include <mpi.h>
int main( int argc , char *argv[] ) {
  int i, myrank, np, sendcount, recvcount=1 ;
  double array[8], myparam ;
  MPI_Init( &argc , &argv );
  MPI_Comm_size( MPI_COMM_WORLD , &np );
  MPI_Comm_rank( MPI_COMM_WORLD , &myrank );

  if( myrank == 0 ) {
    for( i=0 ; i<8 ; i++)
        array[i] = 10000. + i*i ;
    sendcount = 1 ;
  }
  MPI_Scatter( array     , sendcount , MPI_DOUBLE ,
               &myparam , recvcount , MPI_DOUBLE ,
               0 , MPI_COMM_WORLD ) ;

  printf("On proc %d, after scatter, myparam = %g" ,
          myrank , myparam ) ;
  MPI_Finalize();
}
```

MPI_Gather



Processors
Memory
p0
p1
p2
p3
A_{p0}
A_{p1}
A_{p2}
A_{p3}
Processors
Memory
p0
p1
p2
p3
A_{p0} A_{p1} A_{p2} A_{p3}
A_{p1}
A_{p2}
A_{p3}

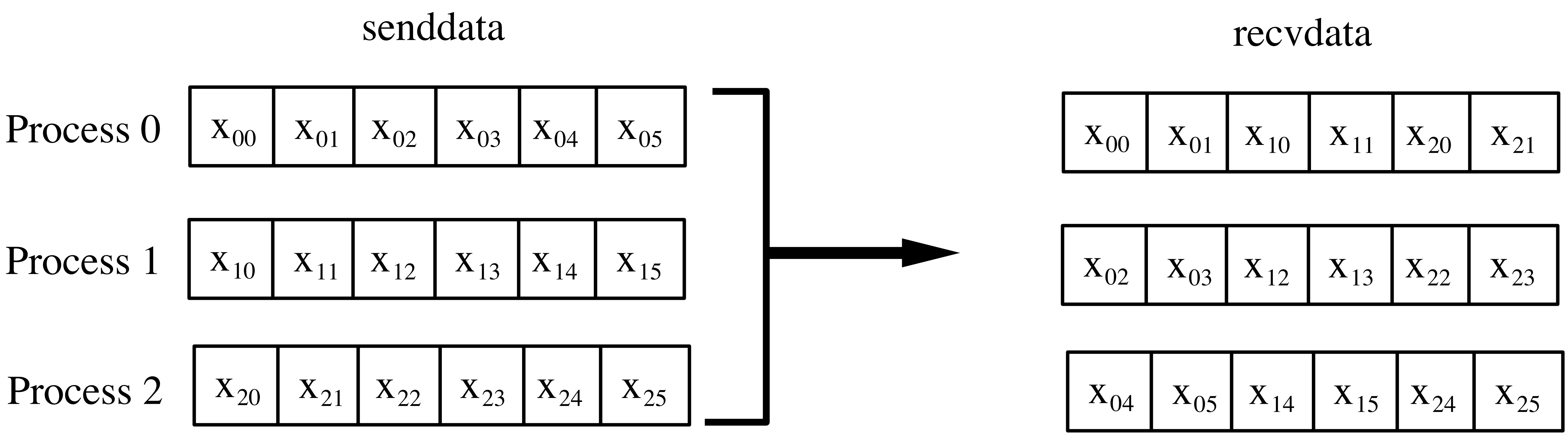
MPI_Gather

- **`MPI_Gather( sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm )`**
- The argument list is the same as for `MPI_Scatter`
- Specify a root process and all processes must specify the same root (and communicator)
- Note that the recvcount (at the root) is the number of elements to be received from each process, not in total. Therefore if sendtype = recvtype , sendcount = recvcount
- The recvbuf , recvcount , recvtype arguments are significant only at the root
- datas in recvbuf are held by rank order

MPI_Gather Example

```
#include <stdio.h>
#include <mpi.h>
int main( int argc , char *argv[] ) {
  int i, myrank, np, sendcount=1, recvcount=1 ;
  double array[8] , myparam ;
  MPI_Init( &argc , &argv );
  MPI_Comm_size( MPI_COMM_WORLD , &np );
  MPI_Comm_rank( MPI_COMM_WORLD , &myrank );
  myparam = 20000. + myrank*myrank ;

  MPI_Gather( &myparam, sendcount , MPI_DOUBLE ,
              array ,   recvcount , MPI_DOUBLE ,
              0 , MPI_COMM_WORLD ) ;

  if( myrank == 0 )
    for( i=0 ; i < 8 ; i++ )
      printf("On proc %d , after gather, array[%d] = %g " ,
              myrank, i, array[i] ) ;
  MPI_Finalize();
}
```

MPI_Alltoall



data distribution to all processes of
two data objects from each process

```
MPI_Alltoall( sendbuf, sendcount, sendtype,
              recvbuf, recvcount, recvtype,
              comm )
```

Back to Sum of the first N Integers

```
// parallel solution
#include <stdio.h>
#include <mpi.h>
int main(int argc, char *argv[]){
  int myrank, nbproc, i, j ;
  int startval, endval, partial_sum, tmp_sum, N=1000 ;
  MPI_Status  status ;
  MPI_Init( &argc , &argv );
  MPI_Comm_size( MPI_COMM_WORLD , &nbproc );
  MPI_Comm_rank( MPI_COMM_WORLD , &myrank );
  startval = N *  myrank    / nbproc + 1
  endval  =  N * (myrank+1) / nbproc
  partial_sum = 0 ; tmp_sum = 0
  for( i=startval ; i<=endval ; i++ )
       partial_sum = partial_sum + i ;
  printf("Partial sum from %d to %d on proc %d equals %d",
          startval, endval, myrank,  partial_sum ) ;
```

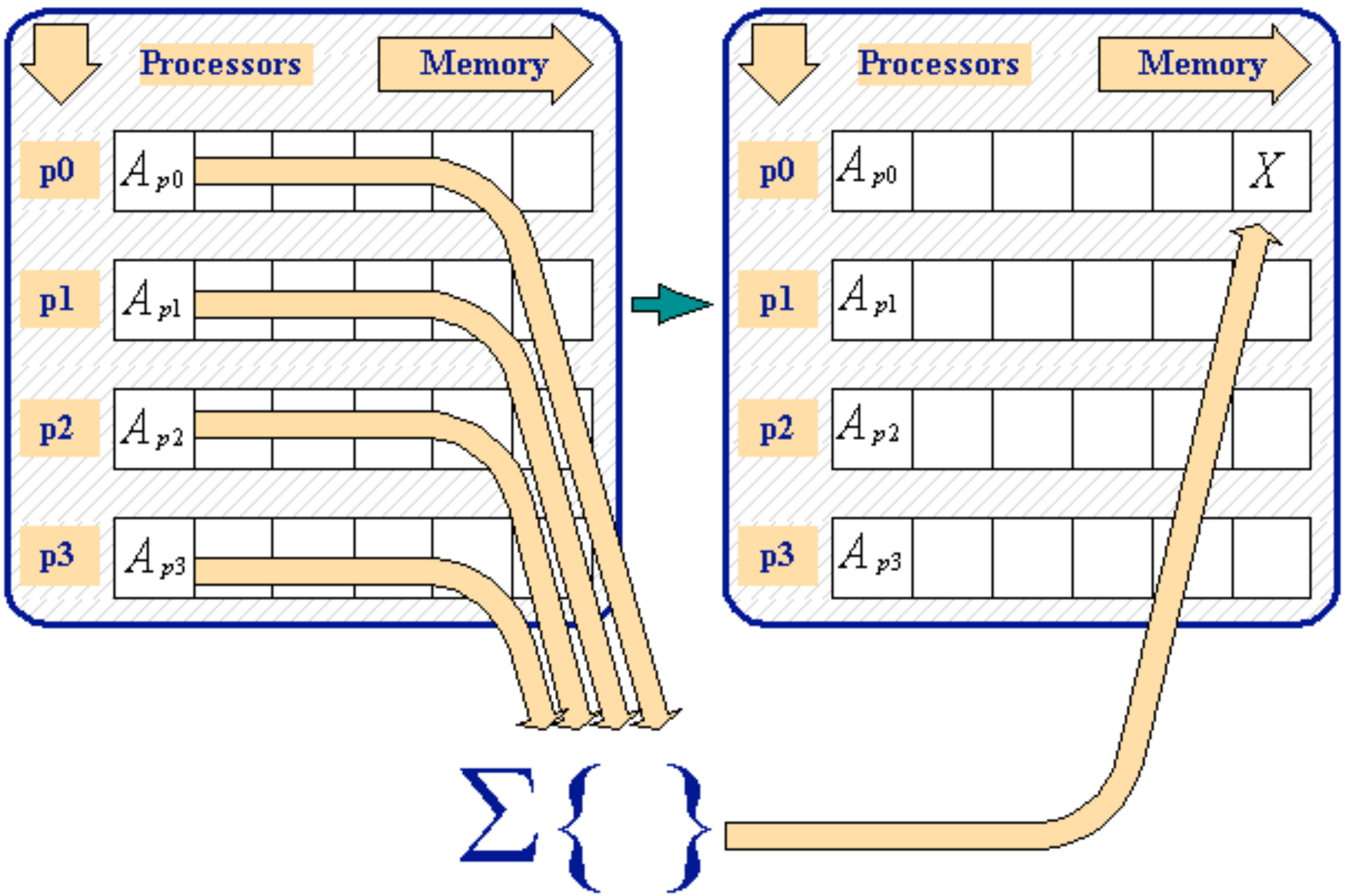
Back to Sum of the first N Integers

```
  if( myrank != 0 )
      MPI_Send( &partial_sum , 1 , MPI_INT ,
                0 , 23 , MPI_COMM_WORLD ) ;
  else
      for( j=1 ; j<np ; j=j+1 ) {
          MPI_Recv( &temp_sum , 1 , MPI_INT ,
                    j , 23 , MPI_COMM_WORLD , &status ) ;
          partial_sum = partial_sum + temp_sum ;
      }
  if( myrank == 0 )
    printf(" The sum from  1  to  %d  is:  %d  ",
            N , partial_sum );
  MPI_Finalize();
}
```

Sum of the first N Integers: MPI_Reduce

```
MPI_Reduce( &partial_sum, &tmp_sum, 1, MPI_INT,
            MPI_SUM, 0, MPI_COMM_WORLD ) ;



if( myrank == 0 )
  printf(" The sum from  1  to  %d  is:  %d  ",
          N , partial_sum );
MPI_Finalize();
}
```

MPI_Reduce



Processors
Memory
p0
p1
p2
p3
A_{p0}
A_{p1}
A_{p2}
A_{p3}
Processors
Memory
p0
p1
p2
p3
A_{p0}
A_{p1}
A_{p2}
A_{p3}
X
$\Sigma\{\ \}$

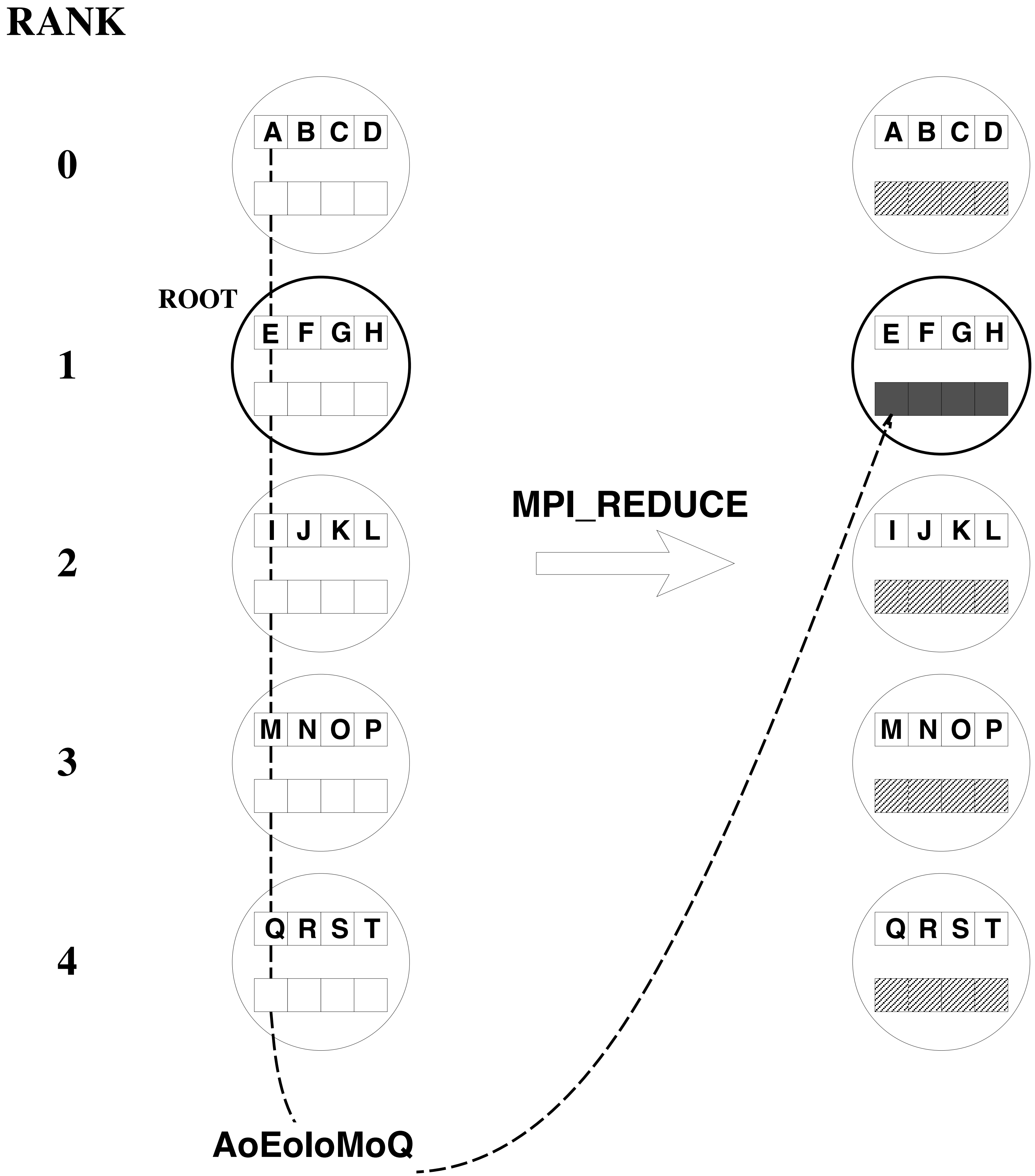
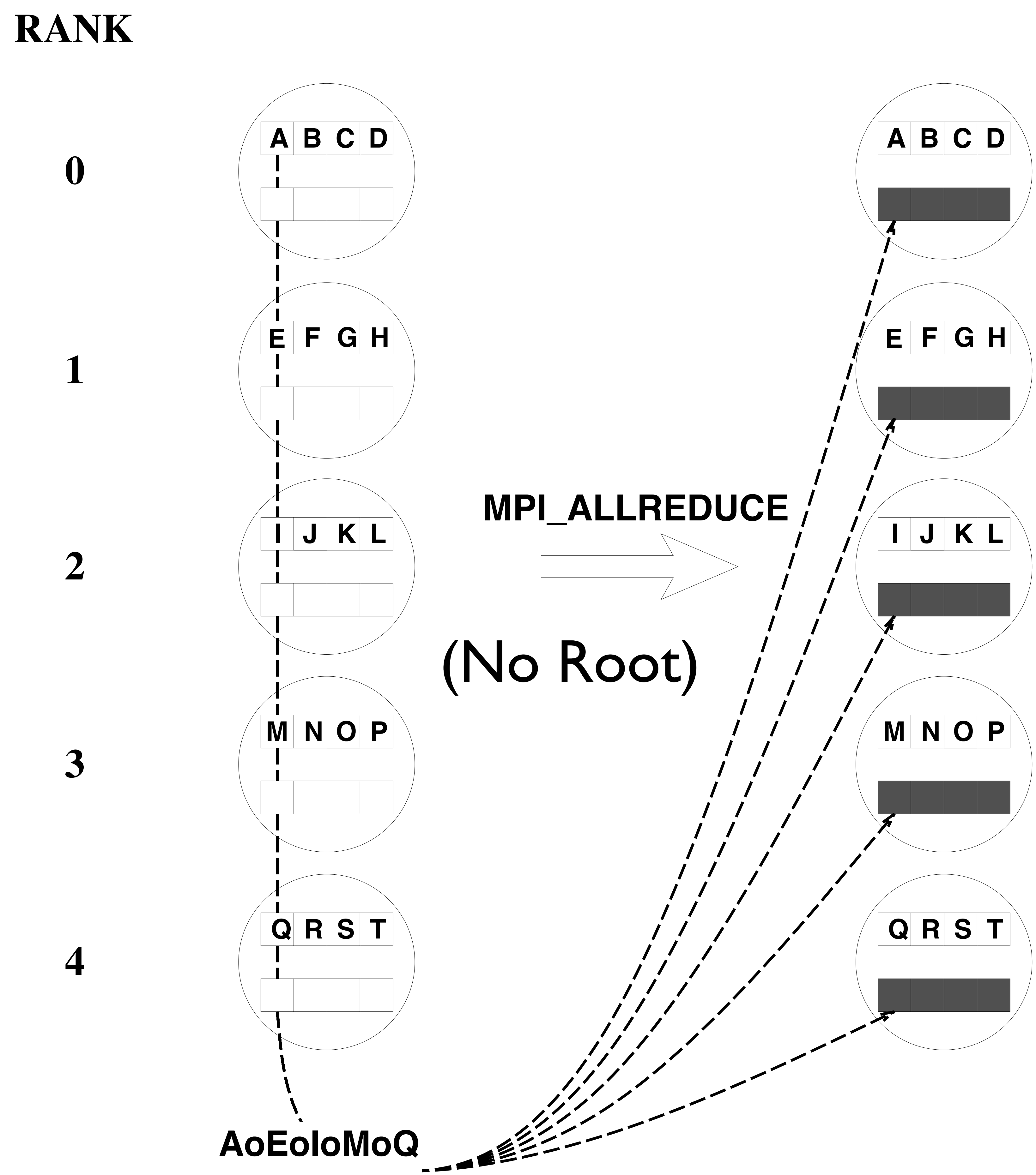

RANK
0
1
2
3
4
ROOT
A B C D
E F G H
I J K L
M N O P
Q R S T
MPI_REDUCE
AoEoIoMoQ

Table 7: Predefined operators

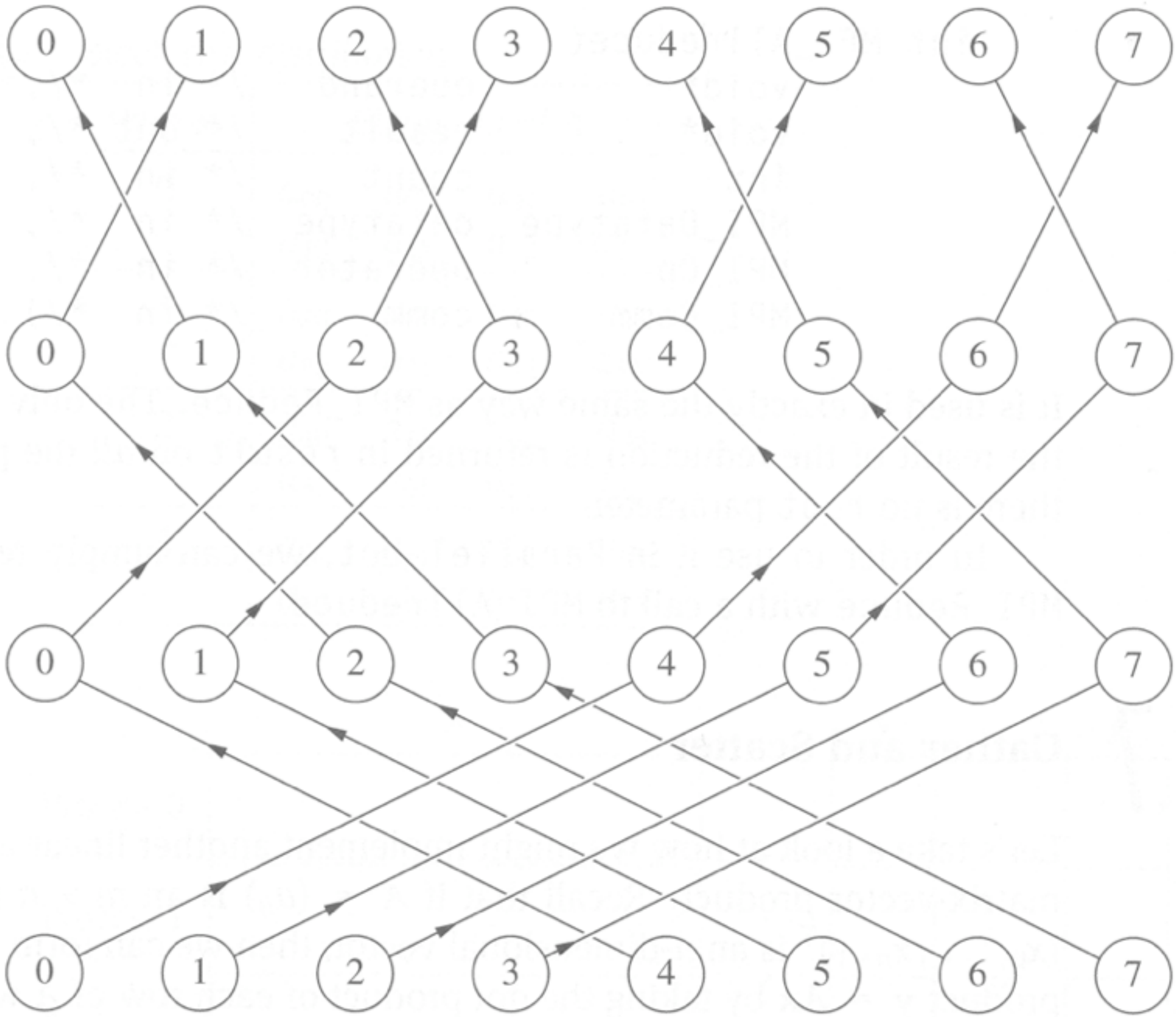
MPI Name	Function
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bitwise AND
MPI_LOR	Logical OR
MPI_BOR	Bitwise OR
MPI_LXOR	Logical exclusive OR
MPI_BXOR	Bitwise exclusive OR
MPI_MAXLOC	Maximum & location
MPI_MINLOC	Minimum & location

MPI_Reduce

- Exemple of Global Reduction Operation
- `MPI_Reduce( senbuf, recvbuf, count, datatype, operation, root, comm )`
- All processes must specify the same root (and communicator).
- Possibility to define your own reduction operation acting on your own datatype ...
- The operation must be associative (evaluation order doesn't account)


RANK
0
1
2
3
4
A B C D
E F G H
I J K L
M N O P
Q R S T
MPI_ALLREDUCE
(No Root)
AoEoIoMoQ

MPI_Allreduce



Communication Modes

- The standard `MPI_Send` call is **blocking** : it does not return until the message data and envelope have been safely stored away so that the sender is free to modify the send buffer.
- "**Completion**" of a send means by definition that the **send buffer can safely be re-used.**
- The message might be copied directly into the matching receive buffer, or it might be copied into a temporary system buffer.
- To be continued, see good References next slide

Communication Modes

- References:
  - Standard Send and Recv:

    http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report/node41.htm
  - Communication Modes:

    http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report/node53.htm
  - "MPI par l'exemple" :

    http://algernon.cism.ucl.ac.be/mpi/mpi.html
  - epcc "Writing Message Passing Parallel Programs with MPI"

    http://www.ia.pw.edu.pl/~ens/epnm/mpi_course.pdf

    Section 4 page 25

MPI Tips

- Load Balancing: Distribute evenly the work among all the processes
- Minimize the communication:
  Because of the latency, even a zero-byte message takes an uncompressible minimum time.
- Superpose/Mix calculation and communication

4 Communication Modes

	Completion Condition	
Synchronous Send	Only completes when the receive has completed.	MPI_Ssend
Buffered Send	Always completes (unless an error occurs), irrespective of whether the receive has completed.	MPI_Bsend
Standard Send	Either synchronous or buffered.	MPI_Send
Ready Send	Always completes (unless an error occurs), irrespective of whether the receive has completed.	MPI_Rsend
Receive	Completes when a message has arrived.	MPI_Recv

4 Communication Modes

- All four modes exist in both blocking and non-blocking forms.
- In the blocking forms, return from the routine implies completion.
- In the non-blocking forms, all modes are tested for completion with the usual routines (MPI_Test, MPI_Wait, etc.). See `mpi_isend_irec.c`

MPI References

- MPI standard :
  http://www.mpi-forum.org/
- MPICH :
  http://www.mpich.org/
- Open-MPI :
  http://www.open-mpi.org/
- Where can I learn about MPI ? Are there tutorials available ?
  http://www.open-mpi.org/faq/?category=all
- epcc "Writing Message Passing Parallel Programs with MPI"
  http://www.ia.pw.edu.pl/~ens/epnm/mpi_course.pdf
- "MPI par l'exemple" :
  http://algernon.cism.ucl.ac.be/mpi/mpi.html

MPI References (2)

- ME964: High-Performance Computing for Engineering Applications (Dan Negrut) http://sbel.wisc.edu/Courses/ME964/2008/LectureByLecture/me964Nov11.pdf
- 03-29-2011 - Running MPI on Newton. MPI Point-to-Point and Collective Communication. http://sbel.wisc.edu/Courses/ME964/2011/Lectures/lecture0329.pdf
- HLRS - Parallel Programming Workshop ONLINE
  https://fs.hlrs.de/projects/par/par_prog_ws/
  https://fs.hlrs.de/projects/par/par_prog_ws/pdf/mpi_1_rab.pdf
- A Comprehensive MPI Tutorial Resource
  http://mpitutorial.com/
- ...