MESSAGE PASSING PARADIGM

Dr. Lavika Goel

Principles of Message Passing

- The logical view of a machine supporting the message-passing paradigm consists of p processes, each with its own exclusive address space.
- Each data element must belong to one of the partitions of the space; hence, data must be explicitly partitioned and placed.
- All interactions (read-only or read / write) require cooperation of two processes - the process that has the data and process that wants to access the data.
- These two constraints make underlying costs very explicit to the programmer.

Principles of Message Passing

- Message-passing programs are often written using the *asynchronous* paradigm or *loosely synchronous* paradigm.
- In the *asynchronous* paradigm, all concurrent tasks execute asynchronously.
- In the *loosely synchronous* model, tasks or subsets of tasks synchronize to perform interactions. Between these interactions, tasks execute completely asynchronously.
- Most message-passing programs are written using the *single program multiple data* (SPMD) model.

Send and Receive Operations

- The prototypes of these operations are as follows:

```
send(void *sendbuf, int nelems, int dest)

receive(void *recvbuf, int nelems, int source)
```

- Consider the following code segments:

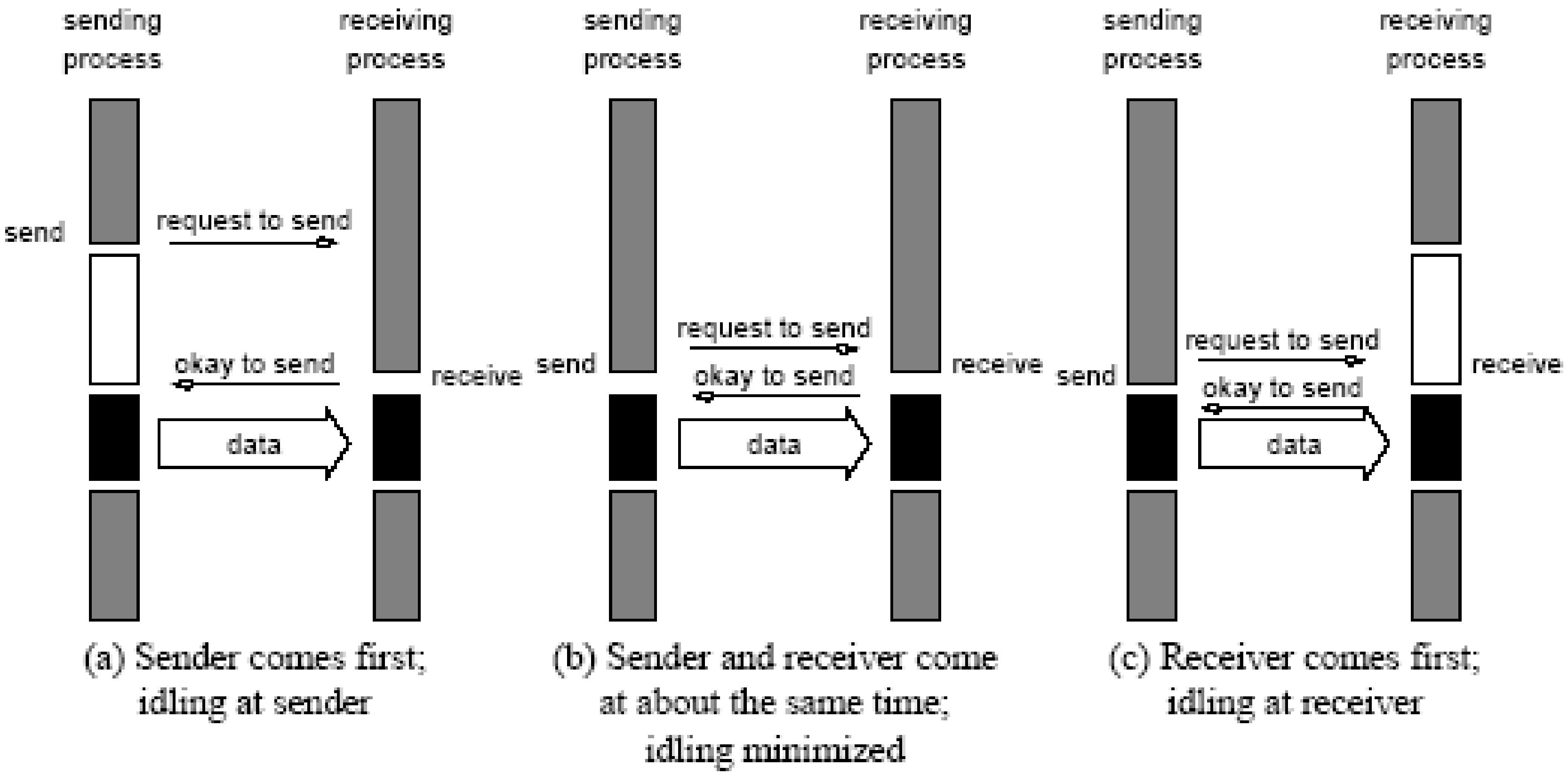
```
P0                  P1
a = 100;            receive(&a, 1, 0)
send(&a, 1, 1);     printf("%d\n", a);
a = 0;
```

- The semantics of the send operation require that the value received by process P1 must be 100, but not 0.
- This motivates the design of the send and receive protocols.

Non Buffered Blocking

- A simple method for forcing send / receive semantics is for the send operation to return only when it is safe to do so.
- In the non-buffered blocking send, the operation does not return until the matching receive has been encountered at the receiving process.
- Idling and deadlocks are major issues with non-buffered blocking sends.

Non Buffered Blocking

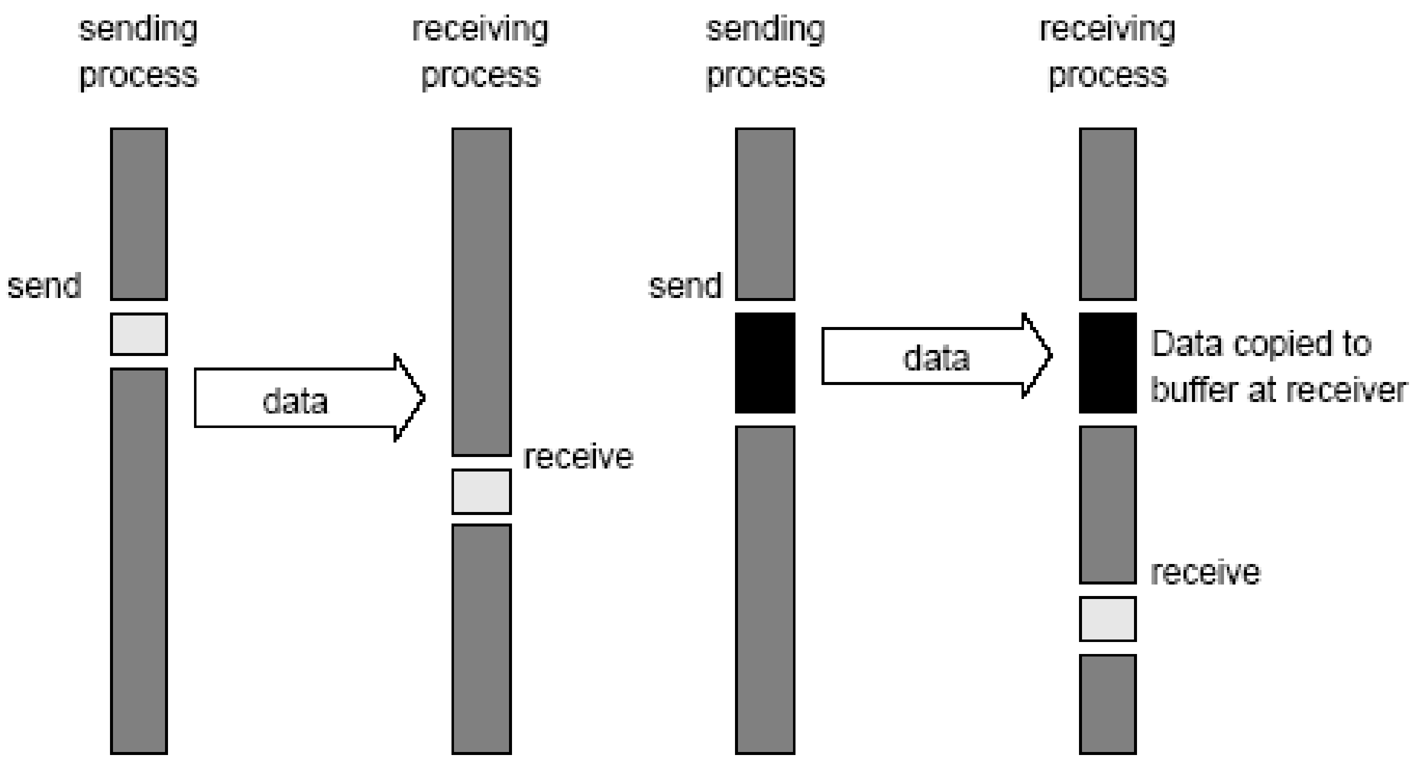


Handshake for a blocking non-buffered send/receive operation.

Buffered Blocking

- A simple solution to the idling and deadlocking problem outlined above is to rely on buffers at the sending and receiving ends.
- The sender simply copies the data into the designated buffer and returns after the copy operation has been completed.
- The data must be buffered at the receiving end as well.
- Buffering trades off idling overhead for buffer copying overhead.

Buffered Blocking



Blocking buffered transfer protocols
(a) in the presence of communication hardware
(b) in the absence of communication hardware

Buffered Blocking

- Bounded buffer sizes can have significant impact on performance.

```
P0
for (i = 0; i < 1000; i++)
{
    produce_data(&a);
    send(&a, 1, 1);
}
```

```
P1
for (i = 0; i < 1000; i++)
{
    receive(&a, 1, 0);
    consume_data(&a);
}
```

- What if consumer was much slower than producer?

Buffered Blocking

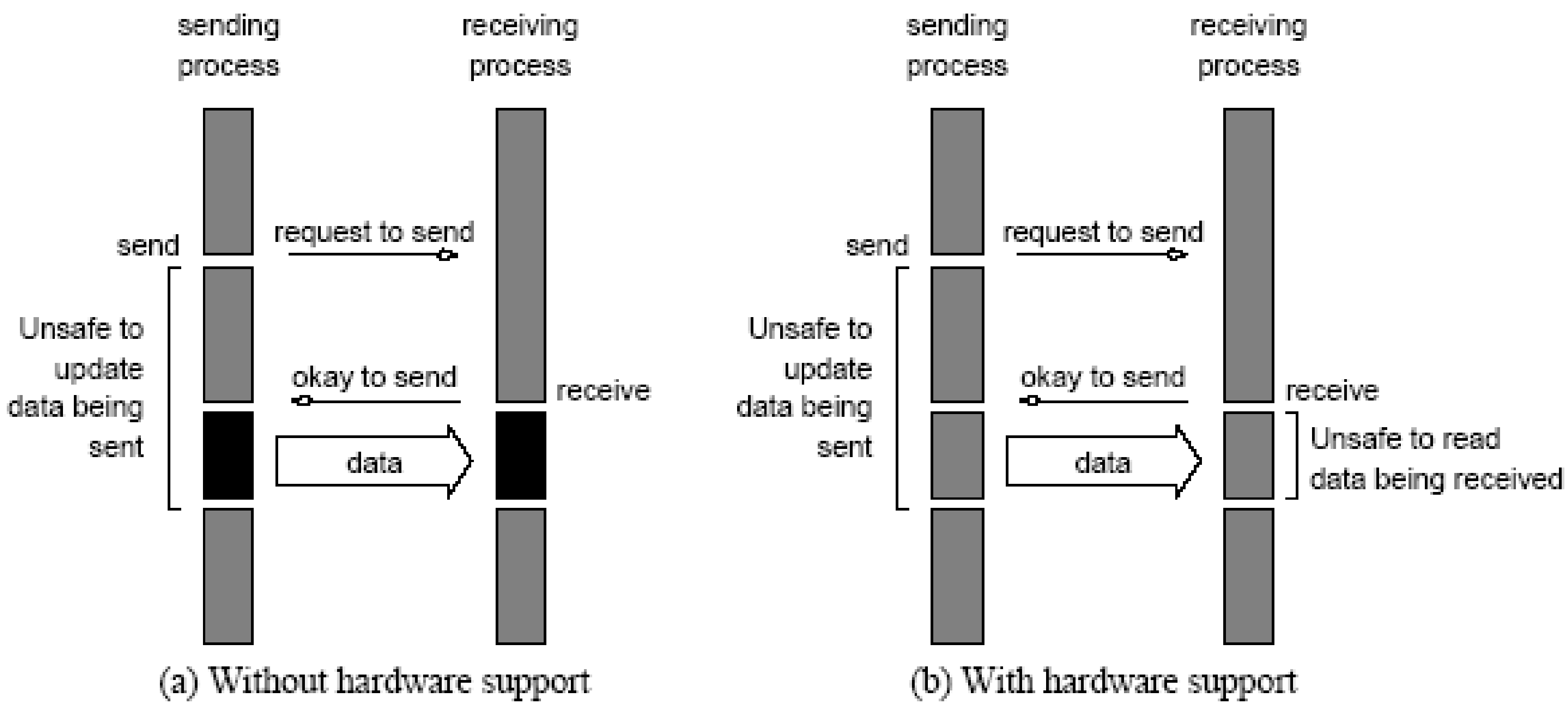
- Deadlocks are still possible with buffering since receive operations block.

```
P0                          P1
receive(&a, 1, 1);          receive(&a, 1, 0);
send(&b, 1, 1);             send(&b, 1, 0);
```

Non Blocking

- The programmer must ensure semantics of the send and receive.
- This class of non-blocking protocols returns from the send or receive operation before it is semantically safe to do so.
- Non-blocking operations are generally accompanied by a check-status operation.
- When used correctly, these primitives are capable of overlapping communication overheads with useful computations.
- Message passing libraries typically provide both blocking and non-blocking primitives.

Non Blocking



Non-blocking non-buffered send and receive operations
(a) In absence of communication hardware;
(b) in presence of communication hardware.

Send and Receive Protocols

	Blocking Operations	Non−Blocking Operations
Buffered	Sending process returns after data has been copied into communication buffer	Sending process returns after initiating DMA transfer to buffer. This operation may not be completed on return
Non−Buffered	Sending process blocks until matching receive operation has been encountered	
	Send and Receive semantics assured by corresponding operation	Programmer must explicitly ensure semantics by polling to verify completion

MPI: Message Passing Interface

- MPI defines a standard library for message-passing that can be used to develop portable message-passing programs using either C or Fortran.
- The MPI standard defines both the syntax as well as the semantics of a core set of library routines.
- Vendor implementations of MPI are available on almost all commercial parallel computers.
- It is possible to write fully-functional message-passing programs by using only the six routines.

What is MPI

- Message Passing Interface
- What is the message?

DATA

- Allows data to be passed between processes in a distributed memory environment

Goals and Scope

- MPI’s prime goals are:
  - To provide source-code portability
  - To allow efficient implementation
- It also offers:
  - A great deal of functionality
  - Support for heterogeneous parallel architectures

MPI Routines

`MPI_Init`	Initializes MPI.
`MPI_Finalize`	Terminates MPI.
`MPI_Comm_size`	Determines the number of processes.
`MPI_Comm_rank`	Determines the label of calling process.
`MPI_Send`	Sends a message.
`MPI_Recv`	Receives a message.

Starting and Terminating

- `MPI_Init` is called prior to any calls to other MPI routines. Its purpose is to initialize the MPI environment.
- `MPI_Finalize` is called at the end of the computation, and it performs various clean-up tasks to terminate the MPI environment.
- The prototypes of these two functions are:

```
int MPI_Init(int *argc, char ***argv)

int MPI_Finalize()
```

Starting and Terminating

- `MPI_Init` also strips off any MPI related command-line arguments.
- All MPI routines, data-types, and constants are prefixed by "`MPI_`". The return code for successful completion is `MPI_SUCCESS`.
- "`mpi.h`" is the header file including all data structures, routines and constants of MPI.

Communicator

- A communicator defines a *communication domain* - a set of processes that are allowed to communicate with each other.
- Information about communication domains is stored in variables of type `MPI_Comm`.
- Communicators are used as arguments to all message transfer MPI routines.
- A process can belong to many different (possibly overlapping) communication domains.
- MPI defines a default communicator called `MPI_COMM_WORLD` which includes all the processes.

Number and Rank of Process

- The `MPI_Comm_size` and `MPI_Comm_rank` functions are used to determine the number of processes and the label of the calling process.
- The calling sequences of these routines are as follows:

```
int MPI_Comm_size(MPI_Comm comm, int *size)
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

- The rank of a process is an integer that ranges from zero up to the size of the communicator minus one.

First MPI Program

```
#include <mpi.h>
main(int argc, char *argv[])
{
      int npes, myrank;

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

      printf("From process %d out of %d,
                 Hello World!\n", myrank, npes);
      MPI_Finalize();
}
```

Sending and Receiving Messages

- The basic functions for sending and receiving messages in MPI are the `MPI_Send` and `MPI_Recv`, respectively.
- The calling sequences of these routines are as follows:

```
int MPI_Send(void *buf, int count,
MPI_Datatype datatype, int dest, int tag,
MPI_Comm comm)


int MPI_Recv(void *buf, int count,
MPI_Datatype datatype, int source, int
tag, MPI_Comm comm, MPI_Status *status)
```

Arguments

`buf`	starting *address* of the data to be sent
`count`	number of elements to be sent
`datatype`	MPI datatype of each element
`dest`	rank of destination process
`tag`	message marker (set by user)
`comm`	MPI communicator of processors involved

```
MPI_SEND(data,500,MPI_REAL,6,33,MPI_COMM_WORLD,IERROR)
```

Sending and Receiving Messages

- MPI provides equivalent datatypes for all C datatypes. This is done for portability reasons.
- The datatype `MPI_BYTE` corresponds to a byte (8 bits) and `MPI_PACKED` corresponds to a collection of data items that has been created by packing non-contiguous data.
- The message-tag can take values ranging from zero up to the MPI defined constant `MPI_TAG_UB`.

MPI Datatypes

MPI Datatype	C Datatype
`MPI_CHAR`	`signed char`
`MPI_SHORT`	`signed short int`
`MPI_INT`	`signed int`
`MPI_LONG`	`signed long int`
`MPI_UNSIGNED_CHAR`	`unsigned char`
`MPI_UNSIGNED_SHORT`	`unsigned short int`
`MPI_UNSIGNED`	`unsigned int`
`MPI_UNSIGNED_LONG`	`unsigned long int`
`MPI_FLOAT`	`float`
`MPI_DOUBLE`	`double`
`MPI_LONG_DOUBLE`	`long double`
`MPI_BYTE`	
`MPI_PACKED`	

Sending and Receiving Messages

- MPI allows specification of wildcard arguments for both source and tag.
- If source is set to `MPI_ANY_SOURCE`, then any process of the communication domain can be the source of the message.
- If tag is set to `MPI_ANY_TAG`, then messages with any tag are accepted.
- On the receive side, the message must be of length equal to or less than the length field specified.

Sending and Receiving Messages

- On the receiving end, the status variable can be used to get information about the `MPI_Recv`.
- The corresponding data structure contains:

```
typedef struct MPI_Status {
  int MPI_SOURCE;
  int MPI_TAG;
  int MPI_ERROR; };
```

- The MPI_Get_count function returns the precise count of data items received.

```
int MPI_Get_count(MPI_Status *status,
MPI_Datatype datatype, int *count)
```

Sample Program

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
/* Run with two processes */
 void main(int argc, char *argv[]) {
    int rank, i, count;
    float data[100],value[200];
    MPI_Status status;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    if(rank==1) {
       for(i=0;i<100;++i) data[i]=i;
       MPI_Send(data,100,MPI_FLOAT,0,55,MPI_COMM_WORLD); }
    else
      {
   MPI_Recv(value,200,MPI_FLOAT,MPI_ANY_SOURCE,55,MPI_COMM_WORLD,&status);
       printf("P:%d Got data from processor %d \n",rank,
                                                status.MPI_SOURCE);
       MPI_Get_count(&status,MPI_FLOAT,&count);
       printf("P:%d Got %d elements \n",rank,count);
       printf("P:%d value[5]=%f \n",rank,value[5]);
    }
    MPI_Finalize();
 }
```

Non-Blocking Communications

- Separate communication into three phases:

1. Initiate non-blocking communication ("post" a send or receive)
2. Do some other work not involving the data in transfer
    - Overlap calculation and communication
    - Latency hiding
3. Wait for non-blocking communication to complete

Non-Blocking Send

```
int MPI_Isend(void *buf,
int count,
MPI_Datatype datatype,
int dest, int tag,
MPI_Comm comm,
MPI_Request *request)
```

Non-Blocking Receive

```
int MPI_Irecv(void *buf,
int count,
MPI_Datatype datatype,
int source, int tag,
MPI_Comm comm,
MPI_Request *request)
```

- There is no status argument

Request Object

- A request object is allocated when a non-blocking communication is initiated
- The request object is used for testing if a specific communication has completed
- It is used as an argument to the `MPI_Test` and the `MPI_Wait` functions to identify the operation whose status we want to query or to wait for its completion.

Completion Tests

- **`wait`** and **`test`**

- **`wait`** - routine does not return until completion finished

- **`test`** - routine returns a TRUE or FALSE value depending on whether or not the communication has completed

Completion Tests

```
int MPI_Wait(
MPI_Request *request,
MPI_Status *status)

int MPI_Test(
MPI_Request *request,
int *flag, MPI_Status *status)
```

- Here is where status appears

Comparison

Blocking:

call **`MPI_RECV (x, N, MPI_Datatype, …, status, …)`**

Non-Blocking:

call **`MPI_IRECV (x, N, MPI_Datatype, …, request, …)`**

… do work that does not involve array x

call **`MPI_WAIT (request, status)`**

… do work that does involve array x

Comparison

Non-Blocking:

call **MPI_IRECV (x,N,MPI_Datatype,…,request,…)**

call **MPI_TEST (request,flag,status,…)**
do while (flag .eq. FALSE)
… work that does not involve the array x …
call **MPI_TEST (request,flag,status,…)**
end do
… do work that does involve the array x …

Derived Datatypes

- MPI allows you to create your own data types analogous to defining structures in a programming language.

- There are two problems with using only basic datatypes:
- MPI communication routines can only send multiples of a single data type: it is not possible to send items of different types, even if they are contiguous in memory.
- It is also ordinarily not possible to send items of one type if they are not contiguous in memory.

Derived Datatypes

- With MPI data types you can solve these problems in several ways.
- You can create a new *contiguous data type* consisting of an array of elements of another data type. There is no essential difference between sending one element of such a type and multiple elements of the component type.
- You can create a *vector data type* consisting of regularly spaced blocks of elements of a component type. This is a first solution to the problem of sending non-contiguous data.

Derived Datatypes

- For not regularly spaced data, there is the *indexed data type* , where you specify an array of index locations for blocks of elements of a component type. The blocks can each be of a different size.
- The *struct data type* can accomodate multiple data types.

Procedure

- ***Construct*** **the new datatype using appropriate MPI routines**

  `MPI_Type_contiguous, MPI_Type_vector, MPI_Type_struct, MPI_Type_indexed`

- ***Commit*** **the new datatype**

  `MPI_Type_Commit`

- ***Use*** **the new datatype in sends / receives, etc.**

Contiguous Datatype

- The simplest derived datatype consists of a number of contiguous items of the same datatype.
- A contigous type describes an array of items of a basic type. There is no difference between sending one item of a contiguous type and multiple items of the constituent type.

A contiguous datatype is built up out of elements of a constituent type

```
int MPI_Type_contiguous (int count,
     MPI_Datatype oldtype,
     MPI_Datatype *newtype)
```

Sample Program

```
#include <stdio.h>
#include <mpi.h>
/* Run with four processes */
 void main(int argc, char *argv[]) {
    int rank;
    MPI_Status status;
    struct {
     int x;      int y;      int z;
    } point;
    MPI_Datatype ptype;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
```

Sample Program

```
    MPI_Type_contiguous(3,MPI_INT,&ptype);
      MPI_Type_commit(&ptype);
      if(rank==3){
         point.x=15; point.y=23; point.z=6;

    MPI_Send(&point,1,ptype,1,52,MPI_COMM_WORLD);
      } else if(rank==1) {

    MPI_Recv(&point,1,ptype,3,52,MPI_COMM_WORLD,&
    status);
         printf("P:%d received coords are
    (%d,%d,%d) \n",rank,point.x,point.y,point.z);
      }
      MPI_Finalize();
  }
```

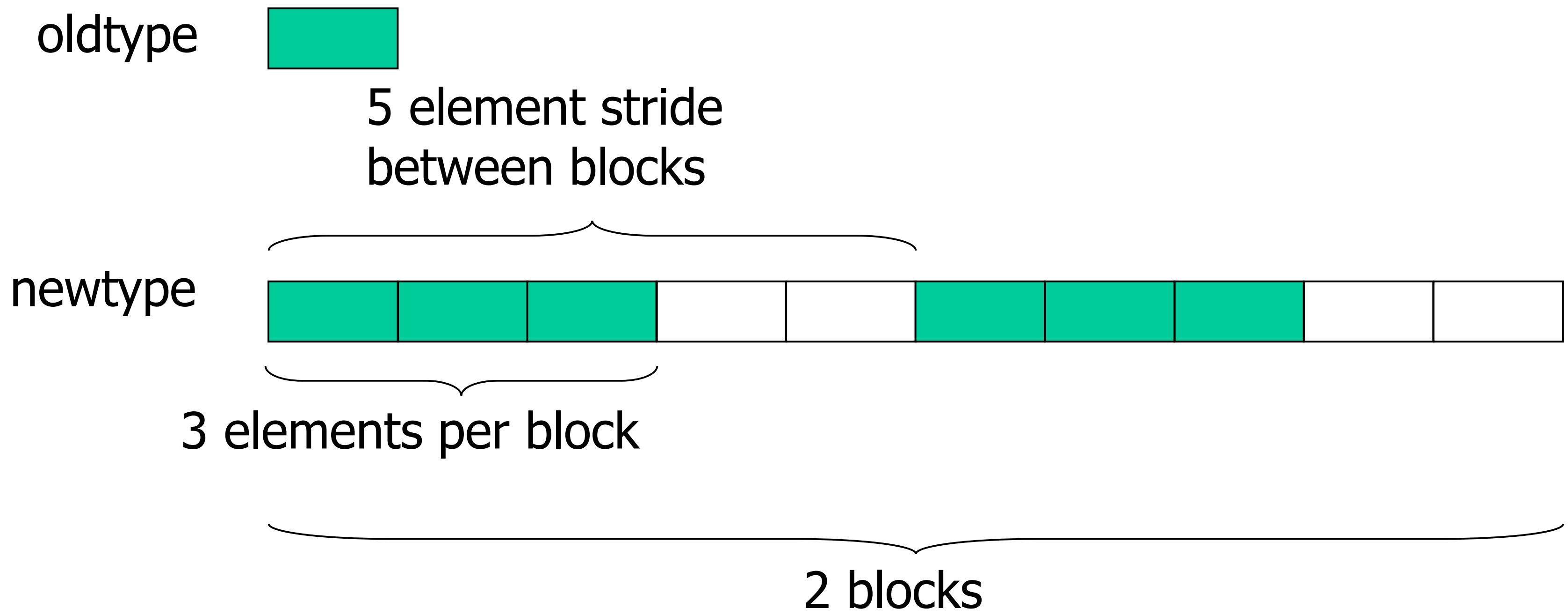
Vector Datatype

- The simplest non-contiguous datatype is the `vector' type.
- A vector type describes a series of blocks, all of equal size, spaced with a constant stride.

```
int MPI_Type_vector(int count,
    int blocklength, int stride,
    MPI_Datatype oldtype,
    MPI_Datatype *newtype)
```

- ***newtype*** has ***count*** blocks each consisting of ***blocklength*** copies of ***oldtype***
- Displacement between blocks is set by ***stride***

Vector Datatype



- count = 2, stride = 5, blocklength = 3

Sample Program

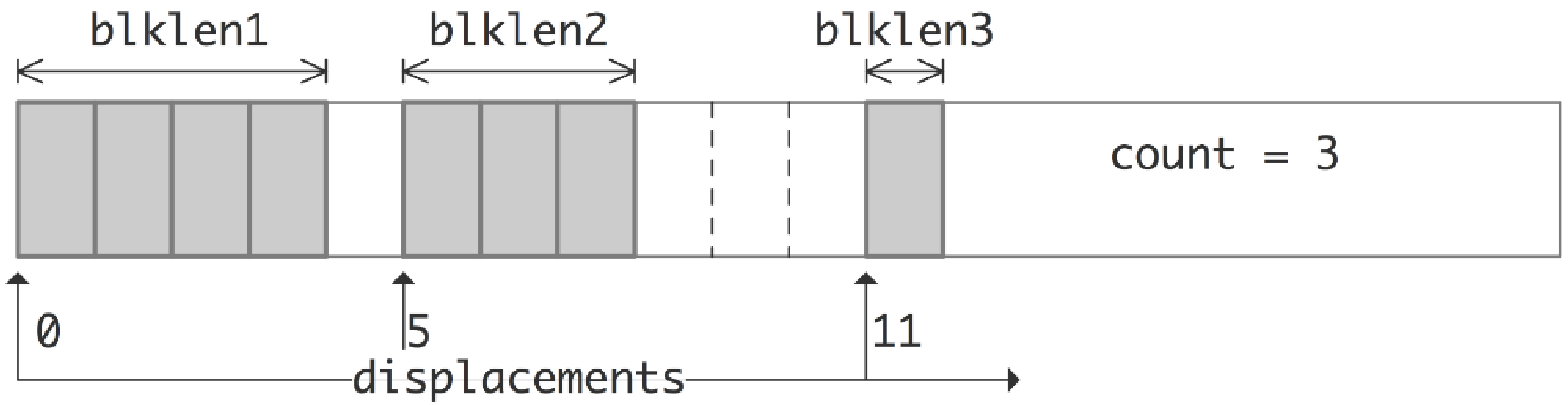
```
#include <mpi.h>
#include <math.h>
#include <stdio.h>
 void main(int argc, char *argv[]) {
    int rank,i,j;
    MPI_Status status;
    double x[4][8];
    MPI_Datatype coltype;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Type_vector(4,1,8,MPI_DOUBLE,&coltype);
    MPI_Type_commit(&coltype);
```

Sample Program

```
  if(rank==3)  {
      for(i=0;i<4;++i)
         for(j=0;j<8;++j) x[i][j]=pow(10.0,i+1)+j;
   MPI_Send(&x[0][7],1,coltype,1,52,MPI_COMM_WORLD);
  }
  else if(rank==1) {
      MPI_Recv(&x[0][2],1,coltype,3,52,
                                    MPI_COMM_WORLD,&status);
       for(i=0;i<4;++i)
       printf("P:%d my x[%d][2]=%1f\n",
                                            rank,i,x[i][2]);
  }
    MPI_Finalize();
}
```

Indexed datatype

- It can send arbitrarily located elements from an array of a single datatype. You need to supply an array of index locations, plus an array of blocklengths with a separate blocklength for each index. The total number of elements sent is the sum of the blocklengths.



The elements of an MPI Indexed datatype

Structure

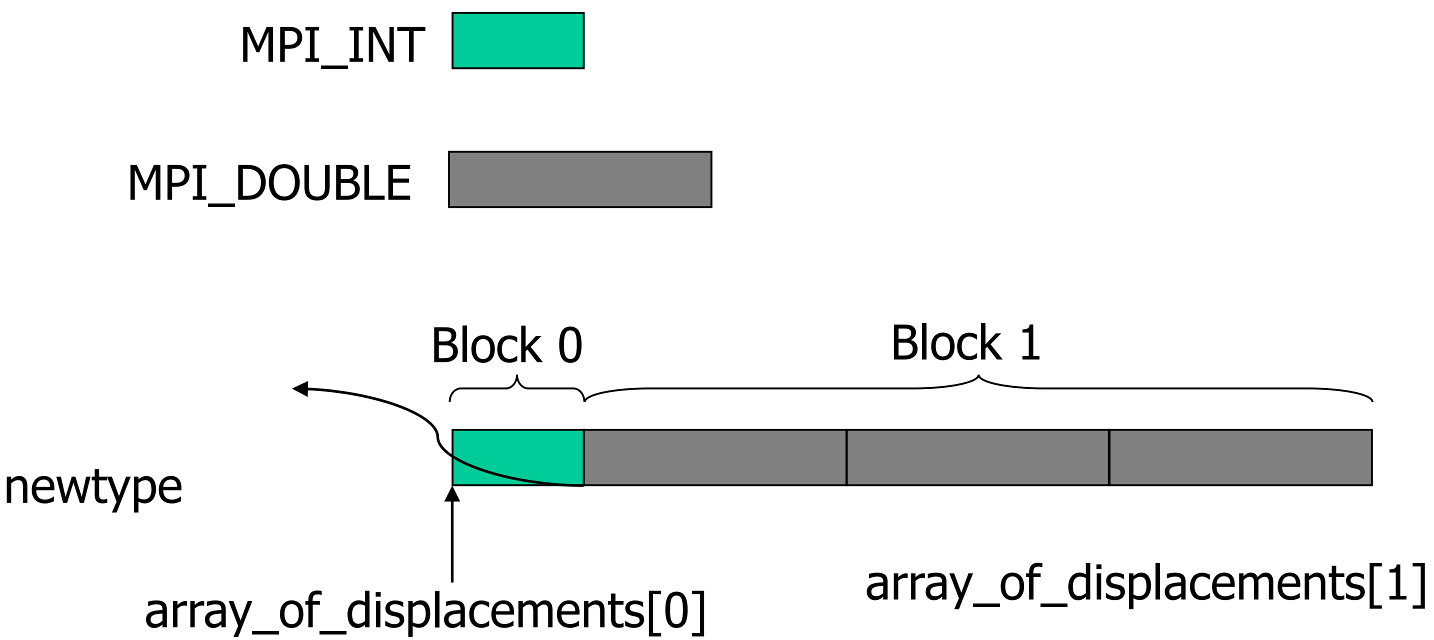
- Use for variables comprised of heterogeneous datatypes
  - C structures
- This is the most general derived data type

```
int MPI_Type_struct (int count,
   int *array_of_blocklengths,
  MPI_Aint *array_of_displacements,
  MPI_Datatype *array_of_types,
  MPI_Datatype *newtype)
```

Structure

- ***`newtype`*** consists of **`count`** blocks where the ith block is **`array_of_blocklengths[i]`** copies of the type **`array_of_types[i]`**.
- The displacement of the `i`th block (in bytes) is given by **`array_of_displacements[i]`**.

Structure Example



- `count = 2, array_of_blocklengths = {1,3}`
- `array_of_types = {MPI_INT, MPI_DOUBLE}`
- `array_of_displacements = {0, extent(MPI_INT)}`

Sample Program

```
#include <stdio.h>
#include<mpi.h>
void main(int argc, char *argv[]) {
 int rank,i;
 MPI_Status status;
 struct {
    int num;
    float x;
    double data[4];
   } a;
   int blocklengths[3]={1,1,4};
   MPI_Datatype types[3] =
               {MPI_INT,MPI_FLOAT,MPI_DOUBLE};
   MPI_Aint displacements[3];
```

Sample Program

```
MPI_Datatype restype;
MPI_Aint intex,floatex;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Type_extent(MPI_INT,&intex);
MPI_Type_extent(MPI_FLOAT,&floatex);
displacements[0]= (MPI_Aint) 0;
displacemtnts[1]=intex;
displacements[2]=intex+floatex;
MPI_Type_struct(3, blocklengths,
     displacements, types, &restype);
 MPI_Type_commit(&restype);
```

Sample Program

```
    if (rank==3){
      a.num=6; a.x=3.14;
       for(i=0;i 4;++i) a.data[i]=(double) i;
        MPI_Send(&a,1,restype,1,52,MPI_COMM_WORLD);
     }
     else if(rank==1)
     {
       MPI_Recv(&a,1,restype, 3, 52,
                  MPI_COMM_WORLD, &status);
       printf("P:%d my a is %d %f %lf %lf %lf %lf\n",
             rank, a.num, a.x,a.data[0], a.data[1],
                              a.data[2], a.data[3]);
     }
     MPI_Finalize();
}
```

Extent

- Handy utility function for datatype construction
- Extent defined to be the memory span (in bytes) of a datatype

```
MPI_Type_extent (MPI_Datatype
datatype,  MPI_Aint* extent)
```

Commit

- Once a datatype has been constructed, it needs to be committed before it is used.
- This is done using **`MPI_TYPE_COMMIT`**

```
int MPI_Type_commit (
      MPI_Datatype *datatype)
```

Collective Communication

- Collective Communications

  Barrier, Broadcast, Scatter, Gather
- Global Reduction Operations

  Reduce, Allreduce, Reduce_scatter, Scan

Collective Communication

- Communications involving a group of processes
- Called by *all* processes in a communicator
- Examples:
  - Broadcast, scatter, gather, etc (Data Distribution)
  - Global sum, global maximum, etc. (Collective Operations)
  - Barrier synchronization

Characteristics of Collective Communication

- Collective communication will not interfere with point-to-point communication and vice-versa
- All processes must call the collective routine
- Synchronization not guaranteed (except for barrier)
- No non-blocking collective communication
- No tags
- Receive buffers must be exactly the right size

Barrier Synchronization

- **Red** light for each processor: turns **green** when all processors have arrived
- Slower than hardware barriers

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

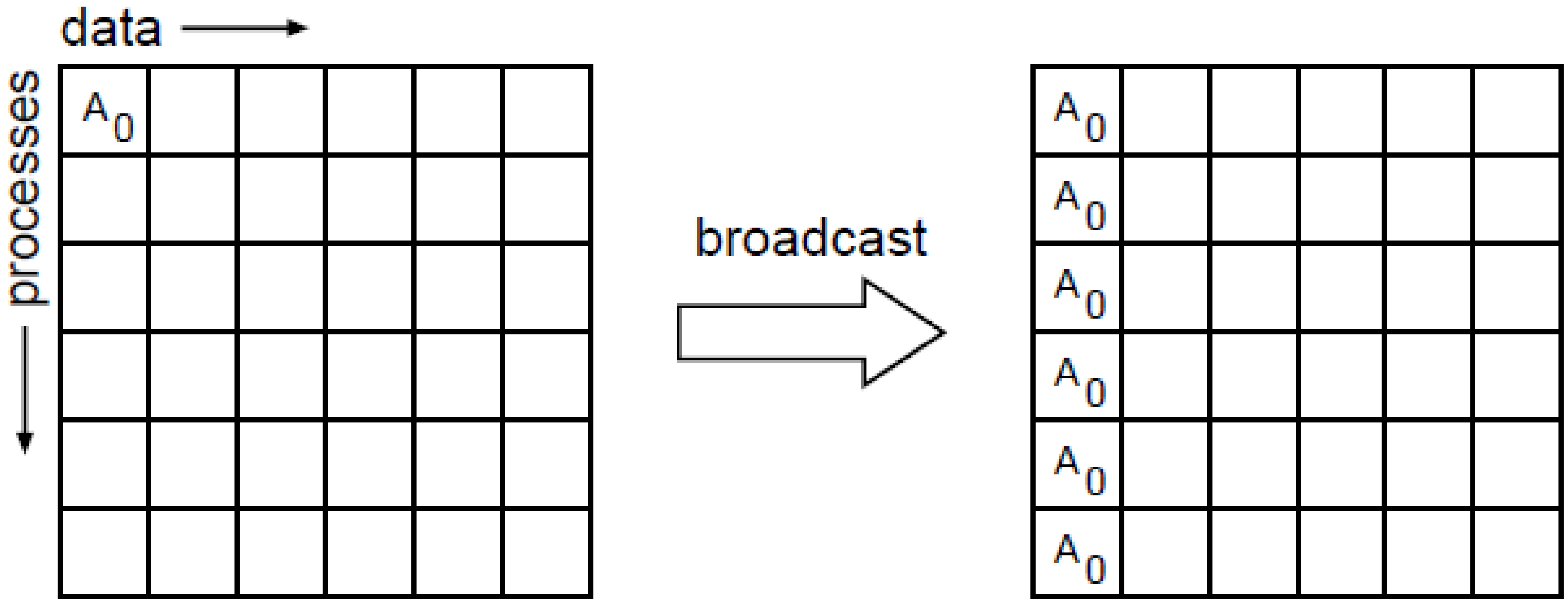
Broadcast

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

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

- All processes must specify same root rank and communicator

Broadcast



Sample Program

```
#include<mpi.h>
void main (int argc, char *argv[])
{
  int rank;
  double param;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  if(rank==5) param=23.0;
     MPI_Bcast(&param,1,MPI_DOUBLE,5,
                              MPI_COMM_WORLD);
  printf("P:%d after broadcast parameter
                         is %f\n",rank,param);
  MPI_Finalize();
}
```

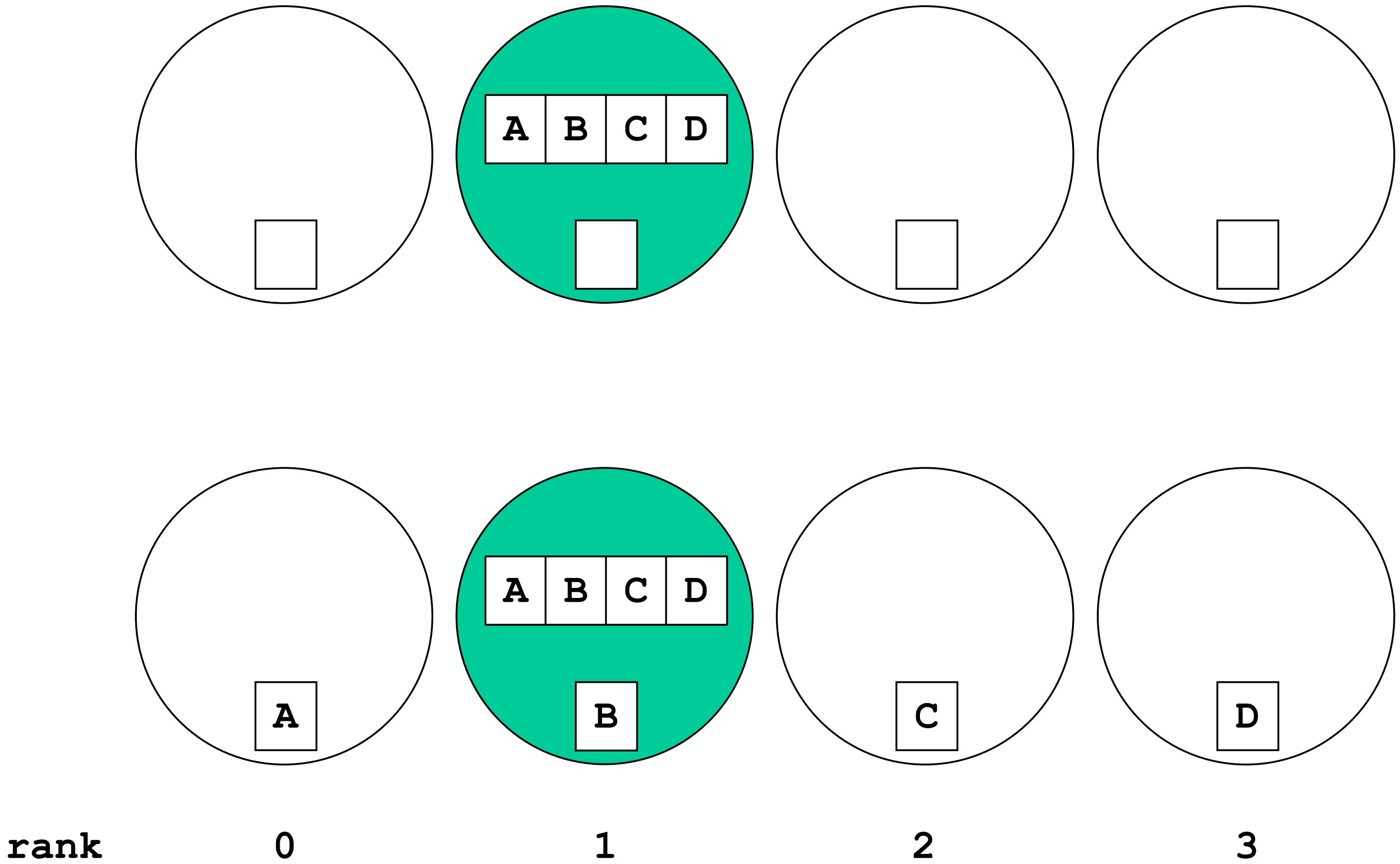
Scatter

- One-to-all communication: different data sent to each process in the communicator (in rank order)

```
int MPI_Scatter(void* sendbuf,
    int sendcount,
    MPI_Datatype sendtype,
    void* recvbuf, int recvcount,
    MPI_Datatype recvtype, int root,
    MPI_Comm comm)
```

- **`sendcount`** is the number of elements sent to each process, not the "total" number sent
  - send arguments are significant only at the root process

Scatter Example



Sample Program

```
#include <mpi.h>
 void main (int argc, char *argv[]) {
    int rank,size,i,j;
    double param[4],mine;
    int sndcnt,revcnt;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    revcnt=1;
    if(rank==3){
       for(i=0;i<4;i++) param[i]=23.0+i;
       sndcnt=1;
    }
    MPI_Scatter(param, sndcnt, MPI_DOUBLE, &mine, revcnt,
                                 MPI_DOUBLE, 3, MPI_COMM_WORLD);
    printf("P:%d mine is %f\n",rank,mine);
    MPI_Finalize();
 }
```

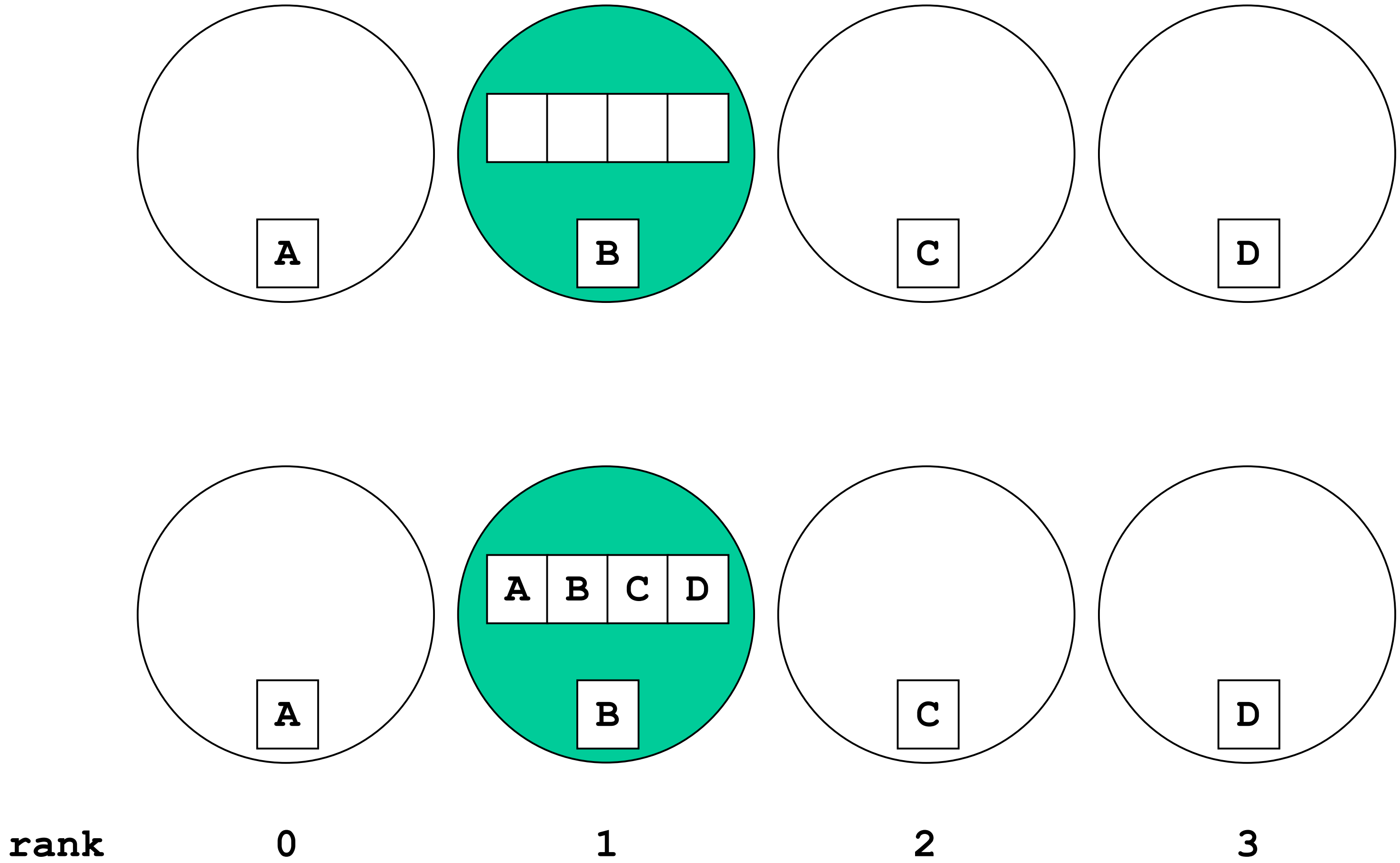
Gather

- All-to-one communication: different data collected by root process
  - Collection done in rank order

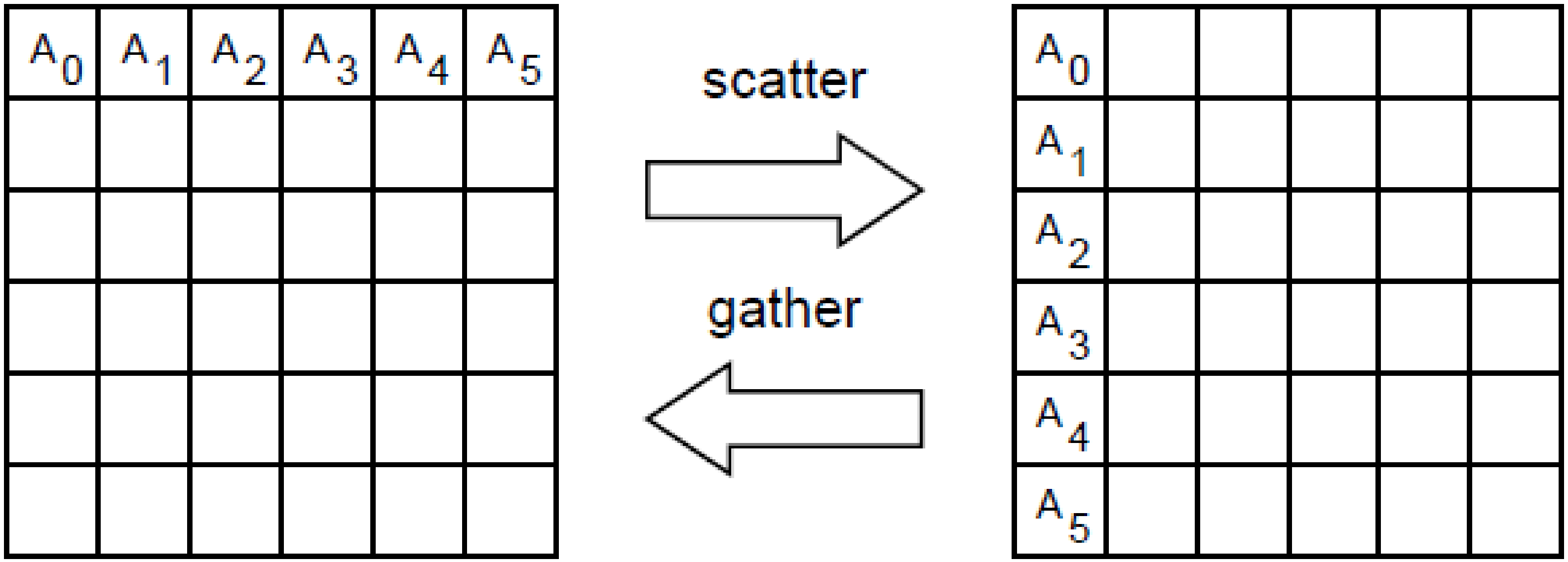
```
int MPI_Gather (void* sendbuf,
    int sendcount,
     MPI_Datatype sendtype,
     void* recvbuf, int recvcount,
    MPI_Datatype recvtype, int root,
    MPI_Comm comm)
```

- Receive arguments only meaningful at the root process

Gather Example



Scatter / Gather



Scatter / Gather Variations

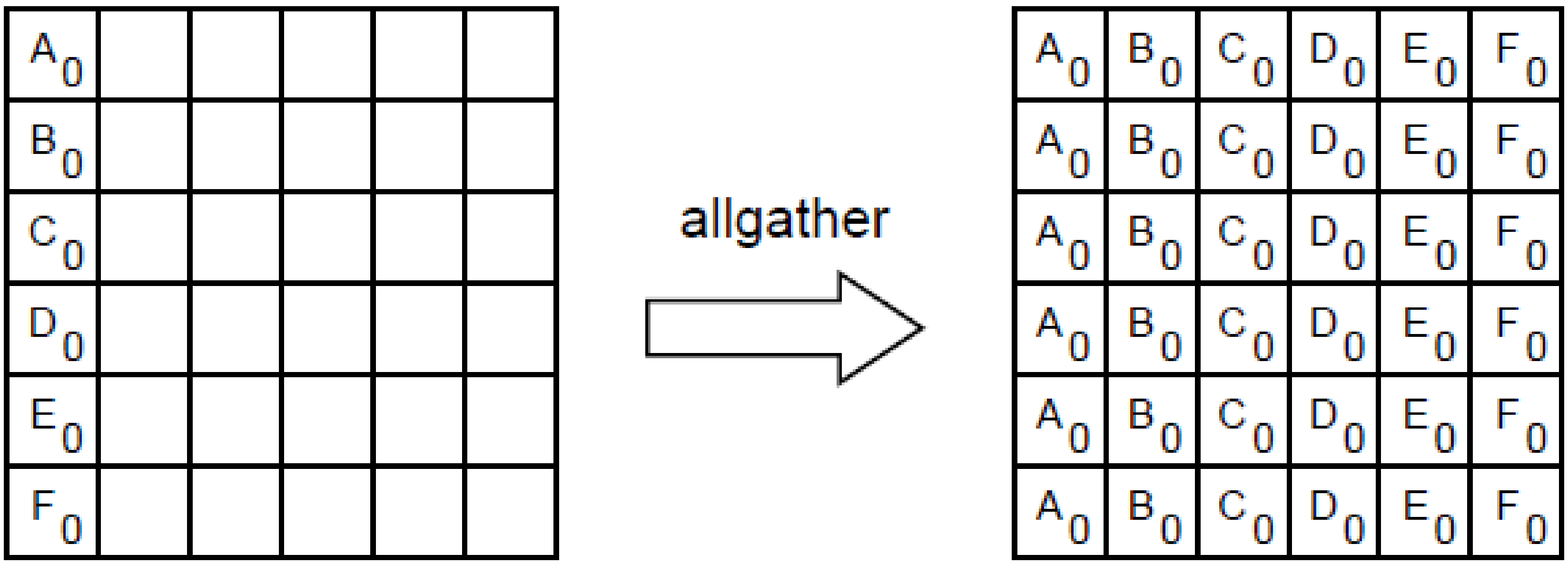
- `MPI_Allgather`
- `MPI_Alltoall`
- No root process specified: all processes get gathered or scattered data
- Send and receive arguments significant for all processes

Scatter / Gather Variations

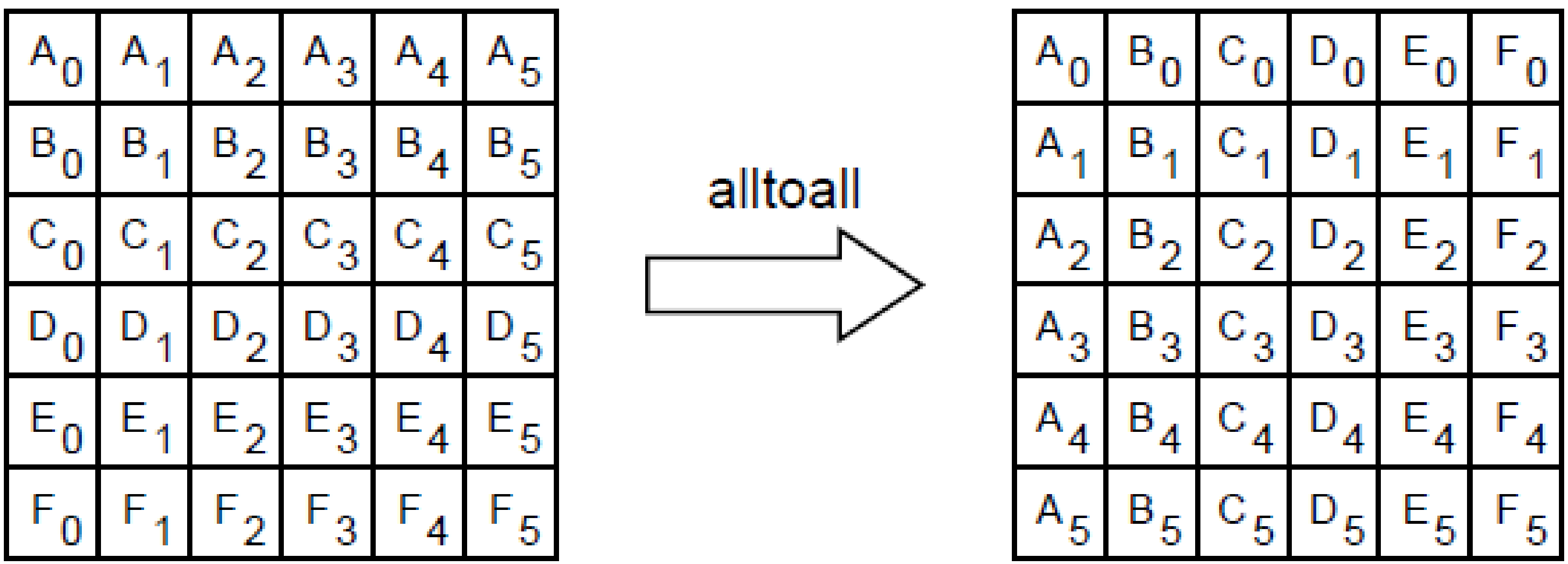
```
int MPI_Allgather (void* sendbuf,
     int sendcount,
     MPI_Datatype sendtype,
     void* recvbuf, int recvcount,
     MPI_Datatype recvtype,
     MPI_Comm comm)

int MPI_Alltoall (void* sendbuf,
     int sendcount,
     MPI_Datatype sendtype,
     void* recvbuf, int recvcount,
     MPI_Datatype recvtype,
     MPI_Comm comm)
```

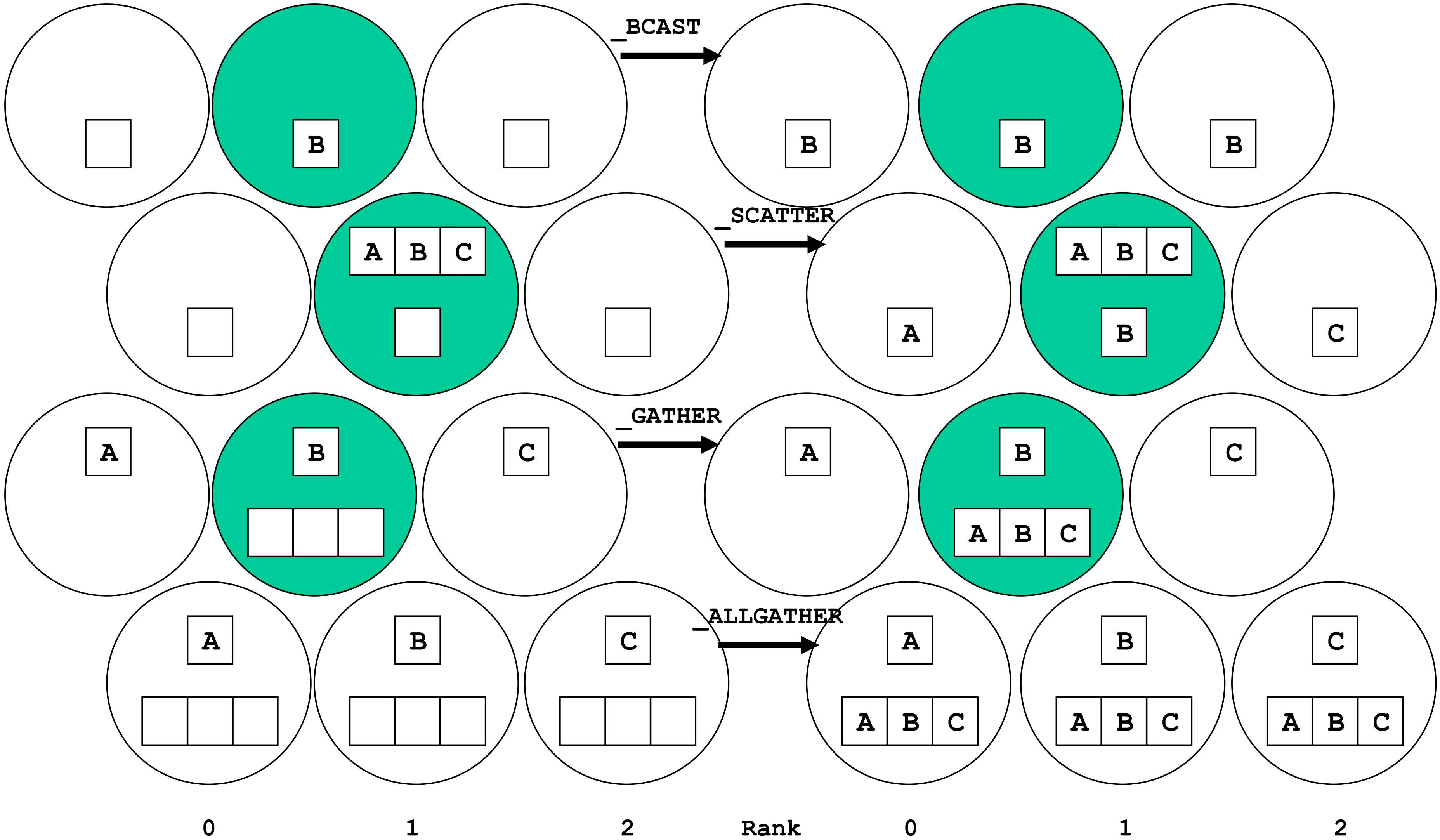
Scatter / Gather Variations



Scatter / Gather Variations



Summary



_BCAST
B
B
B
B
_SCATTER
A
B
C
A
B
C
A
B
C
_GATHER
A
B
C
A
B
C
A
B
C
_ALLGATHER
A
B
C
A
A
B
C
B
A
B
C
C
A
B
C
0
1
2
Rank
0
1
2

Summary

- Root sends data to all processes (itself included): `Broadcast` and `Scatter`
- Root receives data from all processes (itself included): `Gather`
- Each process will communicate with each process (itself included): `Allgather` and `Alltoall`

Global Reduction Operations

- Used to compute a result involving data distributed over a group of processes
- Perform a global reduce operation such as sum, max, logical AND, etc across all the members of a group
- The reduction operation can be either one of a predefined list of operations or a user-defined operation

Global Reduction Operations

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

- **`count`** is the number of "*ops*" done on consecutive elements of **`sendbuf`** (it is also size of **`recvbuf`**)
- **`op`** is an associative operator that takes two operands of type **`datatype`** and returns a result of the same type

Global Reduction Operations

- The global reduction functions come in several flavors
  - a reduce that returns the result of the reduction at one node
  - an allreduce that returns this result at all nodes
  - a scan parallel prefix operation
- A reduce-scatter operation combines the functionality of a reduce and of a scatter operation

Example – Global Sum

- Sum of all the **`x`** values is placed in **`result`** only on processor 0

```
MPI_Reduce(&x,&result,1,
  MPI_INTEGER, MPI_SUM, 0,
  MPI_COMM_WORLD)
```

Predefined Reduction Operations

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 and location
MPI_MINLOC	Minimum and location

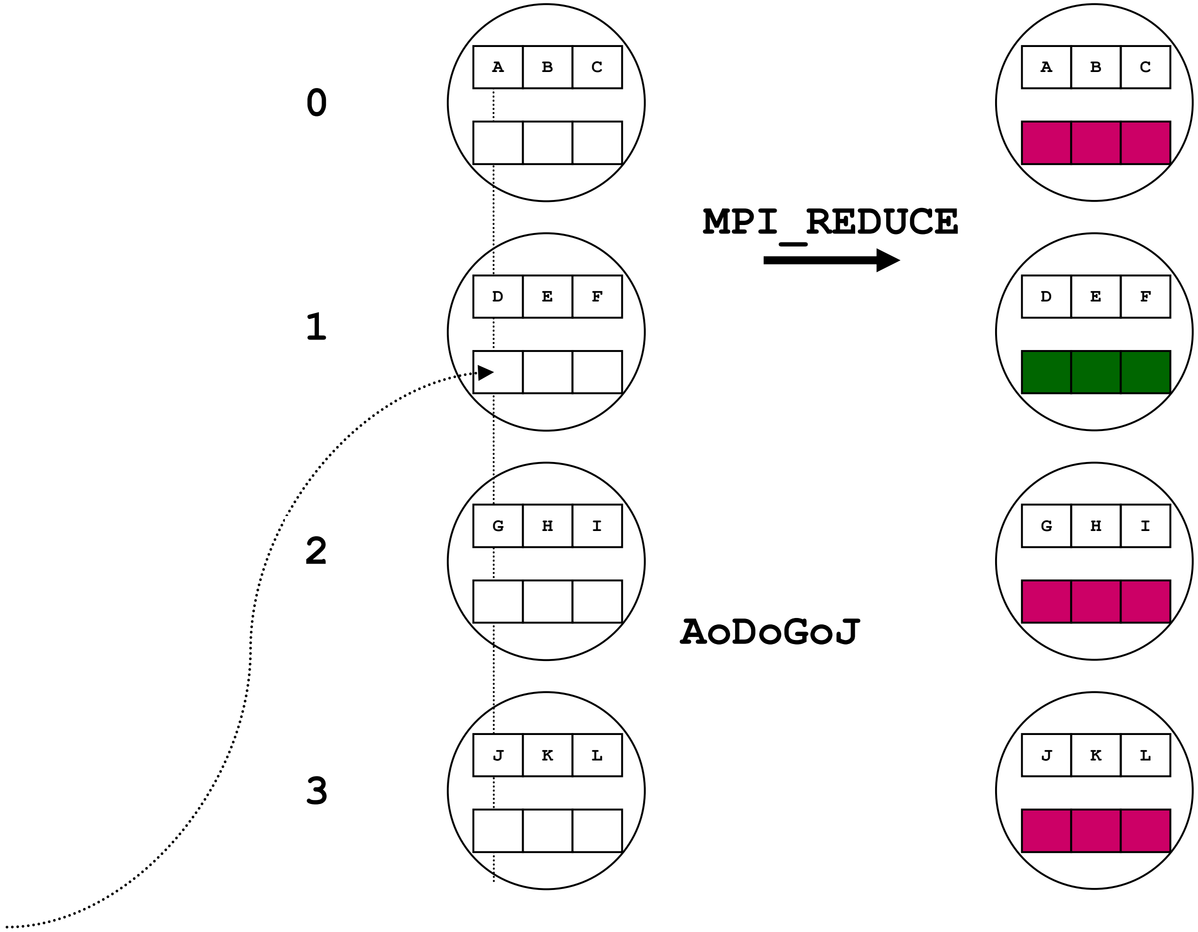
Sample Program

```
#include <mpi.h>
/* Run with 16 processes */
void main (int argc, char *argv[])
{
   int rank;
   struct {
     double value;
     int rank;
   } in, out;
   int root;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   in.value=rank+1;
   in.rank=rank;
   root=7;
   MPI_Reduce(&in, &out, 1, MPI_DOUBLE_INT, MPI_MAXLOC, root,
                                                        MPI_COMM_WORLD);
   if(rank==root) printf("PE:%d max=%lf at rank %d\n", rank,
                                                   out.value, out.rank);
   MPI_Reduce(&in, &out, 1, MPI_DOUBLE_INT, MPI_MINLOC, root,
                                                        MPI_COMM_WORLD);
   if(rank==root) printf("PE:%d min=%lf at rank %d\n", rank,
                                                   out.value, out.rank);
   MPI_Finalize();
}
```

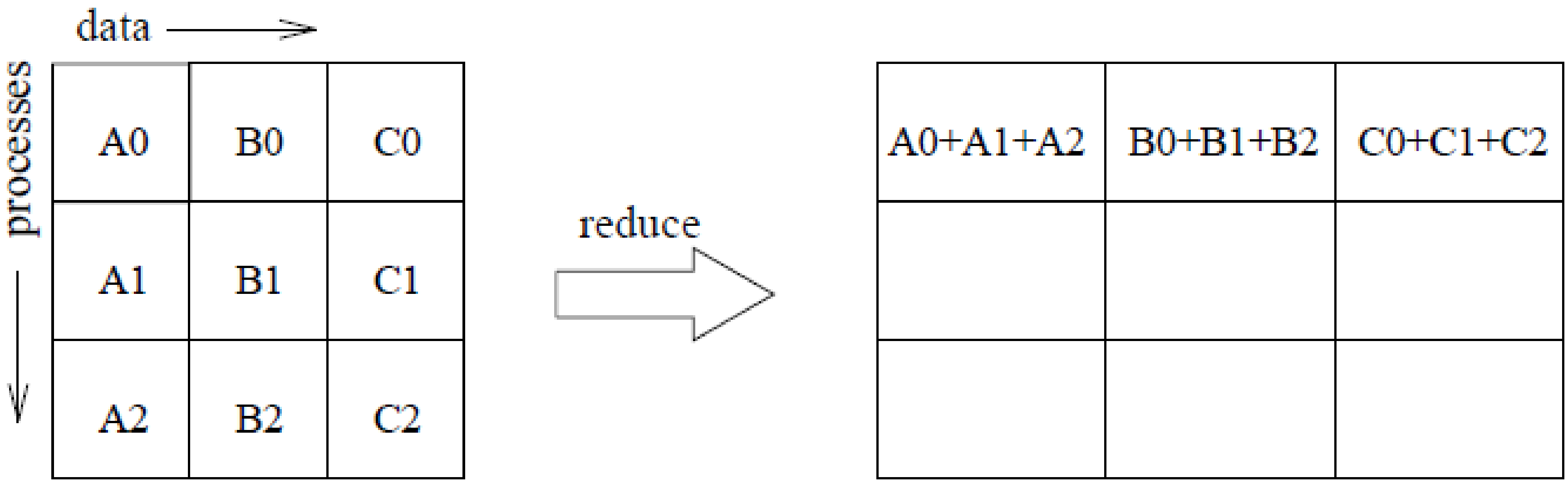
Variations of Reduce

- `MPI_Allreduce` -- no root process (all get results)
- `MPI_Reduce_scatter` -- multiple results are scattered
- `MPI_Scan` -- “parallel prefix”

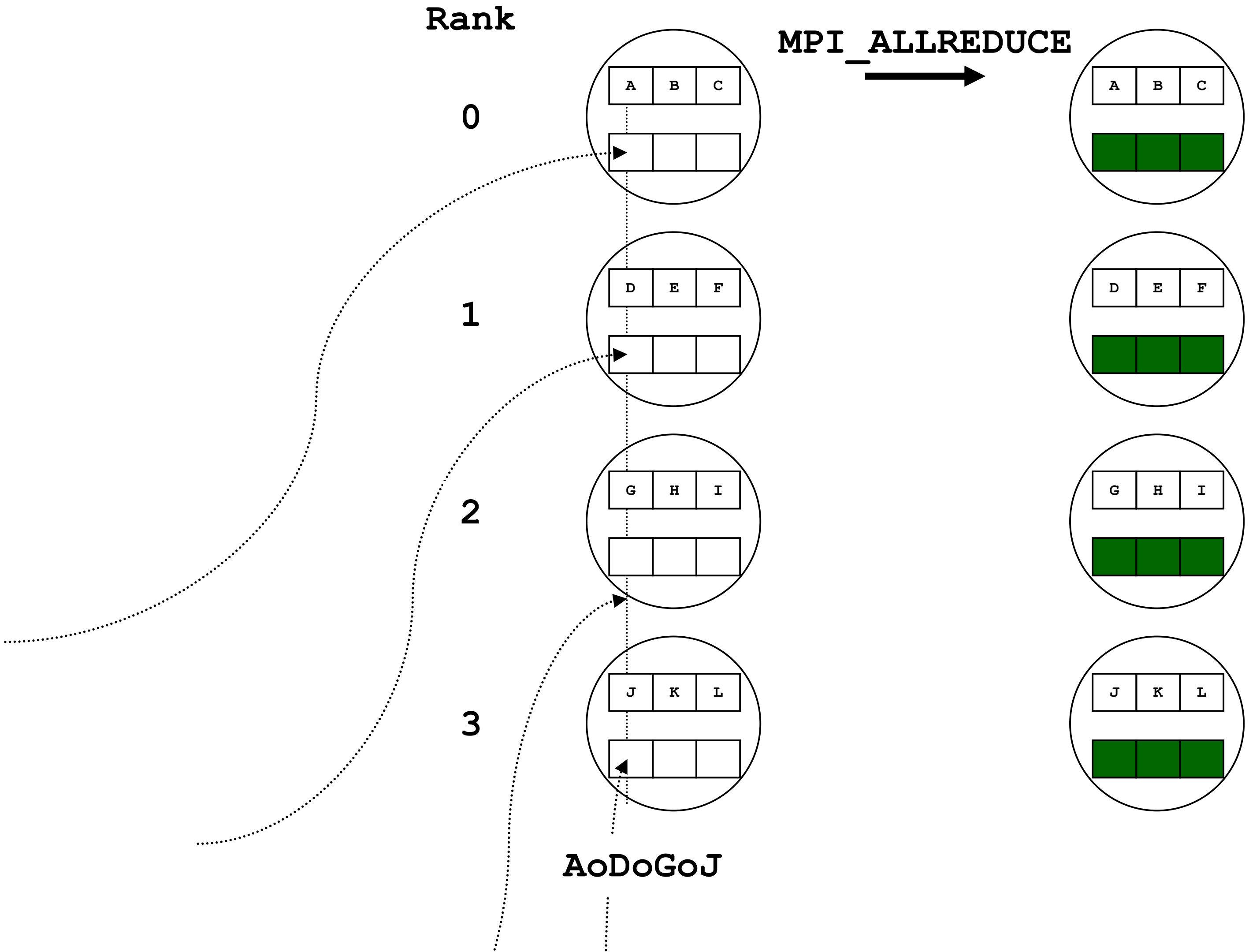
MPI_Reduce



MPI_Reduce


data
processes
A0
B0
C0
A1
B1
C1
A2
B2
C2
reduce
A0+A1+A2
B0+B1+B2
C0+C1+C2

MPI_Allreduce



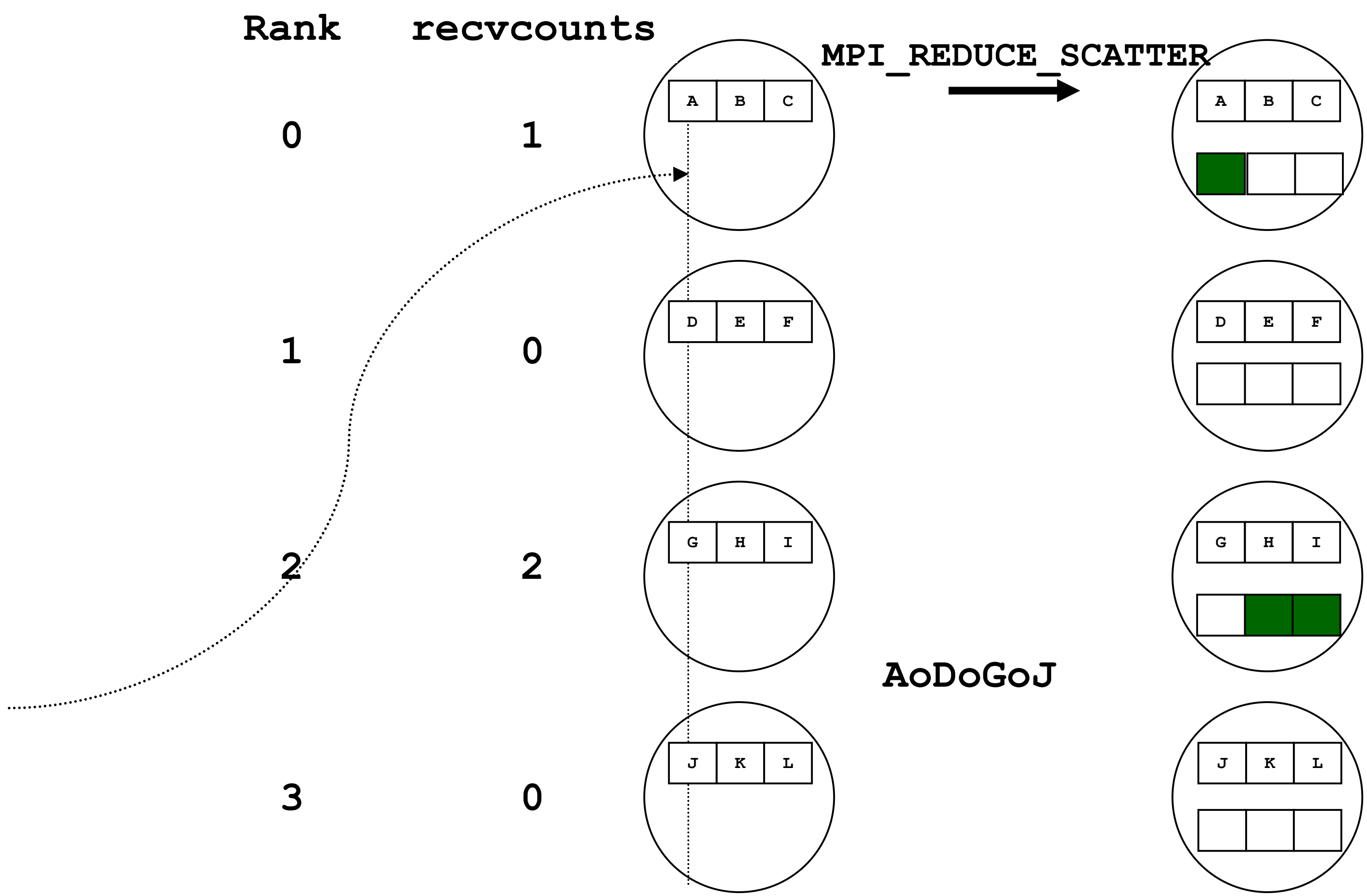
MPI_Allreduce

A0	B0	C0
A1	B1	C1
A2	B2	C2

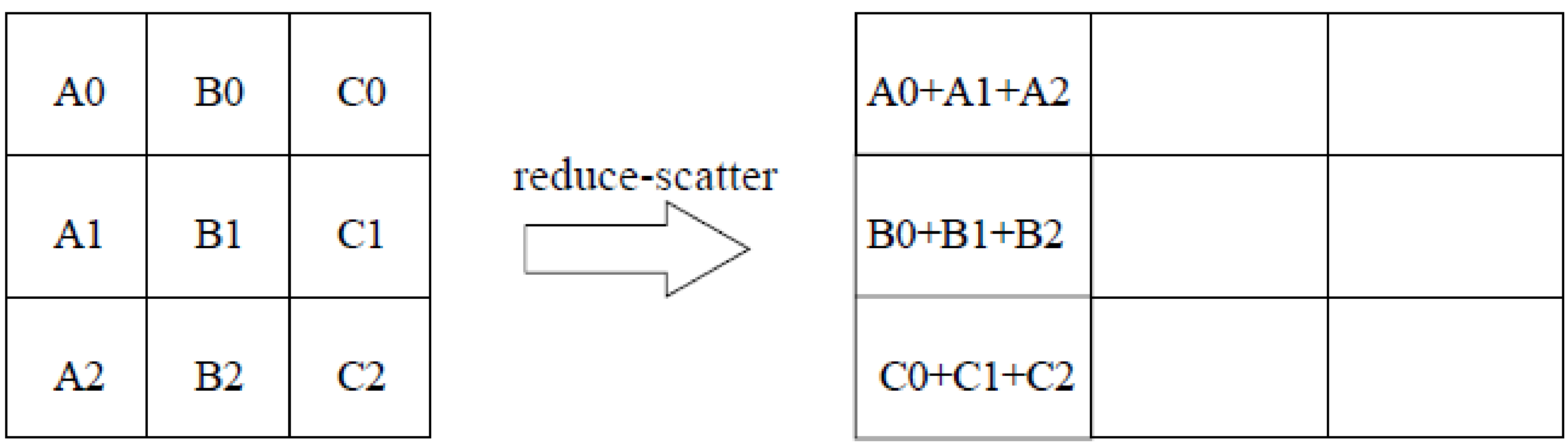
allreduce

A0+A1+A2	B0+B1+B2	C0+C1+C2
A0+A1+A2	B0+B1+B2	C0+C1+C2
A0+A1+A2	B0+B1+B2	C0+C1+C2

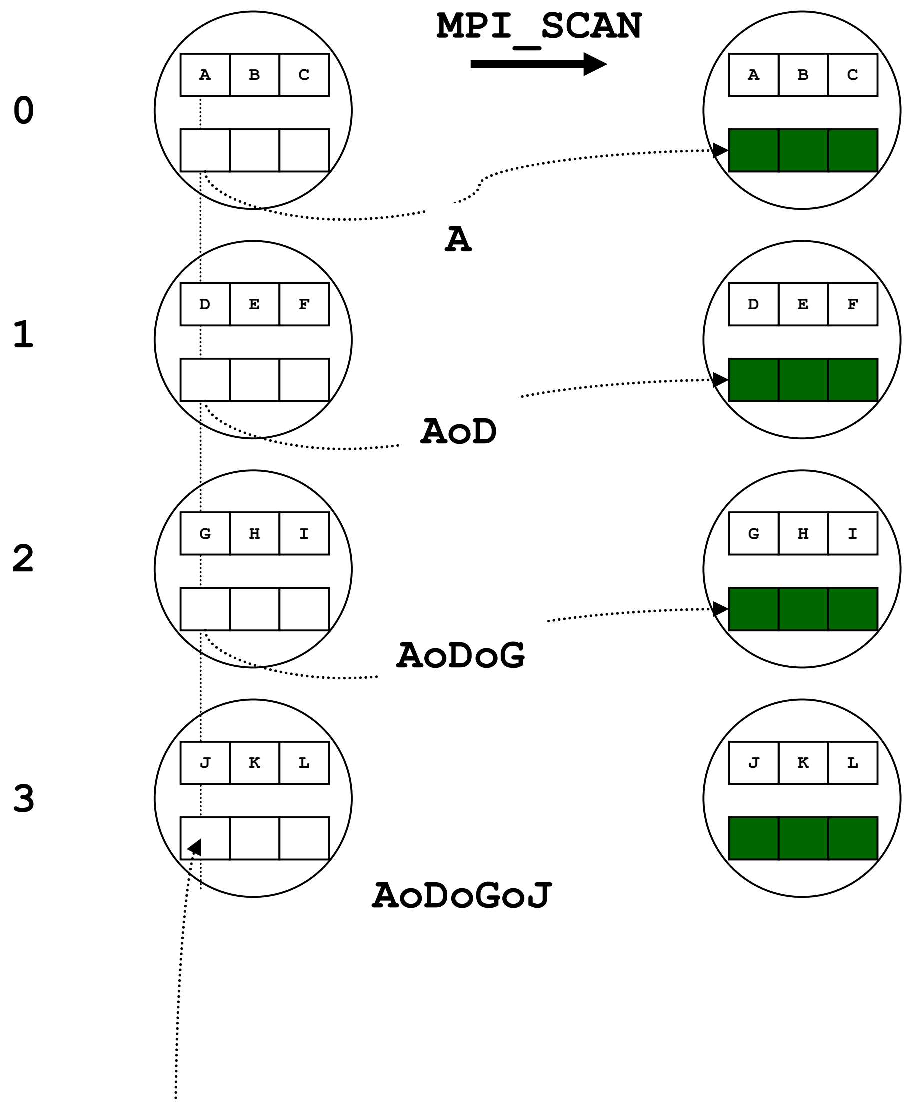
MPI_Reduce_scatter



MPI_Reduce_scatter



MPI_Scan



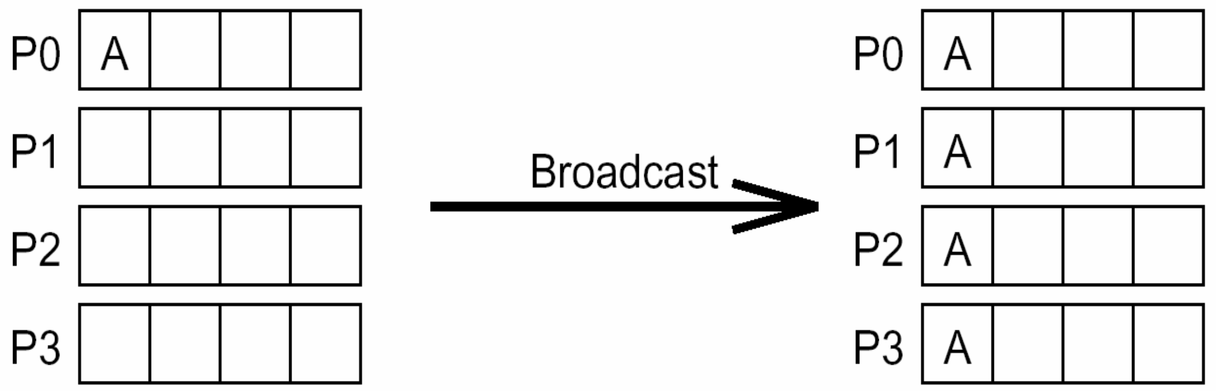
MPI_Scan

A0	B0	C0
A1	B1	C1
A2	B2	C2

scan

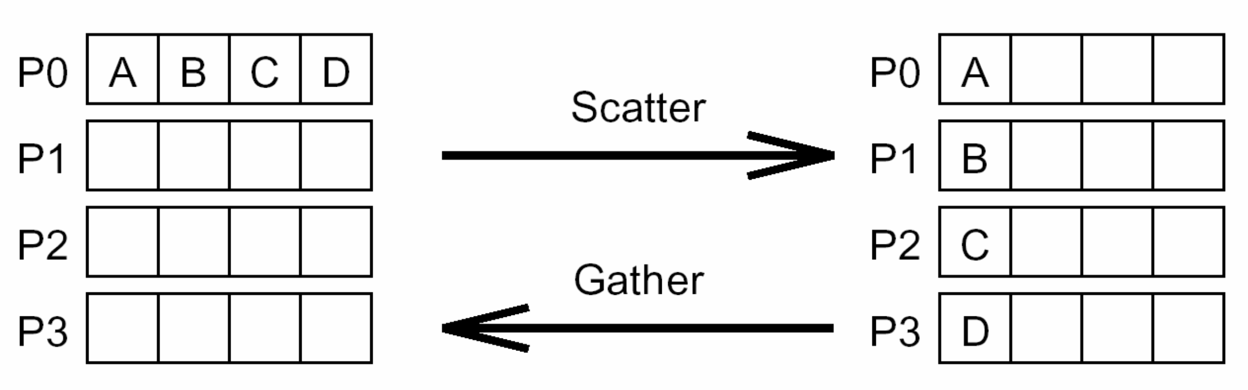
A0	B0	C0
A0+A1	B0+B1	C0+C1
A0+A1+A2	B0+B1+B2	C0+C1+C2

Revision




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

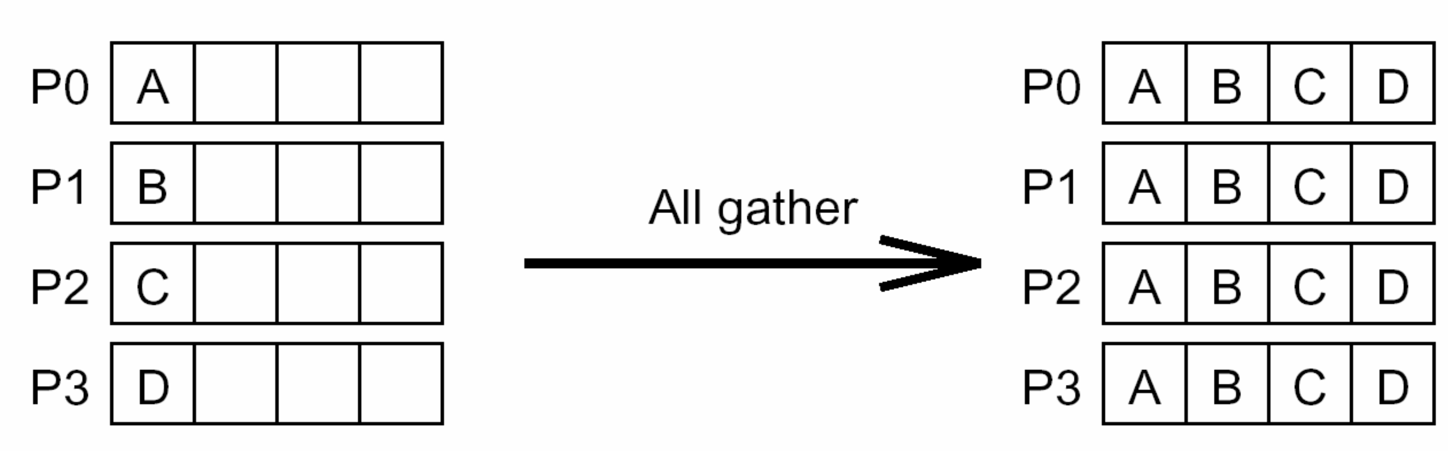
Revision




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

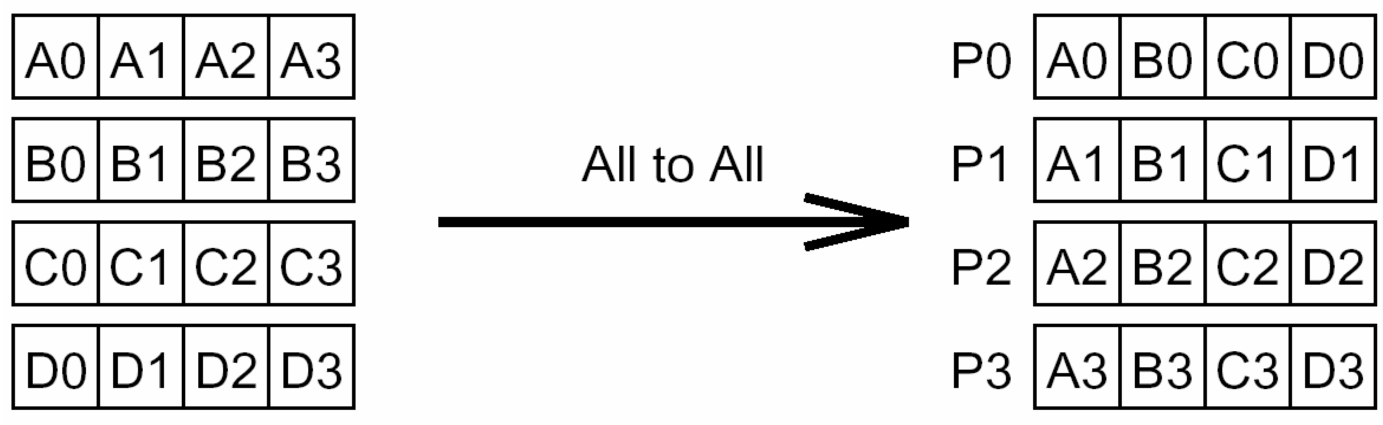
```
int MPI_Gather(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, void *recvbuf, int recvcount,
        MPI_Datatype recvdatatype, int target, MPI_Comm comm)
```

Revision



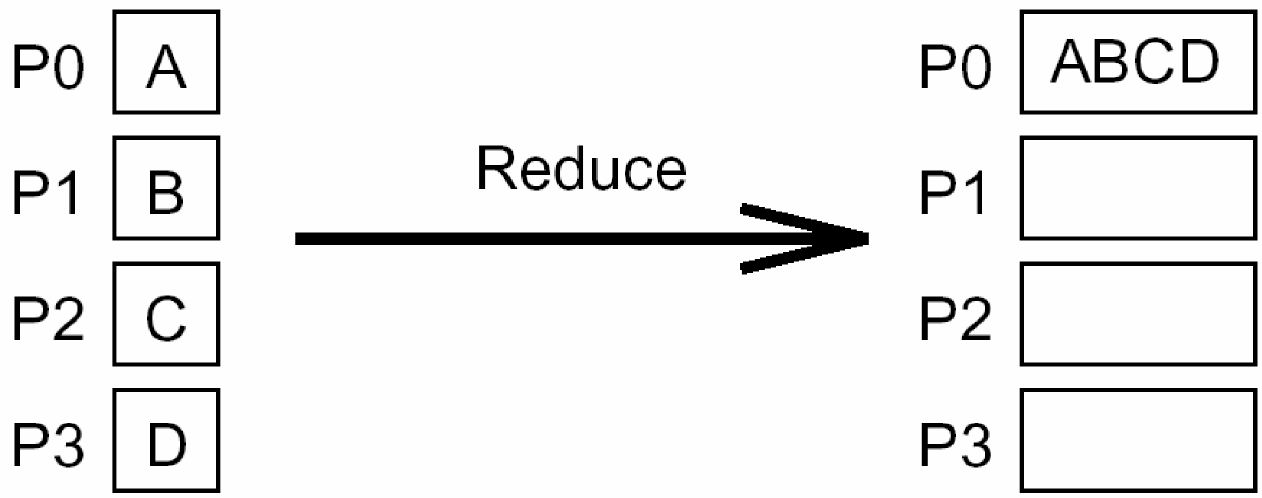

```
int MPI_Allgather(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, void *recvbuf, int recvcount,
        MPI_Datatype recvdatatype, MPI_Comm comm)
```

Revision



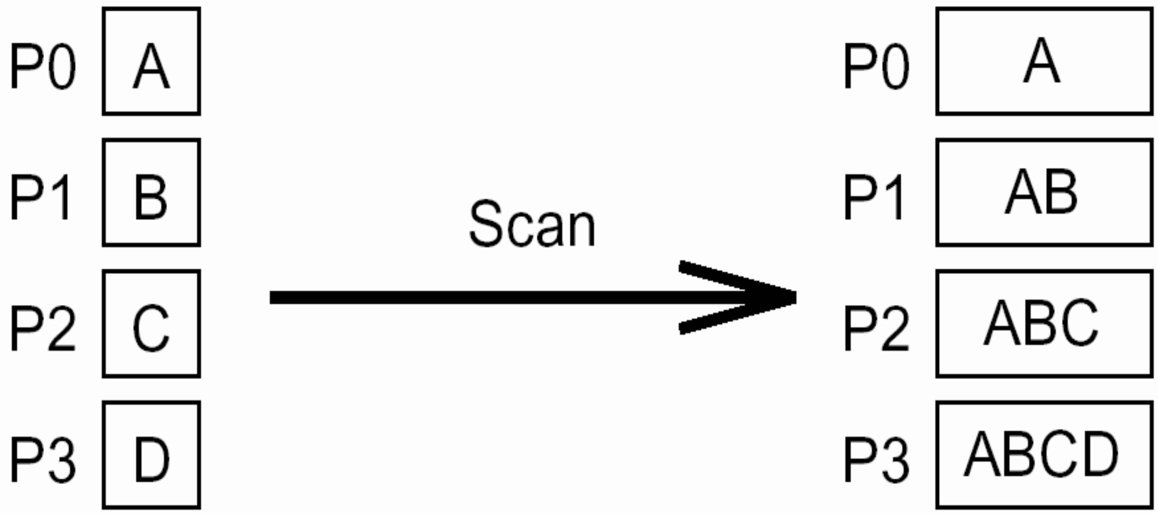

```
int MPI_Alltoall(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, void *recvbuf, int recvcount,
        MPI_Datatype recvdatatype, MPI_Comm comm)
```

Revision




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

Revision




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

MPI Functions

Operation	MPI Name
One-to-all broadcast	MPI_Bcast
All-to-one reduction	MPI_Reduce
All-to-all broadcast	MPI_Allgather
All-to-all reduction	MPI_Reduce_scatter
All-reduce	MPI_Allreduce
Gather	MPI_Gather
Scatter	MPI_Scatter
All-to-all personalized	MPI_Alltoall