MPI Collective Communications

Collective Communications Collective Communications

The sending and/ or receiving of messages to/ from groups of processors. A collective communication implies that all processors need participate in the communication.

- Involves coordinated communication within a group of processes
- No message tags used
- All collective routines block until they are locally complete
- Two broad classes :
  - Data movement routines
  - Global computation routines

MPI Collective Communications

Collective Communication

- Communications involving a group of processes.
- Called by all processes in a communicator.
- Examples:
  - Barrier synchronization.
  - Broadcast, scatter, gather.
  - Global sum, global maximum, etc.jj

MPI Collective Communications

Characteristics of Collective Communication

- Collective action over a communicator
- All processes must communicate
- Synchronization may or may not occur
- All collective operations are blocking.
- No tags.
- Receive buffers must be exactly the right size

MPI Collective Communications

Communication is coordinated among a group of processes

- Group can be constructed " **by hand**" with MPI group-manipulation routines or by using MPI topology-definition routines
- Different communicators are used instead
- No non-blocking collective operations

Collective Communication routines - Three classes

- Synchronization
- Data movement
- Collective computation

MPI Collective Communications

Barrier

A barrier insures that all processor reach a specified location within the code before continuing.

- C:

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

Broadcast

A broadcast sends data from one processor to all other processors.

- **C**:

```
int MPI_Bcast ( void *buffer, int count, MPI_ Datatype
                datatype, int root, MPI_Comm comm);
```

- MPI_Bcast sends the data stored in the buffer buf of process source to all the other processes in the group.
- The data received by each process is stored in the buffer buf The data that is broadcast consist of count entries of type data type .
- The amount of data sent by the source process must be equal to the amount of data that is being received by each process; i.e., the count and data type fields must match on all processes.

MPI Collective Computations

Global Reduction Operations

- Used to compute a result involving data distributed over a group of processes.
- Examples:
  - Global sum or product
  - Global maximum or minimum
  - Global user-defined operation

MPI Collective Computations

```
int MPI_Reduce (void *sendbuf, void *recvbuf, int
                count, MPI_Datatype datatype , MPI_Op
                op,    int root, MPI_Comm comm) ;
```

- MPI_Reduce combines the elements stored in the buffer sendbuf of each process in the group, using the operation specified in op, and returns the combined values in the buffer recvbuf of the process with rank target.
- Both the sendbuf and recvbuf must have the same number of count items of type datatype.

MPI Collective Computations

Collective Computation Operations

MPI_Name	Operation
MPI_LAND	Logical and
MPI_LOR	Logical or
MPI_LXOR	Logical exclusive or (xor)
MPI_BAND	Bitwise AND
MPI_BOR	Bitwise OR
MPI_BXOR	Bitwise exclusive OR

MPI Collective Computations

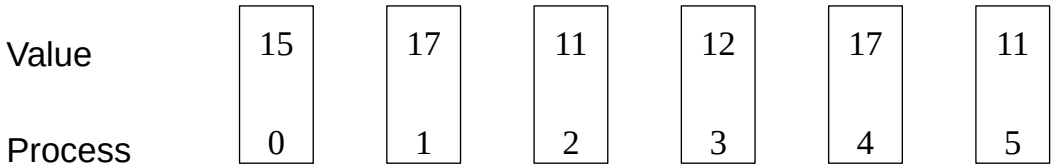
Collective Computation Operation

MPI Name	Operation
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_PROD	Product
MPI_SUM	Sum
MPI_MAXLOC	Maximum and location
MPI_MAXLOC	Maximum and location

Collective Communication Operations

- The operation `MPI_MAXLOC` combines pairs of values (v_i, l_i) and returns the pair (v, l) such that v is the maximum among all v_i's and l is the corresponding l_i (if there are more than one, it is the smallest among all these l_i's).

- `MPI_MINLOC` does the same, except for minimum value of v_i.




```
MinLoc(Value, Process) = (11, 2)
MaxLoc(Value, Process) = (17, 1)
```

An example use of the `MPI_MINLOC` and `MPI_MAXLOC` operators.

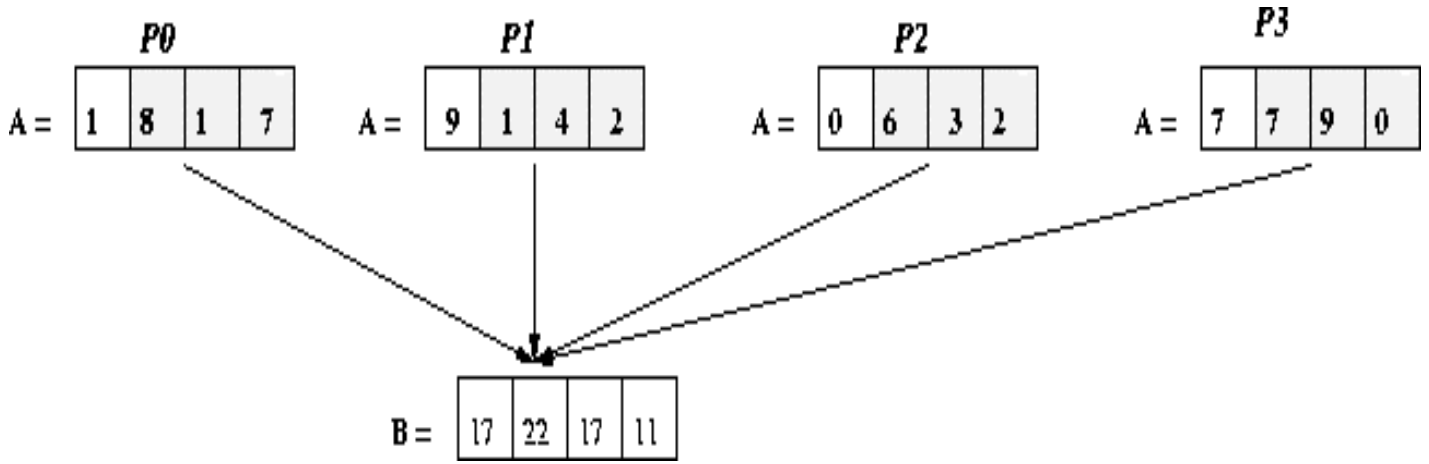
Collective Communication Operations

MPI datatypes for data-pairs used with the MPI_MAXLOC and MPI_MINLOC reduction operations.

MPI Datatype	C Datatype
MPI_2INT	pair of ints
MPI_SHORT_INT	short and int
MPI_LONG_INT	long and int
MPI_LONG_DOUBLE_INT	long double and int
MPI_FLOAT_INT	float and int
MPI_DOUBLE_INT	double and int

Reduction

A reduction compares or computes using a set of data stored on a ll processors and saves the final result on one specified processor.



Global Reduction (sum) of an integer array of size 4 on each processor and accumulate the same on processor P1

MPI_All reduce

MPI provides the MPI_All reduce operation that returns the result to all the processes.

```
int MPI_Allreduce(void *sendbuf, void *recvbuf,
int count,  MPI_Datatype datatype, MPI_Op op,
MPI_Comm comm)
```

Note that there is no target argument since all processes receive the result of the operation.

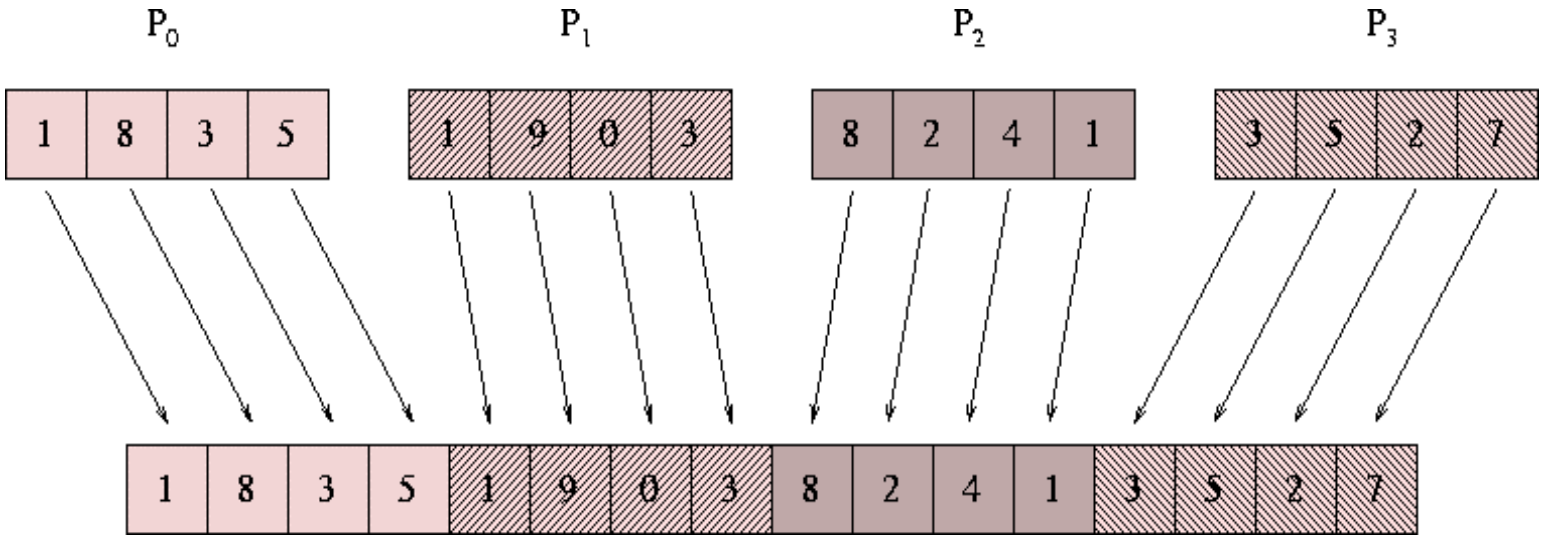
Prefix

int MPI_Scan(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

- MPI_Scan performs a prefix reduction of the data stored in the buffer sendbuf at each process and returns the result in the buffer recvbuf .
- The receive buffer of the process with rank i will store, at the end of the operation, the reduction of the send buffers of the processes whose ranks range from 0 up to and including i .
- The type of supported operations (i.e., op) as well as the restrictions on the various arguments of MPI_Scan are the same as those for the reduction operation MPI_Reduce .

Gather

Accumulate onto a single processor, the data that resides on all processors



Gather an integer array of size of 4 from each processor

Gather

int MPI_Gather(void *sendbuf, int sendcount,
MPI_Datatype senddatatype, void *recvbuf, int recvcount,
MPI_Datatype recvdatatype, int target, MPI_Comm comm)

- Each process, including the target process, sends the data stored in the array sendbuf to the target process.
- As a result, if p is the number of processors in the communication comm, the target process receives a total of p buffers.
- The data is stored in the array recvbuf of the target process, in a rank order.
- That is, the data from process with rank i are stored in the recvbuf starting at location i * sendcount

Gather

MPI also provides the MPI_Allgather function in which the data are gathered to all the processes and not only at the target process.

```
int MPI_Allgather(void *sendbuf, int sendcount,
MPI_Datatype senddatatype, void *recvbuf, int recvcount,
MPI_Datatype recvdatatype, MPI_Comm comm)
```

The meanings of the various parameters are similar to those for MPI_Gather ; however, each process must now supply a recvbuf array that will store the gathered data.

Gather

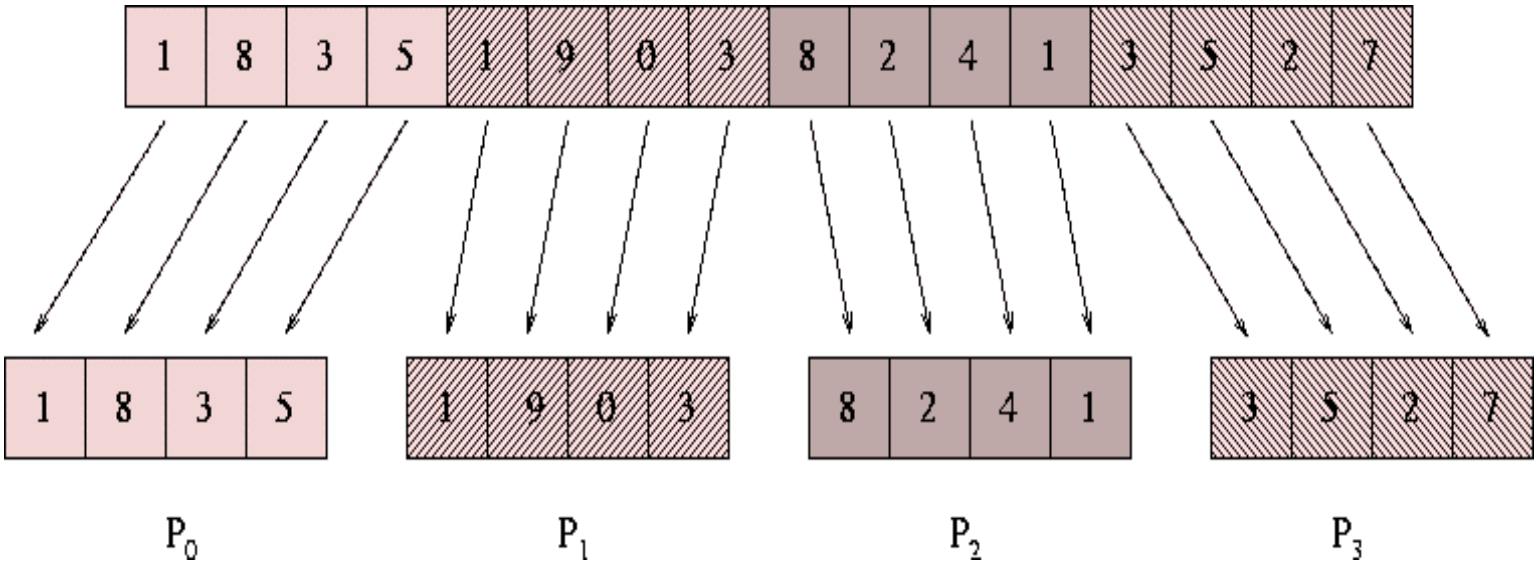
- MPI also provides versions in which the size of the arrays can be different.
- The vector variants of the MPI_Gather and MPI_Allgather operations are provided by the functions MPI_Gatherv and MPI_Allgatherv , respectively.

```
int MPI_Gatherv(void *sendbuf, int sendcount,
MPI_Datatype senddatatype, void *recvbuf, int *recvcounts,
int *displs, MPI_Datatype recvdatatype, int target,
MPI_Comm comm)

 int MPI_Allgatherv(void *sendbuf, int sendcount,
MPI_Datatype senddatatype, void *recvbuf, int *recvcounts,
int *displs, MPI_Datatype recvdatatype, MPI_Comm comm)
```

Scatter

Distribute a set of data from one processor to all other processors.



Scatter an integer array of size 16 on 4 processors

Scatter

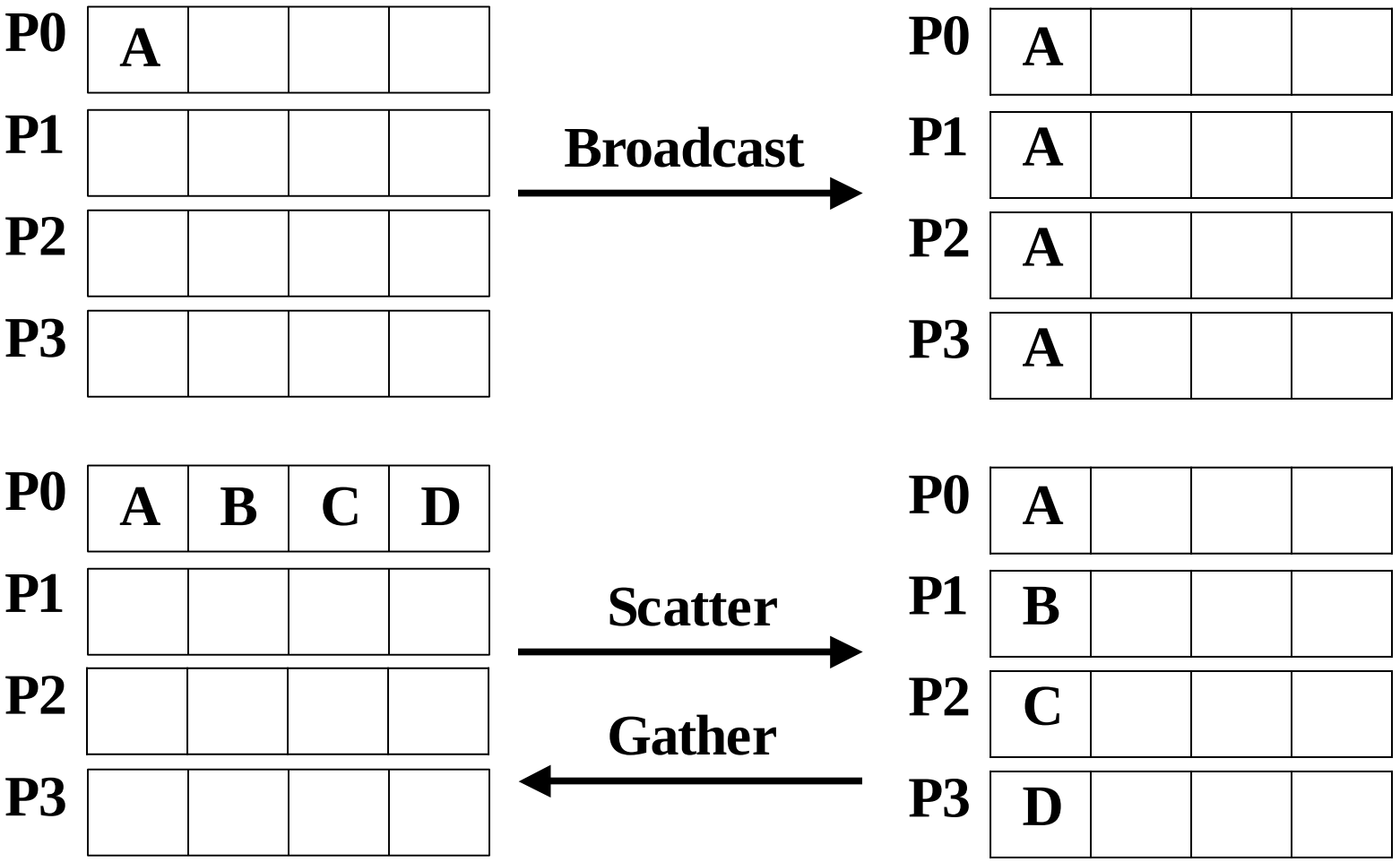
int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype senddatatype, void *recvbuf, int recvcount, MPI_Datatype recvdatatype, int source, MPI_Comm comm)

- The source process sends a different part of the send buffer sendbuf to each processes, including itself.
- The data that are received are stored in recvbuf . Process i receives sendcount contiguous elements of type senddatatype starting from the i * sendcount location of the sendbuf of the source process.
- MPI_Scatter must be called by all the processes with the same values for the sendcount , senddatatype , recvcount , recvdatatype , source , and comm arguments.

Scatter

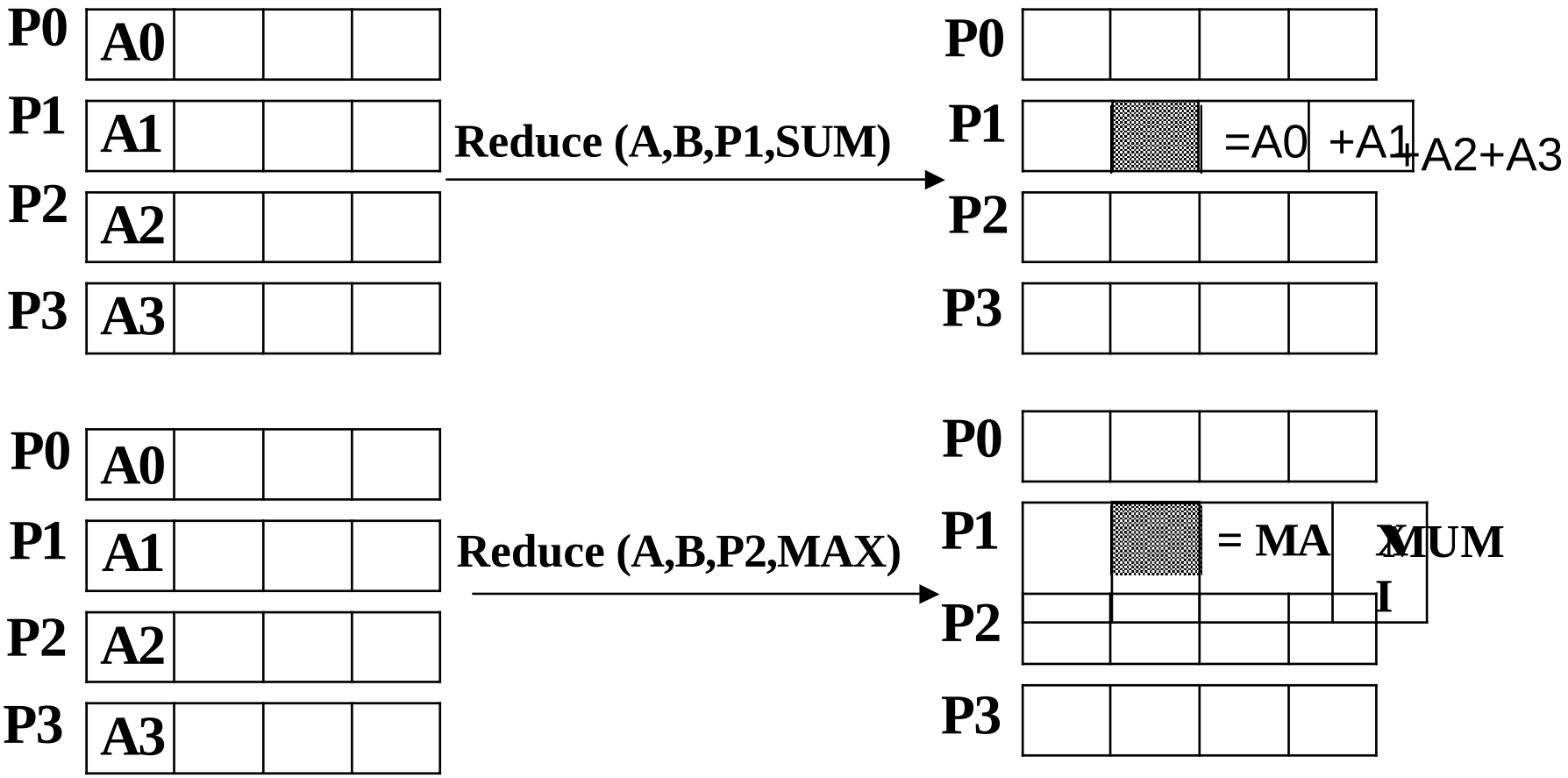
int MPI_Scatterv(void *sendbuf, int *sendcounts, int *displs, MPI_Datatype senddatatype, void *recvbuf, int recvcount, MPI_Datatype recvdatatype, int source, MPI_Comm comm)

MPI Collective Communication



Representation of collective data movement in MPI

MPI Collective Communication



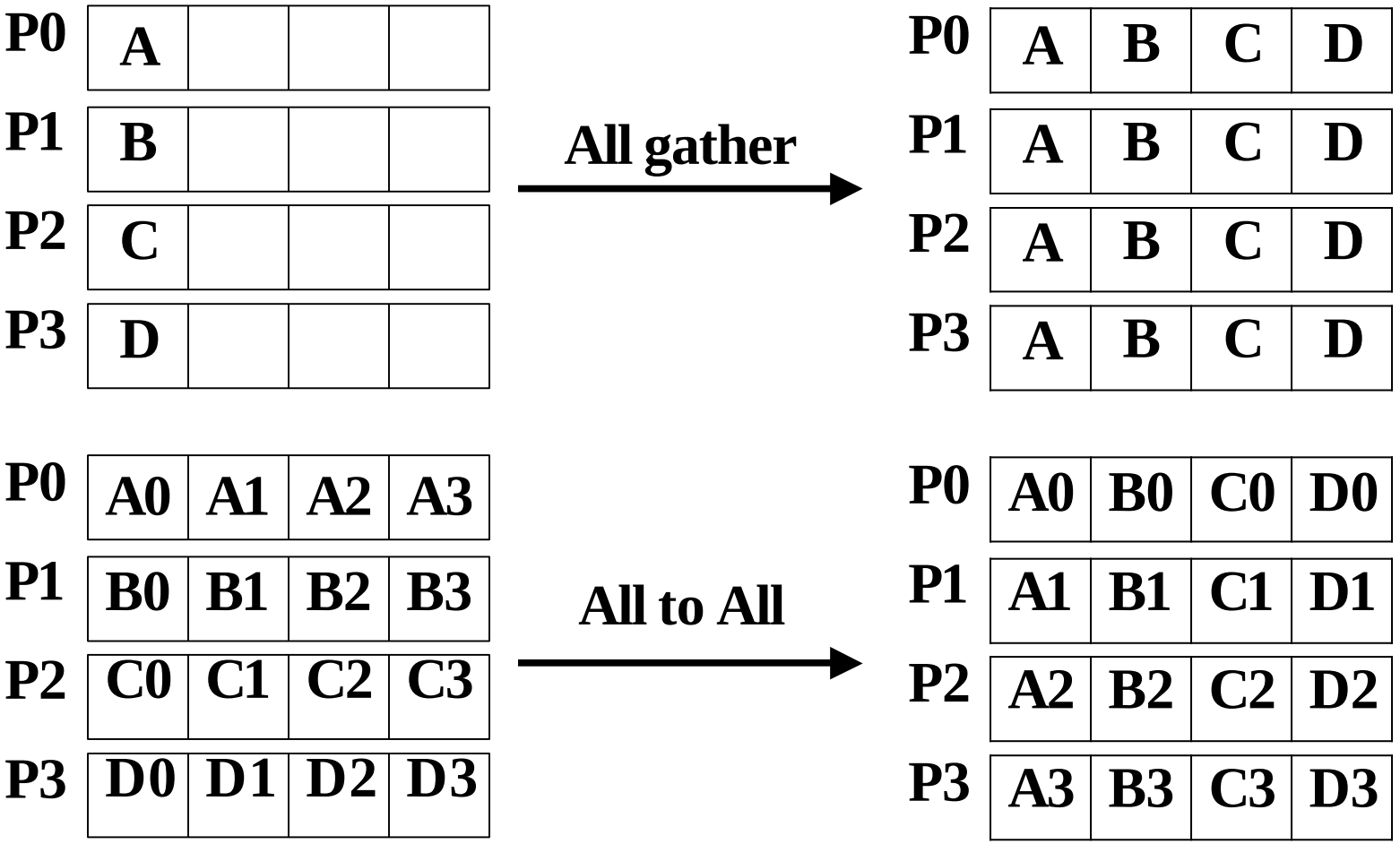
Representation of collective data movement in MPI

MPI Collective Communications & Computations

Allgather	Allgatherv	Allreduce
Alltoall	Alltoallv	Bcast
Gather	Gatherv	Reduce
Reduce Scatter	Scan	Scatter
Scatterv		

- All versions deliver results to all participating processes
- *V* -version allow the chunks to have different non-uniform data sizes (Scatterv, Allgatherv, Gatherv)
- All reduce, Reduce , ReduceScatter, and Scan take both built-in and user-defined combination functions

MPI Collective Communication



Representation of collective data movement in MPI

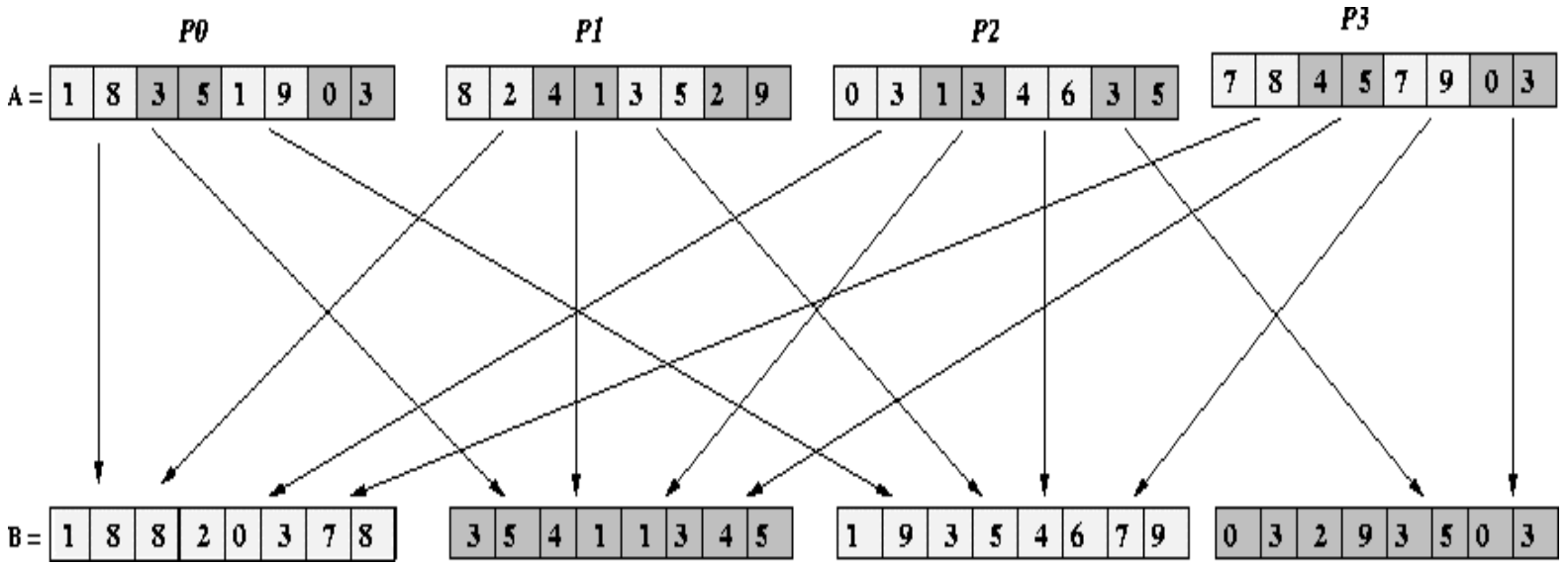
ALL to ALL

int MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype senddatatype, void *recvbuf, int recvcount, MPI_Datatype recvdatatype, MPI_Comm comm)

- Each process sends a different portion of the sendbuf array to each other process, including itself.
- Each process sends to process i sendcount contiguous elements of type senddatatype starting from the i * sendcount location of its sendbuf array.
- The data that are received are stored in the recvbuf array. Each process receives from process i recvcount elements of type recvdatatype and stores them in its recvbuf array starting at location i * recvcount .

All-to-All

Performs a scatter and gather from all four processors to all other four processors. every processor accumulates the final values



All-to-All operation for an integer array of size 8 on 4 processors

Contd..

int MPI_Alltoallv(void *sendbuf, int *sendcounts, int *sdispls MPI_Datatype senddatatype, void *recvbuf, int *recvcounts, int *rdispls, MPI_Datatype recvdatatype, MPI_Comm comm)

- The parameter sendcounts is used to specify the number of elements sent to each process, and the parameter sdispls is used to specify the location in sendbuf in which these elements are stored.
- In particular, each process sends to process i, starting at location sdispls[i] of the array sendbuf, sendcounts[i] contiguous elements.
- The parameter recvcounts is used to specify the number of elements received by each process, and the parameter rdispls is used to specify the location in recvbuf in which these elements are stored.
- In particular, each process receives from process i recvcounts[i] elements that are stored in contiguous locations of recvbuf starting at location rdispls[i]

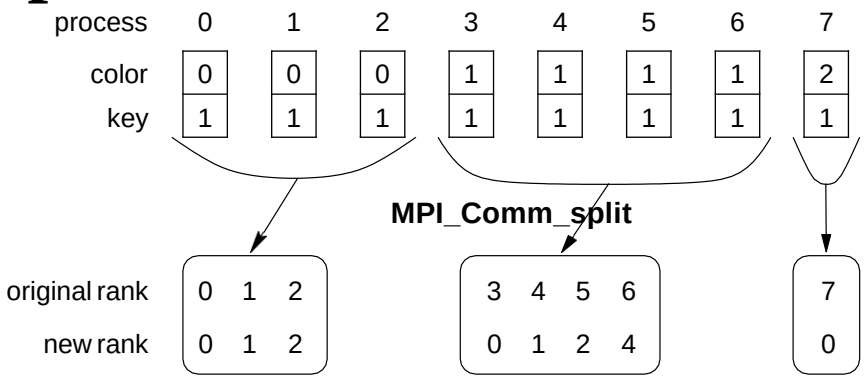
Groups and Communicators

- In many parallel algorithms, communication operations need to be restricted to certain subsets of processes.
- MPI provides mechanisms for partitioning the group of processes that belong to a communicator into subgroups each corresponding to a different communicator.
- The simplest such mechanism is:

```
int MPI_Comm_split(MPI_Comm comm, int color, int key,
        MPI_Comm *newcomm)
```

This operation groups processors by color and sorts resulting groups on the key.

Groups and Communicators



Using `MPI_Comm_split` to split a group of processes in a communicator into subgroups.

Groups and Communicators

- In many parallel algorithms, processes are arranged in a virtual grid, and in different steps of the algorithm, communication needs to be restricted to a different subset of the grid.

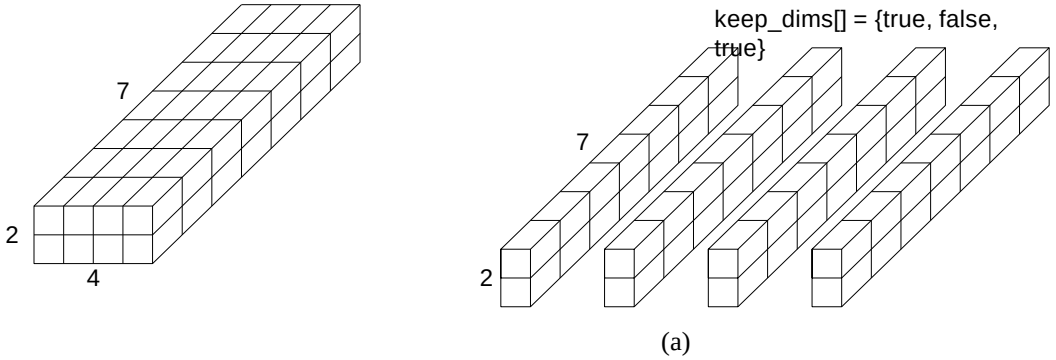
- MPI provides a convenient way to partition a Cartesian topology to form lower-dimensional grids:

```
int MPI_Cart_sub(MPI_Comm comm_cart, int *keep_dims,
        MPI_Comm *comm_subcart)
```

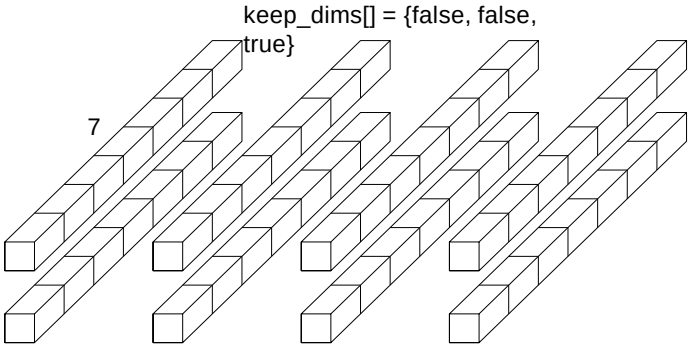
- If `keep_dims[i]` is true (non-zero value in C) then the `i`th dimension is retained in the new sub-topology.

- The coordinate of a process in a sub-topology created by `MPI_Cart_sub` can be obtained from its coordinate in the original topology by disregarding the coordinates that correspond to the dimensions that were not retained.

Groups and Communicators



(a)



(b)

Splitting a Cartesian topology of size 2 × 4 ×7 into (a) four subgroups of size 2 × 1 × 7, and (b) eight subgroups of size 1 × 1 × 7.